CSPro User's Guide

Version 7.7.2

International Programs
Population Division
U.S. Census Bureau

4600 Silver Hill Road
Washington, DC 20233

cspro@lists.census.gov

Table of Contents

Table of Contents 1
CSPro User's Guide 22
The CSPro System 22
What is CSPro? 22
CSPro Capabilities 23
Release History 24
What's New in CSPro? 26
What's New in CSPro? 26
What's New in CSPro 7.0? 26
What's New in CSPro 7.1? 28
What's New in CSPro 7.2? 30
What's New in CSPro 7.3? 31
What's New in CSPro 7.4? 33
What's New in CSPro 7.5? 36
What's New in CSPro 7.6? 37
What's New in CSPro 7.7? 39
CSPro Applications 42
Data Entry Applications 42
Batch Edit Applications 42
Tabulation Applications 43
Data Dictionary 43
Forms Design 44
Tool List 44
CSPro General Concepts 47
CSPro Initial Screen Layout 47
Trees in CSPro 47
Windows 48
CSPro Workspaces 48
CSPro General Functionality 50
CSEntry Data Source File Types 50
Connection String 53
Encrypted Data 54
Unicode Primer 55
Synchronization Overview 57
Paradata 62
Multiple Language Applications 63
Mapping 63
How To ... 64
Create a CSPro Application 64
Open an Existing Application 64
Change the View 65
Designer Font Preferences 65
Change Windows 66
Add Files to an Application 66
Drop Files from an Application 66
Change the Print Page Setup 67
Print All or Part of a Document 67
Save an Application 67

Close an Application 67
Save an Application with a New Name 68
Specify Application File Names 68
Pack an Application 69
CSPro Settings 69
Get Help 69
Collaborating on CSPro Development 70
Data Dictionary Module 72
Introduction to Data Dictionary 72
Organization 73
Questionnaire and Dictionary Organization 73
Data File Type Structure 74
Dictionary Hierarchy 75
Dictionary Concepts 77
General 77
Labels 77
Names 77
Notes 77
Levels 78
Level Description 78
Level Properties 78
Records 80
Record Description 80
Record Properties 80
Record Type 81
Required Record 81
Maximum Number 81
Items 83
Item Description 83
Identification Items 83
Subitems 83
Item Properties 84
Starting Position 85
Length 85
Data Type 85
Occurrences 85
Decimal Places 86
Decimal Character 86
Zero Fill 86
Value Sets 88
Value Sets Description 88
Value Set Properties 89
Value Set Images 90
Values 91
Value Description 91
Value Properties 91
Relations 93
Relation Description 93
Relation Properties 93

Data Dictionary Application 94
Creating a Dictionary for a New File 94
Creating a Dictionary for an Existing File 94
Data Dictionary Screen Layout 95
Dictionary Property Panel 96
Data Dictionary Tree 97
Relative and Absolute Mode 98
Dictionary Types 98
Reconciling Dictionary Changes 99
How to ... 100
Open an Existing Dictionary Application 100
Move Around a Dictionary 100
View the Dictionary Layout 100
Add Dictionary Elements 101
Modify Dictionary Elements 101
Add or Modify Levels 101
Add or Modify Records 102
Add or Modify Items 102
Add or Modify Value Sets 102
Linked Value Sets 103
Generate Numeric Value Set 104
Add or Modify Values 104
Undo and Redo Changes 105
Select Several Dictionary Elements 105
Insert Dictionary Elements 105
Delete Dictionary Elements 106
Move Dictionary Elements 106
Find Dictionary Elements 106
Document Dictionary Elements 107
Convert Items to Subitems 107
Select Relative or Absolute Positioning 107
Create Dictionary with No Record Types 107
Occurrence Labels 107
Add or Modify Relations 108
Print the Dictionary File 108
Save Dictionary as New File 108
Multiple Language Dictionaries 109
Dictionary Macros 109
Dictionary Analysis 111
Dictionary Security Options 111
The CSPro Language 113
Introduction to CSPro Language 113
Data Requirements 113
CSPro Program Structure 113
Programming Standards 115
Code Folding 116
Debugging CSPro Applications 116
Declaration Section 117
Compiler Mode 117
Variables 117
Alias Statement 117

Ensure Statement 118
User-Defined Functions 119
Array Object 119
Audio Object 120
Document Object 121
File Object 122
Freq Object 123
Geometry Object 124
HashMap Object 125
Image Object 126
List Object 127
Map Object 128
Pff Object 130
SystemApp Object 131
ValueSet Object 132
Procedural Sections 134
Statements 134
Proc Statement 134
PreProc Statement 135
OnFocus Statement 136
OnOccChange Statement 136
KillFocus Statement 137
PostProc Statement 137
Logic 139
View Logic 139
Create and Edit Logic 139
Find and Replace Logic 140
Compile Logic / An Application 140
Symbol Analysis 141
Language Elements 142
Delimiters 142
Comments 142
Logic Preprocessor 142
Variables and Constants 145
Data Items 145
This Item ($) 145
Subscripts 145
Numbers 146
Text Strings 146
Expressions 148
Expressions 148
Substring Expressions 148
Boolean Values 149
Special Values 150
Refused Value 150
Operators 152
Operators 152
In Operator 152
Has Operator 153
If and Only If Operator 154

Operator Precedence 154
And/Or Truth Table 155
Files 156
External Files 156
Lookup Files 156
Working Storage File 157
Miscellaneous 158
Display Dimensions 158
Data Entry Module 159
Introduction to Data Entry 159
Data Entry Application 160
General Data Entry Concepts 160
Data Entry Philosophies 160
Skip Issues 160
Errors at Data Entry 161
Adding Logic 161
CSPro Data Entry Concepts 162
Operator vs. System Controlled 162
Data Entry Path 162
Data Entry Elements 163
Blocks 164
Issues to Consider When Designing a Form 166
Tips on Reviewing Data Entry Applications 167
Create a Data Entry Application 169
Create a New Data Entry Application 169
Generate Default Data Entry Forms 170
Drag Options Menu 170
Data Entry Forms Screen Layout 172
Forms Tree 173
Run a Data Entry Application 174
Run Production Data Entry 174
Generate Binary Data Entry Application 177
Change Data Entry Characteristics 179
Change the Order of Entry 179
Change Data Entry Options 179
Data Source Options 181
Simple Synchronizations 181
Display the Case Listing on a Map 183
Change Default Text Font 184
Change Field Colors 184
Change Field Font 185
Change Error Sound 185
Forms Designer 186
Introduction to Forms Design 186
Add Things to a Form 187
Add a Form 187
Add Fields to a Form 187
Add a Roster to a Form 187
Add Things to a Roster 188

Add Text to a Form 188
Add Lines or Boxes to a Form 189
Modify Things in a Form 190
Selecting Items 190
Move Things 190
Align Things 191
Cut, Copy, or Paste Things 192
Resize and Reposition Things in a Roster 192
Join and Split Roster Columns 193
Delete Form Elements 193
Matching the Application to the Data Dictionary 193
Change Form Properties 194
Change Forms File Properties 194
Change Level Properties 194
Change Form Properties 194
Change Block Properties 195
Change Field Properties 196
Change Field Properties for Multiple Fields 198
Change Keyboard Input 199
Change Roster Properties 200
Change Report Properties 201
Change Column Heading Properties 201
Change Row Heading Properties 202
Change Text Properties 203
Data Entry Editing 205
Introduction to Data Entry Editing 205
Editing Concepts 206
Type of Edits in Data Entry 206
Structure Edits 207
Consistency Edits 207
Checking Errors 208
Writing Logic 209
Data Entry Logic Screen Layout 209
Moving Around a Logic Application 210
Order of Executing Data Entry Events 210
Sequence Dictated by Designer 212
OnKey Character Map 212
User and Configuration Settings 212
Run as Batch 213
CAPI Data Entry 215
Introduction to CAPI 215
CAPI Features 215
Capture Types 216
Capture Types 216
Barcode Capture Type 217
Check Box 217
Date Capture Type 218
Drop Down / Combo Box 218
Number Pad 219
Radio Button 219

Slider 220
Text Box 220
Toggle Button 221
Multimedia 222
Multimedia Features 222
Media Store 222
Camera 222
Signature 223
CAPI Strategies 224
Forms 224
Fields 224
Questions 224
Organization of the Instrument 225
Using Multiple Languages 225
Breaking Off the Interview 225
Coming Back Later 226
Prefilling Values 226
How to ... 227
Create a New CAPI Application 227
Define Languages 227
Organize Forms 228
Enter Question Text 228
Create Fills In Questions 228
Edit Question Text Styles 230
Change Formatting 230
Add Images 231
Insert Link In Question Text 231
HTML Question Text 232
Question Text Macros 232
Resource Folders 233
Use Multiple Languages 233
Create Conditional Questions 233
Structure Movement 234
Create Helps for Fields 234
Show Values for Selection 235
Handle Multiple Answers 235
Choose Topic Sections 235
Create General Helps 236
Test Application 237
Network Data Entry 238
Network Data Entry 238
Android Data Entry 240
Android Development 240
Android Limitations 240
Android Navigation 241
CSEntry Settings Modification 246
Batch Editing Applications 248
Introduction to Batch Editing 248
Create a Batch Edit Application 249
Create a New Batch Edit Application 249

Batch Application Screen Layout 250
Batch Edit Tree 252
Run a Batch Edit Application 252
Order of Editing 254
Order of Executing Batch Edit Events 254
Batch Edit Order 254
Change Edit Order 255
Correcting Errors 256
Methods of Correcting Data 256
Guidelines for Correcting Data 256
Imputation 257
Static Imputation 257
Dynamic Imputation (Hot Deck) 258
DeckArrays 259
DeckArray Leftover Rows 261
Types of Edits in Batch Editing 262
How to ... 265
Manipulate Automatic Reports 265
Create a Specialized Report 265
Use Hot Decks 266
Initialize Hot Decks in Program Logic 266
Initialize Hot Decks Using Saved Arrays 267
Interpret Reports 268
Run Production Batch Edits 269
Steps in Developing a Batch Editing Program 272
General Batch Editing Issues 272
Review Edit Specifications 272
Define Coding Standards 272
Code Edits of Individual Data Items 273
Develop Comprehensive Test File 273
Test CSPro Program 273
Re-Test with Live Data 273
Begin Production Editing 274
Tabulation Applications 275
Introduction to Tabulation 275
Parts of a Table 275
Design/Data View Contents 276
Additional Contents for Print View 277
Additional Table Terms 277
Tabulation Application Tree 277
Common Uses of Tabulation 279
Capabilities of Tabulation 280
Cross Tabulations 280
Tabulate Counts or Percents 280
Tabulate Values and/or Weights 281
Produce Summary Statistics 281
Restrict a Universe 282
Format Tables for Printing 282
Load and Save Formatting Preferences 282
Produce Tables by Area 282
Save Tabulations in Different Formats 282

Copy Table to Other Formats 283
Copy and Paste Table Specification 283
Create Multiple Subtables 283
Change Unit of Tabulation 284
Tally Items from Related Records 285
Creating Tables 286
Create a New Tabulation Application 286
Create a Table 287
Create Tables with Multiple Variables 289
Implications of Data Dictionary Value Sets 290
Tabulate Items with Multiple Occurrences 291
Tally Attributes for a Variable 292
Tally Attributes for a Table 296
Add, Insert, and Delete Tables 298
Add Table 299
Insert Table 299
Delete Table 299
Move Between Tables 299
Run a Tabulation Application 299
Renaming Tables and Table Applications 300
How To ... 302
Add a Variable to a Tabulation 302
Remove a Variable from a Tabulation 303
Define a Universe for a Table 303
Add Weights to a Table 305
Tabulate Values Instead of Frequencies 305
Include/Exclude Special Values in a Variable 306
Hide or Change the Position of the Total 307
Add Percents to a Table 307
Add Summary Statistics to a Table 308
Include/Exclude Tables from Run 309
Debug Table Totals 309
Formatting Tables 311
Formats for a Part of a Table 311
Formatting Row, Column, or Cell Data 313
Formats for a Table 314
Formats for an Application 315
Formats for Printing 317
Views of Tables 319
How To ... 321
Customize Table Text 321
Hide or Show a Row or Column 321
Hide Rows Containing All Zeros 323
Change the Number of Decimal Places Displayed 323
Add a Footnote (Pagenote or Footnote) 324
Add Header/Footer Text to a Table 324
Add a Subtitle 326
Add Stub Leadering 326
Add Borders 327
Add Reader Breaks 327
Change the Way Numbers are Displayed 328

Change the Automatically Generated Text 329
Change Fonts or Colors 330
Change Indentation or Alignment 331
Add Borders to Cells 331
Make Captions Span Data Cells 332
Reset Format of Table Item to Default 333
Change the Repeating of Boxheads 333
Change Stub Column Position 334
Creating Tables by Geographic Area 336
Area Processing 336
Create an Area Names File 336
Area Dialog Box 338
Area Captions 339
Custom Consolidation 341
How To ... 344
Display Results for One Geographic Area 344
Tabulate only Certain Levels of Geography 344
Printing Tables 346
Using Print Preview 346
Navigating Between Pages, Tables, and Areas 346
Viewing Multiple and Facing Pages 347
Modifying Row and Column Spacing for Printing 348
Print Preview Options 349
Print Setup 351
Sending Tables to the Printer 352
How To ... 354
Undo or Reset Changes in Print Preview 354
Print Only Selected Tables or Pages 354
Tabulation Preferences 355
Preferences and Default Formats 355
Modifying Preferences 356
Loading and Saving Preferences 356
How To ... 358
Share the Same Format on Multiple Computers 358
Saving and Copying Table Data 359
Save Tables for the Table Viewer 359
Saving Tables as Text, HTML or Rich Text 359
Select and Copy Table Data to Other Applications 360
How To ... 362
Distribute Finished Tables to Other Users 362
Copy Table Data to a Spreadsheet or Word Processor 362
Prepare Tables for Posting to the Web 362
Table Post Calculation 363
Introduction to Table Post Calculation 363
Adding Rows and Columns For Post Calculation 363
Post Calculation For Individual Cells 364
Post Calculation For Rows, Columns and Ranges 367
Row and Column Indexing for Post Calculation 368
Run Production Tabulations 371

Introduction to Production Tabulations 371
Run All in Batch 371
Run in Parts 373
Introduction to Run in Parts 373
Run Tabulate Interactively 373
Run Tabulate in Batch 374
Run Consolidate Interactively 375
Run Consolidate in Batch 376
Run Format Interactively 377
Run Format in Batch 378
Advanced Table Topics 380
Using Subtables 380
Changing the Unit of Tabulation 381
Table Logic (tablogic) 382
Tabulations Using Relations 385
How To ... 387
Tabulate Items in Relations 387
Table Tips and Tricks 389
Add Subtotals to a Table 389
Tabulate Categories With Disjoint Values 390
Hiding Rows and Columns in Subgroupings 391
Recodes in Tables Using Value Sets and Subtables 395
Use Expressions in Universe and Value Tallied 399
CSPro Statements and Functions 400
Statement Format Symbols 400
Alphabetical List of Functions and Statements 400
List of Reserved Words 407
Deprecated Features 409
Declaration Statements 411
Numeric Statement 411
String Statement 411
Config Statement 412
Alpha Statement 412
Relation Statement 413
Function Named Arguments 414
Function Statement 414
Optional Function Parameters 416
Passing Function Arguments by Reference 417
Additional Examples of User-Defined Functions 419
Logic Objects and Dot Notation 420
Array Object 422
Array Statement 422
Array.Length Function 423
Array.Clear Function 424
Audio Object 425
Audio Statement 425
Audio.Load Function 426
Audio.Save Function 427
Audio.Play Function 427
Audio.RecordInteractive Function 428

Audio.Record Function 429
Audio.Stop Function 430
Audio.Concat Function 431
Audio.Length Function 432
Audio.Clear Function 433
Barcode and QR Codes 435
Barcode Namespace 435
Barcode.Read Function 435
Document Object 437
Document Statement 437
Document.Load Function 437
Document.Save Function 437
Document.View Function 438
Document.Clear Function 438
File Object 440
File Statement 440
File.Open Function 440
File.Close Function 441
File.Read Function 441
File.Write Function 442
Freq Object 444
Freq Statement (Named) 444
Frequency Formatting Options 446
Freq.Tally Function 448
Freq.Clear Function 449
Freq.Save Function 449
Freq.View Function 450
Geometry Object 451
Geometry Statement 451
Geometry.Area Function 451
Geometry.Clear Function 452
Geometry.Load Function 452
Geometry.MaxLatitude Function 453
Geometry.MaxLongitude Function 454
Geometry.MinLatitude Function 454
Geometry.MinLongitude Function 455
Geometry.Perimeter Function 456
Geometry.Save Function 456
Geometry.TracePolygon Function 457
Geometry.WalkPolygon Function 458
Geometry.GetProperty Function 459
Geometry.SetProperty Function 460
HashMap Object 462
HashMap Statement 462
HashMap.GetKeys Function 463
HashMap.Contains Function 464
HashMap.Length Function 465
HashMap.Remove Function 466
HashMap.Clear Function 467
Image Object 469

Image Statement 469
Image.Load Function 469
Image.Save Function 470
Image.Width Function 471
Image.Height Function 472
Image.Resample Function 472
Image.TakePhoto Function 474
Image.CaptureSignature Function 475
Image.View Function 476
Image.Clear Function 477
List Object 478
List Statement 478
List.Add Function 478
List.Insert Function 479
List.Seek Function 479
List.Remove Function 480
List.RemoveDuplicates Function 480
List.RemoveIn Function 481
List.Clear Function 482
List.Length Function 482
List.Show Function 483
List.Sort Function 483
Map Object 485
Map Statement 485
Map.Show Function 485
Map.Hide Function 486
Map.SetTitle Function 487
Map.SetBaseMap Function 487
Map.ZoomTo Function 488
Map.ShowCurrentLocation Function 489
Map.SaveSnapshot Function 490
Map.SetOnClick Function 491
Map.GetLastClickLatitude Function 492
Map.GetLastClickLongitude Function 493
Map.AddMarker Function 494
Map.RemoveMarker Function 495
Map.ClearMarkers Function 496
Map.SetMarkerImage Function 497
Map.SetMarkerText Function 497
Map.SetMarkerDescription Function 498
Map.SetMarkerOnClick Function 499
Map.SetMarkerOnClickInfoWindow Function 501
Map.SetMarkerOnDrag Function 503
Map.SetMarkerLocation Function 504
Map.GetMarkerLatitude Function 505
Map.GetMarkerLongitude Function 506
Map.AddGeometry Function 507
Map.RemoveGeometry Function 508
Map.ClearGeometry Function 509
Map.AddTextButton Function 510
Map.AddImageButton Function 511

Map.RemoveButton Function 512
Map.ClearButtons Function 513
Base Map Specification 513
Offline Maps 514
Path 516
Path Namespace 516
Path.Concat Function 516
Path.GetDirectoryName Function 517
Path.GetExtension Function 517
Path.GetFileName Function 518
Path.GetFileNameWithoutExtension Function 519
Path.GetRelativePath Function 520
Path.SelectFile Function 521
Pff Object 523
Pff Statement 523
Pff.Load Function 523
Pff.Save Function 524
Pff.GetProperty Function 524
Pff.SetProperty Function 525
Pff.Exec Function 526
SystemApp Object 528
SystemApp Statement 528
SystemApp.SetArgument Function 528
SystemApp.Exec Function 529
SystemApp.GetResult Function 530
SystemApp.Clear Function 530
ValueSet Object 532
ValueSet Statement 532
ValueSet.Add Function 532
ValueSet.Remove Function 533
ValueSet.Clear Function 534
ValueSet.Length Function 534
ValueSet.Randomize Function 535
ValueSet.Show Function 536
ValueSet.Sort Function 536
Program Control Statements 538
Break Statement 538
Do Statement 538
Exit Statement 539
For Statement 540
ForCase Statement 541
For (Dictionary) Statement 541
If Statement 542
Next Statement 543
Universe Statement 543
When Statement 544
While Statement 545
Assignment Statements 546
Assignment Statement 546
Recode Statement 546

Recode Statement (Deprecated) 547
Impute Function 550
SetValue Function 552
GetValue Function 553
Data Entry Statements and Functions 554
Accept Function 554
Advance Statement 555
Ask Statement 555
ChangeKeyboard Function 556
Connection Function 556
DeMode Function 557
Display Orientation 557
Display Orientation 557
SetOrientation Function 558
EditNote Function 558
EndLevel Statement 559
EndGroup Statement 559
Enter Statement 560
GetCaptureType Function 560
GetCaseLabel Function 561
GetDeviceID Function 561
GetImage Function 562
GetNote Function 563
GetOperatorId Function 563
GetOS Function 564
GetRecord Function 564
GetUserName Function 565
GPS Function 565
Interactive GPS Modes 567
HideOcc Function 568
Highlighted Function 569
InAdvance Function 569
IsPartial Function 570
IsVerified Function 570
Move Statement 571
NoInput Statement 572
OnChangeLanguage Global Function 572
OnChar Global Function 573
OnKey Global Function 574
OnStop Global Function 580
OnSystemMessage Global Function 581
Prompt Function 582
Protect Function 583
PutNote Function 583
RandomizeVS Function 584
Reenter Statement 584
SavePartial Function 585
SelCase Function 586
Set Attributes Statement 587
Set Behavior CanEnter Statement 588
SetCapturePos Function 589
SetCaptureType Function 590

SetCaseLabel Function 590
Set ErrMsg Statement 591
SetFont Function 592
SetOperatorId Function 593
Show Function 594
ShowArray Function 595
ShowOcc Function 596
Skip Statement 596
Userbar Function 598
VisualValue Function 601
Batch Edit Statements 602
EndCase Statement 602
Export Statement 602
Freq Statement (Unnamed) 603
GetDeck Function 606
PutDeck Function 607
Set Behavior Export Statement 607
SetOutput Function 607
Skip Case Statement 608
Numeric Functions 609
ABS Function 609
CMCode Function 609
CountNonSpecial Function 610
Exp Function 611
High Function 611
Inc Function 612
Int Function 612
Log Function 613
Low Function 613
Random Function 613
RandomIn Function 614
Round Function 614
Seed Function 615
Sqrt Function 615
Set Behavior SpecialValues Statement 616
String Functions 617
Compare Function 617
Concat Function 617
String Concatenation Operator 618
Edit Function 618
GetBuffer Function 619
Length Function 619
MakeText Function 620
Message Formatting Options 621
Pos Function 623
PosChar Function 624
RegExMatch Function 624
Replace Function 625
StartsWith Function 626
Strip Function 626
ToLower Function 627

ToNumber Function 627
ToUpper Function 628
Multiple Occurrence Functions 629
Average Function 629
Count Function 629
CurOcc Function 630
Delete Function 630
GetOccLabel Function 631
Insert Function 632
Max Function 633
MaxOcc Function 633
Min Function 634
NOccurs Function 634
Seek Function 635
SeekMax Function 635
SeekMin Function 635
SetOccLabel Function 636
SOccurs Function 637
Sort Function 637
Sum Function 638
Swap Function 638
TotOcc Function 639
General Functions 640
Compress Function 640
Decompress Function 640
Diagnostics Function 641
Encode Function 641
ErrMsg Function 642
ExecSystem Function (Desktop) 645
ExecSystem Function (Mobile) 646
ExecPFF Function 647
GetProperty Function 648
GetLabel Function 650
GetLanguage Function 651
GetSymbol Function 651
GetValueLabel Function 652
Hash Function 652
HtmlDialog Function 653
InValueSet Function 655
Invoke Function 655
IsChecked Function 657
LoadSetting Function 657
LogText Function 658
MaxValue Function 659
MinValue Function 659
Paradata Function 660
PathConcat Function 661
PathName Function 661
Report Function (Deprecated) 662
SaveSetting Function 663
SetLanguage Function 663

SetProperty Function 664
SetReportData Function (Deprecated) 666
SetValueSet Function 667
SetValueSet Function With Arrays 668
SetValueSets Function 669
Special Function 670
SqlQuery Function 670
Stop Function 673
SysParm Function 674
Tr Function 674
Trace Function 675
UUID Function 677
View Function 677
Warning Function 678
Date and Time Functions 680
AdjLBA Function 680
AdjLBI Function 680
AdjUBA Function 681
AdjUBI Function 681
DateAdd Function 682
DateDiff Function 682
DateValid Function 683
PublishDate Function 684
SetLB Function 684
SetUB Function 685
SysDate Function 686
SysTime Function 686
Timestamp Function 687
TimeString Function 688
External File Functions 689
Clear Function 689
Close Function 689
CountCases Function 690
CurrentKey Function 690
DelCase Function 691
DirCreate Function 692
DirDelete Function 692
DirExist Function 692
DirList Function 693
FileConcat Function 694
FileCopy Function 695
FileCreate Function 695
FileDelete Function 696
FileEmpty Function 696
FileExist Function 697
FileName Function 697
FileRead Function 698
FileRename Function 698
FileSize Function 699
FileTime Function 699
FileWrite Function 700

Find Function 701
Key Function 701
KeyList Function 702
LoadCase Function 703
Locate Function 704
NMembers Function 704
Open Function 705
Retrieve Function 706
Set Access Statement 706
SetFile Function 707
Set First Statement 708
Set Last Statement 709
Write Function 709
WriteCase Function 710
Synchronization Functions 712
SyncConnect Function 712
SyncConnect Function (CSWeb) 712
SyncConnect Function (Bluetooth) 713
SyncConnect Function (Dropbox) 714
SyncConnect Function (LocalDropbox) 714
SyncConnect Function (LocalFiles) 715
SyncConnect Function (FTP) 715
SyncServer Function 716
SyncDisconnect Function 717
SyncData Function 718
SyncFile Function 719
SyncMessage Function 720
SyncParadata Function 721
SyncApp Function 722
SyncTime Function 723
GetBluetoothName Function 724
SetBluetoothName Function 725
Export Attributes 726
Export Attributes 726
Export Attribute: decimal-mark 727
Export Attribute: dictionary 727
Export Attribute: encoding 727
Export Attribute: header 727
Export Attribute: mapped-special-values 728
Export Attribute: record 728
Export Attribute: syntax-file 729
Export Attribute: type 729
Export Attribute: value 730
Templated Reporting System 731
Templated Reports 731
Report.Save Function 732
Report.View Function 732
Report.Write Function 733
Templated Reports (Deprecated) 734
Mustache Templated Reports (Deprecated) 736
HTML and JavaScript Integration 738

HTML in CSPro 738
HTML Dialog Boxes 739
JavaScript Interface 740
JSON Format 742
Appendix 744
Appendix A - Installation 744
Hardware and Software Requirements 744
Installing CSPro 744
Installing a Newer Version 744
Uninstalling CSPro 745
Installing Data Entry Applications 745
Appendix B - Keys Summary 746
Data Dictionary Keys 746
Data Entry Keys 747
Batch Edit Keys 748
Tabulation Keys 749
Appendix C - Menu Summary 752
CSPro Menu 752
Data Dictionary Menu 752
Data Entry Menu 754
Batch Editing Menu 755
Tabulation Menu 756
Appendix D - Toolbar Summary 759
CSPro Toolbar 759
Data Dictionary Toolbar 759
Data Entry Toolbar 760
Batch Editing Toolbar 761
Tabulation Toolbar 761
Appendix E - Application Properties 763
Application Properties 763
Paradata Properties 763
Mapping Properties 765
Advanced Features Properties 765
Appendix F - Converting Within IMPS or ISSA 767
Converting a Data Dictionary 767
Converting within IMPS 767
Converting within ISSA 768
Converting an IMPS Data Entry Application 768
Converting an ISSA Data Entry Application 768
Appendix G - Errors in Censuses and Surveys 770
Nature of Census and Survey Data 770
Errors in Censuses and Surveys 770
Appendix H - File Types 773
File Types 773
Importing Data to CSPro Format 774
Locking Application Files 776
Temporary Data File 777
CSPro DB File Format 777
Files Description 780

Data Dictionary File (.dcf) 780
Data Entry Application File (.ent) 780
Forms File (.fmf) 780
Logic File (.apc) 780
Message File (.mgf) 781
Question File (.qsf) 782
Application Properties File (.csprops) 784
Program Information File (.pff) 785
Binary Data Entry Application File (.pen) 785
Index File (.csidx) 785
Notes File (.csnot) 786
Status File (.sts) 786
Paradata Log File (.cslog) 787
Operator Statistics File (.log) 787
Listing File 788
Frequencies File 788
Batch Edit Application File (.bch) 789
Edit Order File (.ord) 789
Saved Arrays File (.sva) 789
Tabulation Application File (.xtb) 790
Table Specifications File (.xts) 790
Tables File (.tbw) 790
Area Names File (.anm) 790
Table Matrices File (.tab) 790
Table Matrices Index File (.tabidx) 791
Frequency Specification File (.fqf) 791

CSPro User's Guide

The CSPro System

What is CSPro?

The **C**ensus and **S**urvey **Pro**cessing System (CSPro) is a software package for entering, editing, tabulating, and disseminating data from censuses and surveys. Using CSEntry, the data collection component, you can run a data entry operation in the Windows and Android environments. CSPro combines the features of the Integrated Microcomputer Processing System (IMPS) and the Integrated System for Survey Analysis (ISSA).

CSPro runs under Windows Vista, 7, 8, 10, or 11. It does not run under other operating systems such as Linux or Mac OS. It is a public domain product, so it can be used and distributed at no cost.

CSPro can be used to process data from censuses and surveys, both small and large. Typical subject areas include:

- Housing and Population
- Demographic Characteristics
- Health and Nutrition
- Agriculture
- Labor Force
- Business Establishments
- Education
- Living Standards
- Energy
- Immigration
- Household Income and Expenditure
- Community
- Institutional
- Post-Enumeration
- Vital Statistics

CSPro uses data dictionaries to provide a common description of each data file used. CSPro encodes data in UTF-8 format. CSPro provides tools to view data and text files, to view tables created by CSPro, to convert IMPS and ISSA data dictionaries to and from CSPro.

CSPro is not intended to provide database management capabilities. However, the data generated and/or manipulated by a CSPro application can be imported into a database system. While CSPro provides some tabulation capabilities, it is not intended to replace more sophisticated statistical analysis software such as R, SAS, SPSS, Stata, etc.

CSPro includes the following modules:

- Data Entry Applications
- Batch Edit Applications

- Tabulation Applications
- Tools

If you have never used CSPro before, you can refer to the Getting Started Guide, a tutorial that gives an overview of CSPro's capabilities. If you are a previous user, you may want to look at a list of this version's new features.

This section includes the following information:

- CSPro Capabilities
- CSPro Applications
- General Concepts
- General Functionality
- How to ...

CSPro Capabilities

Process Census or Survey Data

Given an existing data file, a user can develop a CSPro application that will examine the file for inconsistencies, structural defects, or other errors. CSPro permits the user to generate detailed reports on all errors found; the user may also create subfiles from the original data, and may use multiple look-up files during the validation and/or report-generation process.

Enter, Modify, and Verify Data

CSPro users can create data entry forms (screens) for data capture. The application designer has full control over form layout. CSPro supports rosters, consistency checks and skip patterns of unlimited complexity, user-defined messages and menus, multiple lookup files, and produces operator statistics.

Once a case has been completely entered, the operator can modify any part of the existing data and can add or remove information, as well (subject to application constraints).

CSPro supports both dependent and independent verification (double keying) to ensure the accuracy of the data entry operation. Using independent verification, operators can key data into separate data files and use CSPro utilities to compare them. Using dependent verification, operators can key data a second time and have CSPro immediately compare it to what was keyed the first time on a field-by-field basis.

Manipulate Data Files

CSPro permits the user to re-structure existing data files and to create subsets of data in separate files. New files can be created by merging two or more case-related files. Data files in software-specific formats may be created for import into spreadsheets and some statistical packages.

Tabulate Data

The user can create an application to produce frequency distributions or cross-tabulations using two to four variables. Results can be displayed either globally (for the totality of the data file) or according to one or more elements of the geographic hierarchy. Tabulations may show only percentages, or percentages in conjunction with counts. Data may be weighted or unweighted.

Use and Share External Files

When a data file is to be used by more than one person, a CSPro dictionary can be created and distributed among users of the data to facilitate access. Including multiple value sets for variables can cater to the different needs of individual users so that each user's requirements are met.

Examine Data Files

CSPro provides language elements that will permit the specification of logic to carry out a detailed examination of a data file. Elements of the file may be tested against other elements of the same file or against elements of one or more other files, and the user may generate reports showing the results of the examination. CSPro also provides a facility for comparing the contents of two data files. This utility will generate a detailed report to the user documenting any differences found.

Interactive Editing

CSPro language elements can be used to construct a series of tests to be carried out on a case-by-case basis using the CSEntry module. Whether adding a new case or modifying an existing case, CSPro instructions permits interactive editing and correction of data elements. If the user desires, a report on editing activity may be generated and saved for printing after the session is completed.

Examine Results of Editing

Whether the user is carrying out interactive or batch editing, the CSPro language permits the preparation of reports with detailed information on cases tested, errors found, and errors corrected. These reports are written to disk in text format and may be viewed (or printed) with any text viewer, such as CSPro's utility Text Viewer. They provide documentation of work carried out and permit analysis of types and frequency of errors.

Release History

CSPro was first released in 2000. Over the years, many new features have been added to the software, and in addition to the Windows desktop version, new versions have been released for Android. The following table shows the software's release history:

Version	First Release	Last Release	Releases	Notable Features
2.0	May 5, 2000	October 30, 2000	8	First version of CSPro
2.1	April 20, 2001	July 18, 2001	6	
2.2	November 7, 2001	December 18, 2002	16	
2.3	January 24, 2003	May 7, 2003	6	Improved form designer
2.4	August 6, 2003	April 30, 2004	8	
2.5	May 19, 2004	February 24, 2005	7	CAPI mode
2.6	June 13, 2005	September 29, 2005	7	Export tool
3.0	November 30, 2005	March 27, 2006	5	Improved tabulation system
3.1	May 17, 2006	August 29, 2006	3	
3.2	October 25, 2006	February 8, 2007	2	
3.3	May 11, 2007	June 4, 2008	3	
4.0	December 17, 2008	June 11, 2010	4	Windows CE (Pocket PC) version
4.1	May 5, 2011	December 12, 2011	2	Improved CAPI features
5.0	July 30, 2012	August 8, 2013	3	Unicode support
6.0	September 25, 2014	September 25, 2014	1	Android version

6.1	April 1, 2015	April 1, 2015	1	
6.2	November 26, 2015	November 26, 2015	1	
6.3	May 5, 2016	May 5, 2016	1	
7.0	May 8, 2017	June 29, 2017	3	CSPro DB and CSWeb
7.1	March 15, 2018	July 2, 2018	4	CSDeploy, paradata, and templated reports (deprecated)
7.2	February 22, 2019	April 1, 2019	2	Blocks
7.3	September 3, 2019	November 1, 2019	2	Mapping and logic objects
7.4	May 5, 2020	July 13, 2020	2	Encrypted and relational data files
7.5	October 8, 2020	December 11, 2020	2	Audio recording and new listing file formats
7.6	March 31, 2021	June 24, 2021	3	HTML question text and enhanced frequencies
7.7	January 12, 2022	June 17, 2022	3	HTML and JavaScript integration, logic-based reports, and vector geometry

See also: What's New in CSPro?

What's New in CSPro?

What's New in CSPro?

Every release of CSPro includes a few major features as well as the addition of some minor features and bug fixes. A general overview of the new features can be found in the *readme.txt* file located in the CSPro installation folder. However, for more recent versions of CSPro, a detailed list of new features is also available.

CSPro 7.x

- CSPro 7.7
- CSPro 7.6
- CSPro 7.5
- CSPro 7.4
- CSPro 7.3
- CSPro 7.2
- CSPro 7.1
- CSPro 7.0

See also: Release History

What's New in CSPro 7.0?

This is a detailed list of changes in CSPro 7.0. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

CSPro Designer

- You can resize a roster to its ideal size by right-clicking on it and selecting **AutoFit**.

Data Sources

- Introduction of the new **CSPro DB** and **none** data sources.
- Using the synchronization functions, CSPro DB data files can be synchronized with Dropbox and FTP servers on a case by case, rather than data file, basis.
- The creation of listing and operator statistics files can be disabled.

Data Entry

- ID items can be set to auto increment.
- Addition of several data entry options for mobile devices:
    - Automatically advance on selection.
    - Show field labels above question text.
    - Display error message numbers.
- Addition of several new PFF attributes:

    - CaseListingFilter: Filters the list of cases shown in the case listing.
    - ShowInApplicationListing: Specifies whether an application is shown in the list of applications.
    - OnExit: Automatically launches a CSPro application after executing the current one.
    - Lock: New lock flags **CaseListing** (does not show the case listing screen) and **Delete** (prevents the user from deleting cases).
- A case note can be defined, which will display in the case listing.
- Multiple operators can leave notes on the same field.
- On Windows, the dialog box for the `accept` (and `selcase`, `show`, and `showarray`) function has been redesigned to be more tablet friendly.
- Cases can now be verified on a case by case basis, rather than having to verify cases in file order.
- Portuguese language strings added to Android.

Batch

- A new PFF attribute, InputOrder, allows you to override the default order in which cases are processed in a data file.

Logic

- Introduction of new functions and statements:
    - `ask`: A new way to skip fields by programming using enablement conditions.
    - `compress` and `decompress`: A pair of functions to compress and decompress .zip files.
    - `config`: A way to initialize string variables to values defined on a local machine.
    - `diagnostics`: A troubleshooting function for getting information about the CSEntry build.
    - `forcase`: A loop that simplifies the processing of each case in an external data file.
    - `getcaselabel` and `setcaselabel`: Functions to set and retrieve a human readable label associated with a case.
    - `isverified`: A way to determine whether a case has been verified.
    - `keylist`: A function that returns a list of keys in a data file, including the input file in a data entry application.
    - `setoperatorid`: A way to set the operator ID in logic.
    - `syncconnect`, `syncdata`, `syncdisconnect`, `syncfile`, and `syncserver`: A set of functions to support the new synchronization functionality.
    - `timestamp`: A way to easily calculate time spans by working with UNIX times.
- The `skip` statement can be called without an argument, which will skip to the next field in the application.
- The `systime` function can take an optional argument to extract just the current time's hours, minutes, or seconds.
- The `ispartial` function can return the partial save status of cases in external dictionaries.
- Passing ID arguments to `writecase` is deprecated and the arguments are ignored.
- The `key` function can be called on the main input dictionary of an application.
- All settings can be removed by calling `savesetting` with the argument `clear`.
- The `select` options of an `errmsg` can now be used to skip to a field instead of only reentering a field.
- You can manipulate the location and order that external dictionary and batch input data sources are processed using: `set access`, `set first`, and `set last`.
- The `setfile`, `open`, and `close` functions can be used on the input dictionary in a data entry and batch application, allowing you to dynamically modify the data source being processed.

- The `execsystem` function on Android can send SMS text messages.

Tools

- A new tool, Data Viewer, allows for the viewing of data files in dictionary format (in lieu of using Text Viewer). The tool is also used to download data from a synchronization server.
- The **Export Data** tool can now output Stata exports in Unicode format. To open these exports, you need Stata version 14 or later.
- The **Excel to CSPro** tool is much improved, allowing you to create dictionaries from Excel data and to convert data from multiple worksheets.
- The **Concatenate Data** tool and the `fileconcat` function now combine case notes, partial save statuses, etc., and the tool can identify duplicate cases in files.

CSWeb

- CSWeb introduced as a convenient way to synchronize CSPro data files on a central server.

Other

- Introduction of a lightweight CSPro installer, with the CSPro examples now installed to the Documents folder.
- Introduction of a new custom help documentation system.
- Fixes to allow CSPro to fully work on Windows 10 systems.

See also: What's New in CSPro?

What's New in CSPro 7.1?

This is a detailed list of changes in CSPro 7.1. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

Paradata

- The introduction of a new system for collecting paradata during an application's run. The paradata events are saved to a log file with the extension .cslog. You can control what kinds of events are stored by modifying an application's paradata properties.
- A new logic function, `paradata`, that can be used to control or query the paradata log.
- A new tool, Paradata Viewer, that displays reports about the collected paradata.
- Another new tool, Paradata Concatenator, that combines multiple paradata logs into a single log.

Deployment

- A new tool, Deploy Application, that deploys one or multiple applications to a server (or a local file) so that they can be downloaded onto interviewer devices.
- Deployed applications can be downloaded using CSEntry on mobile devices.

Reports

- An interface for generating HTML-templated reports from case data as well as from results from queries into paradata and other data sets.
- New logic functions, `setreportdata` and `report`, for setting the inputs and creating the reports.

Messages

- Expanding multiple language application support, messages defined in the message file can be specified in multiple languages. These messages, along with string literals wrapped in the `tr` function, will be displayed in the application's current language.
- Applications can link to multiple external message files, which can include messages that override the default system messages.

CSPro Designer

- You can move a value set to the first position, to become the default one shown in a data entry application, by right-clicking on a value set and selecting **Make Primary Value Set**.
- If you right-click on a dictionary from the dictionary tree, the **View Data** option will open the last data file associated with that dictionary (as stored in the PFF file) within Data Viewer.

Data Entry

- A data source option that allows you to specify the number of minutes after which a case will automatically be partially saved.
- Question text fills can show the return values of user-defined functions.
- A data entry option, **Validate alpha fields**, instructs CSEntry to validate values entered into alphanumeric fields using an item's value set, as happens to numeric fields by default.
- A mobile option, **Display value set codes alongside labels**, displays a value set's codes in addition to the labels when displaying a field using a control other than a text box.

Logic

- An `errmsg`-like function, `warning`, that can be used for "soft check" error messages. These messages will not be displayed to the operator when advancing in the case (such as when resuming from a partial save).
- A multiple language application function, `tr`, for correctly displaying string literals or numbered messages in the application's current language.
- A new function, `countcases`, counts the number of cases in an external file that meet a certain criterion.
- New functions, `getproperty` and `setproperty`, for accessing and changing the properties of fields or the entire application.
- A new function, `logtext`, for writing user-defined messages to the paradata log file.
- A new function, `sqlquery`, that executes a query on a SQLite database and returns a result set in a variety of formats. Paradata logs are SQLite databases.
- A new message formatter, `"%v"`, that displays items in the format specified in the dictionary.
- The `timestamp` function returns the number of milliseconds in addition to the seconds.
- A new function, `timestring`, returns a custom formatted date and time string based on a given timestamp.
- The `sysdate` and `systime` functions can return a formatted local date or time from a UNIX time value returned by the `timestamp` function.
- The `uuid` function can return the UUID (internal key) associated with a case.
- The `loadcase` and `delcase` functions will give compiler warnings if the zero fill settings of the arguments do not match the settings of the external dictionary's IDs.
- The `writecase` and `delcase` functions now work in a `forcase` loop.
- Multiple string literals are automatically concatenated into a single string when separated by whitespace (e.g., `"A" "B"` is evaluated as `"AB"`).
- The `selcase`, `show`, and `showarray` functions can display title headings.
- The `setfont` function now sets the number pad's font differently from that of the other capture types.
- In a data entry application, when splitting items from a multiply occurring record into multiple groups, you no longer have to specify an occurrence number when referring to items on the split groups (assuming that you want

the value of the current occurrence).

Tools

- From Data Viewer's **Tools** menu, you can launch the **Export Data** and **Tabulate Frequencies** tools to directly process the cases of the open file.
- Data Viewer can display each value's label instead of only the code.
- Data Viewer now shows additional information about data files, including the synchronization history. The tool also has new filtering options and can show deleted or duplicate cases and can display the contents of a file in sorted key order.
- Excel to CSPro conversions can now be run via PFF files to streamline deployment conversions.
- The Excel to CSPro tool has two new options, one for conditionally running conversions based on the existence of file changes, and the other to allow you to update existing files rather than to always create new files.

Synchronization

- Faster Bluetooth transfer speeds.
- Fix to allow faster transfer speeds when calling `syncdata` or `syncfile` in the `GET` direction across all synchronization options (Bluetooth, CSWeb, Dropbox, and FTP).

See also: What's New in CSPro?

What's New in CSPro 7.2?

This is a detailed list of changes in CSPro 7.2. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

General

- A new concept, blocks, for grouping several fields into a related unit. Blocks can be used in various ways in logic and can have question text that appears when entering any of the fields in the block. On mobile devices, all fields on a block can be displayed on the same screen and the operator can enter values into these fields in any order.
- Dictionary names can be longer than 32 characters.

CSPro Designer

- Several reports about dictionary completeness can be generated using dictionary analysis.
- A variety of tasks can be performed and reports generated from CAPI question text using question text macros.

Data Entry

- The Windows desktop version of CSEntry has options to view cases with duplicate case IDs and to view deleted cases. Cases can be undeleted.
- A new option specifies that combo boxes should display only discrete (non-range) values.

Logic

- A function, `dirdelete`, deletes an empty directory or group of empty directories.
- A function, `getbluetoothname`, returns the name of the device as broadcast to other Bluetooth devices.
- A function, `getvaluelabel`, returns the label from the value set corresponding to an item's current value.
- A function, `regexmatch`, returns whether a string matches the regular expression.

- Additional variables (`array`, `list`, and `file`) can be declared locally. The value of each variable is reset upon entry to the `PROC` or `function` where it is declared.
- The `gps` function, when called with the `read` argument, returns the most accurate reading, not the most recent reading, after timing out.
- The `exit` statement takes an optional argument that simplifies setting a user-defined function's return value.
- Implicitly declaring variables (using set implicit) is no longer allowed.
- The `ensure` statement allows for the conditional creation of temporary numeric variables.
- Loops using `forcase` can be nested as long as they operate on different dictionaries.
- The `randomizevs` function can operate on a specific value set (instead of only a field's current value set).
- Arrays of more than three dimensions can be declared. The `length` function returns the size of an array's dimensions.
- Alpha and string arrays, and arrays of more than three dimensions, can be stored as saved arrays. The format of the saved arrays file has changed but CSPro will correctly read pre-7.2 .sva files.
- A new property can disable the updating of saved arrays files.
- DeckArrays can be used in data entry applications.
- User-defined functions accept `list` variables in the parameter list.
- Multiple sorting keys can be used in the `sort` function.
- Weeks can be used as time periods with the `dateadd` and `datediff` functions.
- Error messages using variable-numbered message numbers can include `denom` specifiers.

Synchronization

- The synchronization functionality from CSPro 6.x (.pnc files and the sync function) has been removed. Syncing files and data must be done with the approaches introduced in CSPro 7.0.

Messages

- Translations of some runtime messages added for Chinese, French, Portuguese, Russian, and Spanish. You can help translate additional messages or add messages in another language.

Android

- CSEntry is distributed in both 32-bit and 64-bit versions.
- When making calls to the `gps` function, if the device's GPS receiver is not enabled, the operator will be prompted to enable location services.
- The properties and behaviors of CSEntry's menus can be modified via logic. The way that hidden applications are shown is now controlled using this mechanism.
- Automatically launching an application, when only one exists on the device, is no longer the default behavior.
- The Start New Case option is now a floating green icon.

CSWeb

- Add many users at once by uploading a .csv file that specifies user details.

See also: What's New in CSPro?

What's New in CSPro 7.3?

This is a detailed list of changes in CSPro 7.3. For a more general overview, see the *readme.txt* file located in the CSPro

installation folder.

Mapping

- The introduction of a new mapping module for displaying maps and allowing users to interact with points. This feature is only available on Android.
- A simple version, the map case listing, replaces the text-based case listing with a map-based one.
- Advanced users can use the `map` object to create dynamic and responsive maps in logic.

CSPro Designer

- Replacement of the logic editor with a Scintilla-derived editor with better performance and functionality, including the ability to change the font size, view line numbers, go to a specific line, display autocomplete suggestions, and more. There are additional keyboard shortcuts to improve efficiency while writing logic, and the reference window has been reworked to provide more useful information.
- When right-clicking on the main dictionary of a batch application, an option allows you to view the output data in Data Viewer.

Data Entry

- A field's capture type is now considered when validating fields. Date fields must now contain a valid date and check box fields can now only contain values that exist in the value set.
- The Force Out-of-Range field property has been renamed to the Validation Method field property. This change allows you to specify out-of-range fields without operator confirmations directly from the form designer (rather than requiring logic).
- The **Validate alpha fields** data entry option has been removed. If an alphanumeric field has a capture type other than textbox, it will automatically be validated.
- A new PFF attribute, Key, combines the functionality of StartMode along with the ability to automatically fill in IDs for new cases.

Batch

- The default listing width has been increased from 80 to 120 and the space allocated to displaying denominators now supports 10-digit values.

Logic

- Introduction of logic objects with functions that can be called on the object using dot notation.
- New functions for arrays: `clear` and `length`.
- New functions for files: `close`, `open`, `read`, and `write`.
- The list object is now fully functional, with assignment and new functions: `add`, `clear`, `insert`, `length`, `remove`, `seek`, and `show`.
- Simplified creation of dynamic value sets with the new valueset object that has the functions: `add`, `clear`, `length`, `remove`, and `show`.
- Easier way to manipulate and run PFF files with the new pff object that has the functions: `exec`, `getproperty`, `load`, `save`, and `setproperty`.
- A new `map` object that allows the displaying of maps on Android that has the functions: `addImageButton`, `addMarker`, `addTextButton`, `clearButtons`, `clearMarkers`, `getLastClickLatitude`, `getLastClickLongitude`, `getMarkerLatitude`, `getMarkerLongitude`, `hide`, `removeButton`, `removeMarker`, `setBaseMap`, `setMarkerDescription`, `setMarkerImage`, `setMarkerLocation`, `setMarkerOnClick`, `setMarkerOnClickInfoWindow`, `setMarkerOnDrag`, `setMarkerText`, `setOnClick`, `setTitle`, `show`, `showCurrentLocation`, `zoomTo`.
- A new function, `pathconcat`, simplifies combining multiple strings into a single string representing a file path.

- A new function, `view`, launches the system's default viewer to display a file or website.
- A new message formatter, `"%l"`, that displays an item's label from the value set corresponding to an item's current value.
- A logical condition can be specified in `for` loops to restrict the universe of the loop. Additionally, the occurrence counting variable can be declared local to the `for` loop.
- Boolean values `true` and `false` translate to the numeric values 1 and 0.
- Several of the file functions take string lists as arguments, allowing mass concatenating (`fileconcat`), copying (`filecopy`), deleting (`filedelete`), and renaming (`filerename`) of files. The `filerename` function also accepts a folder as the new file name target.
- The `fileread` and `filewrite` functions also take string lists as arguments, allowing for the simple reading of all the lines of a file, or the writing out of multiple lines to a file.

Android

- Reading GPS coordinates now uses Google's Fused Location Provider, which makes it easier to capture coordinates while indoors.
- The way that hidden applications are shown has been restored to the behavior used up to CSPro 7.1.
- The add/insert/delete occurrence options on the case tree has been restored to the behavior used up to CSPro 7.1 (displaying in both operator- and system-controlled modes), though these options can be disabled via logic.

Tools

- The Map Viewer and Convert Shape to Map tools have been removed from the CSPro installation. Users still needing these tools can find them on www.csprousers.org.
- The Export Data tool now writes labels at the end of processing, so if an item's value set is changed during the run, labels from that value set will be exported.
- The Reformat Data and Sort Data tools allow you to output data to a different data source type than the input data. The Concatenate Data tool no longer requires the specification of the data source type of the output file.
- The Excel to CSPro PFF parameters have changed from InputData to Excel.
- The Paradata Concatenator PFF parameters have changed from InputData to InputParadata and OutputData to OutputParadata.
- The ExportXMLMetadata tool's HTML export properly outputs Unicode question text.

See also: What's New in CSPro?

What's New in CSPro 7.4?

This is a detailed list of changes in CSPro 7.4. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

General

- A new special value, refused, allows for processing refusals differently from other numeric responses. Refusals are defined in a dictionary value set, referenced in logic using `refused`, and are conditionally shown to operators. The entry of refusals can be overridden using the `OnRefused` function.
- A CSPro Settings dialog, accessible in the CSPro Designer as well as in CSEntry, allows users to clear saved synchronization and data source credentials.
- Licenses for open-source software that CSPro uses have been included and can be accessed using the Help -> About menu in the CSPro Designer.

Data Sources

- Change to the format of the CSPro DB data source to store data in a relational database format that can be queried using SQL. CSPro DB files created in version 7.4 cannot be read in earlier versions of CSPro.
- Introduction of the new Encrypted CSPro DB data source, allowing for file-based data encryption with the ability to cache passwords.
- Ability to include, as part of a PFF or in a `setfile` function call, a connection string that details how to open a data source.
- Dictionary options include the ability to disallow use of a dictionary to modify data in Data Viewer or to export data to another output format.

CSPro Designer

- Enhancements to the logic editor, including parentheses matching and improvements to the find and replace dialogs.
- The synchronization options have been updated. Synchronization can be removed from an application by unchecking the Enable Synchronization box. The options for downloading application files has been removed in favor of the new Update Installed Applications.
- Question text occurrence values can be harmonized with dictionary records using a new question text macro.

Data Entry

- When starting a new case, fields can be prefilled from values specified in a PFF's Parameters section.
- The compiler no longer issues an error when reentering a protected field.
- The Combo Box capture type can now be used on discrete numeric fields and alphanumeric fields.
- On Windows, the search bar on capture type popup windows is now hidden by default.
- Inserting a case now only allows for the addition of a single case at a time (regardless of the PFF's AutoAdd value).

Data Viewer

- The tool has been rewritten with additional features including multiple language support and additional ways to view case data.
- A case's notes can be extracted as another CSPro data file, which can then be used for exporting or other data operations.
- Ability to download or synchronize data from the local Dropbox folder or a local FTP server folder which can be faster than downloading data over the Internet.

Logic

- A new function, `ischecked`, returns whether a code is checked as part of a check box field's selections.
- A new function, `protect`, simplifies changing the protected property of a field.
- A new function, `filetime`, returns the last modification date/time of a file or a directory.
- A new function, `hash`, returns the hash value of a string.
- A new statement, `when`, executes a statement based on the value of one or more other variables.
- A new version of `recode`, sharing the syntax of `when`, with additional options for what the recoded value can be assigned to.
- A new function, `startswith`, that returns if a string begins with a specified prefix.
- A new function, `syncapp`, for downloading new versions of an application from a deployment server.
- A new function, `syncmessage`, sends and receives string messages from a Bluetooth server.
- A new function, `barcode.read`, reads a barcode using an Android device's camera.

- A new object, SystemApp, that simplifies interacting with external applications. It has the functions: `clear`, `exec`, `getResult`, and `setArgument`.
- Using `valueset.add` it is possible to add a single value or a range of values from an existing value set.
- New options for the `syncconnect` function to connect to a local Dropbox or FTP folder.
- A range of occurrences can be supplied to `insert` for mass insertions or `delete` for mass deletions.
- The `tolower` and `toupper` functions work properly with accented characters.
- The `view` function, on Windows, now displays webpages in an embedded browser.
- The field name of the last location of a partial save can now be retrieved in any procedure or function.
- The `loadcase` and `delcase` functions can now load or delete cases based on a full key string argument.
- The `countcases` and `keylist` functions can now be called, in certain circumstances, on the input dictionary.
- The `find` and `locate` functions have a new relational operator option, `uuid`, that allows loading cases by a case's UUID.
- The `set access` function, or optional arguments to the dictionary argument of `countcases`, `forcase`, `keylist`, or `selcase`, allows permanent or temporary modification of the dictionary access parameters, which control the order and what kinds of cases are processed.
- The `set first` and `set last` functions now use the dictionary access parameters when determining what is the first or last case.
- The `sysparm` function can be used to determine the presence of command line arguments on Windows.

Messages

- Messages can be defined for multiple languages by including multiple language names in the `Language` directive.
- Translations of some runtime messages added for Vietnamese. You can help translate additional messages or add messages in another language.

Android

- An operator (or respondent) can sign the Android screen using the execsystem function and have the signature saved as an image file.
- A new menu option, Update Installed Applications, to automatically check for and install new versions of an application that has been updated on a deployment server.
- Applications can be added to the device by scanning a QR code.
- A new Settings screen, accessed via the menu on the Entry Applications screen, allows user to clear saved credentials.
- The option for selecting that hidden applications are shown has been moved to the Settings screen.
- The default setting for whether or not the case tree shows on phones and tablets can be modified via logic.

Tools

- The Concatenate Data, Index Data, and Reformat Data tools now allow the saving of PFFs with the specified settings and the loading of the settings of previous runs.
- The Reformat Data tool has been completely rewritten. If using a data source with an embedded dictionary, such as a CSPro DB file, you do not need to specify the input dictionary. The dictionary differences are displayed with more detailed information, including information on destructive changes that would occur during a reformat.
- The Index Data tool has also been completely rewritten. It has a new interface that simplifies running the program on multiple input files; it now works with non-text data files; and there is a more modern interface for viewing cases while selecting which duplicate cases to keep.
- The Sort Data tool allows case sorts on non-text data files that have duplicates.
- When the Sort Data tool processes a sorting key item that does not exist, it treats it as the lowest possible value,

sorting it first when sorting in ascending order. In the past, such values were sorted as 0 (so they would be sorted after negative values when sorting in ascending order).

- The Deploy Application tool supports setting files to only be updated when an application is first installed and not on subsequent updates.
- The Deploy Application tool can generate a QR code that can be used to install applications on mobile devices.

Case Processing

- Items not defined in the dictionary are not written out when a case is saved. In the past, when using absolute positioning, contents of a record that were not defined in the dictionary were still written out when the case was saved.
- Items that are not valid for the item type are not written as they initially appeared in the input data. For example, if the value for a numeric item was "XX," previously, as long as it was not modified, it would be written as "XX." Now, because it cannot be represented as a valid numeric value, it will be written as `default`.
- CSPro processes items using the item type specified in the dictionary. Previously, some tools processed items by using the text representation of the item. For example, the Compare Data tool would indicate that "05" and " 5" were different values because of the differing zero fill setting, but now the tool treats those values as equal.
- When values are saved, special values are converted only based on linkages defined in the item's primary value set. Any special value specifications defined in non-primary or dynamic value sets are ignored.
- Case construction is more forgiving of errors, allowing level 2+ records to appear before their parent level's records. Additionally, level 2+ nodes are joined (by ID) even if some records from a different node separate the nodes. For example, if there is a case with a first level ID of 1 and second level IDs of A and B, in the past, 1/1A/1B/1A would be processed as three second-level nodes, but now they will be handled as two second-level nodes (1/1A+1A/1B).
- The notes and status files, used when processing text data files, are now created only when necessary. (For example, if there are no notes to save, the file is not created.) The information in those files can now be accessed in non-entry applications using functions like `getnote`. Case labels are also now supported for text data files, with the labels saved to the status file.

CSWeb

- A Sync Report summarizes the total number of cases uploaded to CSWeb.
- Create custom roles that specify dashboard and dictionary permissions.
- Downloading data from a CSWeb server is now faster.

See also: What's New in CSPro?

What's New in CSPro 7.5?

This is a detailed list of changes in CSPro 7.5. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

General

- On Android devices it is possible to record and play audio in two modes: interactive mode, with the enumerator able to control the recording; and background mode, with the recording controlled by logic functions using the new `audio` object.
- Listing files can now be written as comma-separated values (CSV) or CSPro data files. Traditional listing files in text format are still supported.

Android

- The *csentry* directory on Android has moved. For new installations the *csentry* directory will be located at:
  <external storage>/Android/data/gov.census.cspro.csentry/files/csentry

If you are upgrading from an earlier version of CSEntry, the *csentry* directory will remain at:
<external storage>/csentry

This change is required to support the more stringent security requirements introduced in the latest versions of Android.

Logic

- A new object, audio, that allows for recording or playing audio. It has the functions: `clear`, `concat`, `length`, `load`, `play`, `record`, `recordInteractive`, `save`, and `stop`.
- A new object, hashmap, that facilitates storage of numbers or strings in an associative array. It has the functions: `clear`, `contains`, `getKeys`, `length`, and `remove`.
- A new function, `syncparadata`, similar in behavior to `syncdata`, allows for the syncing of paradata between devices over Bluetooth, or between a device and a CSWeb, Dropbox, or FTP server.
- A new function, `replace`, returns a string with one of more instances of a substring replaced with new text.
- A new function, `encode`, escapes special characters to facilitate writing to HTML or CSV files, or encodes characters when writing out URIs or URI components.
- A new namespace, path, has functions for interacting with file paths: `path.concat`, `path.getDirectoryName`, `path.getExtension`, `path.getFileName`, and `path.getFileNameWithoutExtension`.
- User-defined functions now support optional parameters as well as passing numeric and string values by reference.
- Conditional compilation of logic is now possible due to the addition of a logic preprocessor.
- New functions for list objects: `list.removeDuplicates` removes duplicate values; `list.removeIn` removes values specified in an in list; and `list.sort` sorts a list.
- The `list.show` and `valueset.show` functions work with string lists and string value sets.
- The `valueset.randomize` function is a new way to randomize value sets (which previously could be done using `randomizevs`). These value set randomizations now obey any seed value provided using the `seed` function.
- The `valueset.sort` function sorts the order of entries in the value set by either label or code.
- The `randomin` function now accepts non-integer values in the input in list.
- New numeric and string variables can be declared inline in the `recode` statement's destination variables section.
- A new callback function, `OnSystemMessage`, provides a way to override the displaying of system error messages.
- It is possible to check a value against all special values using the code `special` in an in list (which are also used by `recode` and `when` statements).
- Lists can also be used as inputs of an in list.
- The `loadsetting` and `savesetting` functions accepts numeric values as part of the attribute-value pair.
- The `hash` function accepts numeric values as inputs.

Tools

- The Deploy Application tool has options to refresh all files or all data entry applications within a folder.

CSWeb

- CSWeb can convert case data to MySQL/MariaDB relational tables using a command line process, allowing for more dynamic reporting.

See also: What's New in CSPro?

What's New in CSPro 7.6?

This is a detailed list of changes in CSPro 7.6. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

General

- The CAPI question text system has been redesigned to use HTML when displaying question text. In addition, question text fills and conditions can now include any logic expression, so using the full set of CSPro logic functions is now possible when working with question text.
- The frequency generation system has been overhauled. Frequencies can now be written to four formats, including Excel and HTML, and they can be generated in data entry applications. The new freq logic object offers a way to tally frequencies in a different way from the traditional `freq` statement.
- Listing files can now be written as Excel or HTML files.

Android

- The new barcode capture type allows for scanning numeric or string barcodes using the device's camera.
- The new slider capture type facilitates capturing numeric range values using a graphical slider.
- When using `execsystem` to launch the camera or signature widgets, it is now possible to specify overlay text that will appear on the device's screen while the widget is active.
- Production PFFs for the Compare Data, Concatenate Data, Index Data, Paradata Concatenator, Reformat Data, and Sort Data tools can now be executed on Android (with a few limitations).
- Applications copied to the *csentry* directory on a mobile device via a local folder deployment (created using the Deploy Application tool) can now be updated by the Update Installed Applications feature.

CAPI Question Text

- Question text now fully supports images, tables, hyperlinks, and HTML code.
- The question text editor supports customizable styles to make it easy to use standard formats throughout a survey. Styles replace the fonts *CAPI Font 1* and *CAPI Font 2* used in earlier versions.
- Fills in question text are now delimited with **~~** instead of **%**. Fills using % in applications created in earlier versions of CSPro are automatically converted to ~~ when opening the application in version 7.6.
- The format of the question text file (.qsf) has changed. The new file format uses HTML instead of Rich Text Format (RTF). Question text files from earlier versions of CSPro will be converted to the new format when opened in version 7.6, though you must edit the QSF text to force the conversion to occur. The original RTF version of the QSF file will be retained as xxx.qsf.backup. Applications with question text created or edited in version 7.6 can no longer be opened in older versions.

Frequencies

- The Tabulate Frequencies tool has new options, including the ability to create percentiles, sort the frequencies, and output frequencies in several different formats. The tool can also directly create a batch application to generate the specified frequencies (using the `freq` statement).
- A new logic object, freq, allows for the creation of frequencies with control over when the values are tallied. It has the functions: `clear`, `save`, `tally`, and `view`.

Logic

- A new feature in the logic editor, Symbol Analysis, shows where each dictionary symbol, form symbol, and user-defined function is used in an application.
- Double-clicking on an error or warning in the compiler output window that originated from an external logic file will result in that file being opened in Notepad++ (if that software is installed).
- The optional `stop` code has new behavior when running batch applications. If not 1, the application's postproc will be executed; if 1, it will not, which was the behavior for all codes prior to CSPro 7.6.

- Saved arrays are now supported in data entry applications.
- The `impute` function has several new behaviors: alphanumeric items can be imputed; the return value of the function, when used on a numeric item, is the value of the imputed value; the `vset` option, that worked using a value set index, has been replaced with a `valueset` option that takes a value set name; and the `stat` data file format has changed, as have the `stat` options, including a way to turn on `stat` automatically for all imputations.
- The `impute` function is fully functional in data entry and tabulation applications (and can be used on Android).
- Encrypted SQLite databases can be queried using `sqlquery` by specifying a password in a connection string or by having an operator enter the password manually.
- The counter variable used in a `do` loop can now be any numeric variable (such as a dictionary item or a list element).
- The `execpff` function, when used on PFFs for several tools with the `wait` flag, now launches the tool without creating a new process.
- The `pff.setproperty` function accepts a dictionary name as an argument, which can be used to run several tools using a dictionary embedded in a .pen file.

Tools

- The Export Data tool can no longer export to DDI 2.0 or CSPro XML formats. In addition, the Export XML Metadata tool has been removed from the CSPro installation. Users still needing this tool can find it on www.csprousers.org.
- The Export Data tool can directly create a batch application with the specified export settings (using the `export` statement).

See also: What's New in CSPro?

What's New in CSPro 7.7?

This is a detailed list of changes in CSPro 7.7. For a more general overview, see the *readme.txt* file located in the CSPro installation folder.

CSPro Designer

- The dictionary editor has an additional view, the property panel. It allows the user to associate additional properties and/or features to a dictionary element.
- External logic files are now editable. Each logic file will appear in the forms or batch edit tree and can be viewed and modified just like any other `PROC`. Double-clicking on an error or warning in the compiler output window will now show the external logic editing window. If you want the old behavior of opening the file in Notepad++, hold down the **Ctrl** button while double-clicking.
- The logic editor supports code folding, a way to collapse and uncollapse sections of code.

Data Entry

- The new toggle button capture type presents field data with two states (selected and not selected).
- Value sets can be shown with colored labels.
- The Review Notes feature, which allows all notes to be reviewed in a single place, is now implemented on Windows.

Android

- The `gps` function accepts new commands that can be used to capture GPS coordinates in interactive modes where details about the current GPS reading are shown or where an enumerator can select a location on a map.

Reporting

- A new logic-based reporting system allows for the creation of dynamic reports, generally created in HTML. The report text, editable in the CSPro Designer, can include logic fills and can use logic to control what parts of the report are generated.

Mapping

- A new object, geometry, provides the ability to display and manipulate vector geometry (points, lines, and polygons). You can interactively trace or walk polygons and can compute a polygon's area and perimeter. Geometry data can be loaded from and saved to GeoJSON files.
- The default base map can be defined in additional ways. If a BaseMap attribute is defined in a PFF, that base map will be used when showing map objects that have not had a base map specified.

CAPI Question Text

- Question text can be added in right-to-left format, facilitating the addition of text in languages such as Arabic or Hebrew.

HTML and JavaScript Integration

- Introduction of HTML dialogs for displaying logic functions (such as `errmsg` and `accept`) as well as other aspects of the CSPro user interface (UI). These are turned on by default but can be disabled. In a future release of CSPro, the old style of dialogs will be removed.
- A JavaScript interface exists to facilitate the invocation of user-defined functions from question text and while viewing web pages (using the `view` or `htmldialog` functions).

Logic

- A new operator, `:=`, allows you to specify arguments by name for some CSPro functions.
- New functions for logic-based reports: `view`, `save`, and `write`.
- A new object, image, allows for the storage and manipulation of images. It has the functions: `captureSignature`, `clear`, `height`, `load`, `resample`, `save`, `takePhoto`, `view`, and `width`.
- A new object, document, facilitates the storage of text or binary files. It has the functions: `clear`, `load`, `save`, and `view`.
- A new object, geometry, allows for the display and manipulation of vector geometry. It has the functions: `area`, `clear`, `getProperty`, `load`, `maxLatitude`, `maxLongitude`, `minLatitude`, `minLongitude`, `perimeter`, `save`, `setProperty`, `tracePolygon`, and `walkPolygon`.
- New map functions for adding and removing geometry from a map: `addGeometry`, `clearGeometry`, and `removeGeometry`.
- Another new map function, `saveSnapshot`, saves the currently displayed map as an image.
- A new function, `path.getRelativePath`, returns a relative path based on a path's location relative to a directory.
- Another new path function, `path.selectFile`, shows the operator a visual listing of the file system and returns the path of the file the operator selected.
- A new function, `inadvance`, indicates if an advance is in progress in a data entry application, allowing for the conditional execution of logic based on whether an enumerator has directly entered a field.
- A new function, `currentkey`, returns a key string comprised of a case's current IDs (as opposed to `key`, which uses a case's ID as they existed when the case was last loaded or written).
- A new function, `synctime`, returns the time of the last synchronization of a data file or a specific case. This function can be used to determine if any cases have not been synced.
- A new function, `setbluetoothname`, allows you to modify the name of the device as broadcast to other Bluetooth devices.
- The `htmldialog` function allows for the creation of custom dialogs using HTML and JavaScript.

- A new function, `invoke`, allows you to execute user-defined functions using runtime binding (where the function name is only known at runtime).
- The Downloads directory path can be retrieved. Copying files to and from this directory is a useful way to share files between applications on Android 11 devices.
- The `dirlist` function, with a new syntax using named arguments, provides additional ways to specify the source directory and wildcard filters. Hidden and system files are no longer included in the directory listing.
- The `getproperty` function can now return some system properties (such as the maximum display dimensions).
- The `setproperty` and `pff.setproperty` functions' property values can be specified as numeric expressions instead of having to be given as strings.
- The `setoutput` function can take a string list as an argument to facilitate specifying multiple output data sources.
- The `view` function has new options that allow users to specify the window size and other attributes of how the embedded web browser will appear when viewing HTML files and websites.
- The `encode` function has two new options (including to encode text to write as a JSON string).
- The `getos` function can fill a hashmap with detailed operating system information.
- Recursive functions can now declare nearly all variable and object types locally, and function pointers can now use nearly all variable and object types as function parameters.

Miscellaneous

- Multiple Output Data entries can be provided for batch applications. Any cases output during the batch run will be written to each of these data sources.
- Some properties are now saved in a file with the extension .csprops. Modifying the properties for paradata must now be done using the new Application Properties dialog.
- The question text file is sorted in form order before saving, potentially making it easier to use when working in a version control environment.

Tools

- Data Viewer allows you to export data directly to CSV, Excel, R, SAS, SPSS, and Stata file types.
- The Export Data and Tabulate Frequencies tools allow you to specify the dictionary language that is used when exporting labels or creating frequency tables.
- The Paradata Viewer's Location Mapper has been redesigned and works using Leaflet mapping. Additionally, when saving reports to Excel format, the Excel column formats will match the formats defined in the *CSPro Reports.csrs* file.

CSWeb

- A Map Report visualizes case data by displaying each case on a map. Further, each case can be opened to view the collected data.
- Ability to upload files to the server via the CSWeb interface.

See also: What's New in CSPro?

CSPro Applications

Data Entry Applications

Description

A Data Entry application contains a set of forms (screens) and logic that a data entry operator uses to key data to a disk file. Data entry applications can be used to add new data and to modify existing data.

You can have the following runtime features in your data entry application:

- Add new cases (questionnaires) or retrieve and modify existing cases
- Execute logic and display messages after any field is entered
- Consistency checks and skip patterns of unlimited complexity
- Multiple lookup files
- Cases indexed to avoid duplication and for easy retrieval
- Collect GPS coordinates
- Use maps
- Collect operator statistics

CSPro is used to develop the data entry application, whereas CSEntry is used to execute the data entry application. For small surveys and for testing applications, you can run CSEntry directly from CSPro, on the same computer. For large surveys and censuses, which require a production environment, you can transfer the application files to other computers and run CSEntry on them.

See also: Create a New Data Entry Application

Batch Edit Applications

A batch edit application contains logic that you can apply against one set of files to produce another set of files and reports. Batch editing applications can be used to gather information about a data file. You can incorporate the following run time features in your batch editing application:

- Write edits (logic) using powerful CSPro language
- Validate individual data items
- Test consistency between items
- Check case/questionnaire structure
- Modify data values
- Use arrays for hot deck or cold deck imputation
- Generate imputation statistics
- Generate edit reports automatically or create a customized report
- Create additional variables
- Read/write to multiple lookup files

You use CSPro to develop the batch edit application. You use CSBatch to run the application. For small surveys and for

testing applications, you can run CSBatch directly from CSPro, on the same computer. For large surveys and censuses, which require a production environment, you can transfer the application files to other computers and run CSBatch on them.

See also: Create a New Batch Edit Application

Tabulation Applications

A **Tabulation application** contains a set of table specifications and a data dictionary describing a data file to be tabulated. When you create your application, you can use an existing data dictionary or you may create one as you create the application.

In a Tabulation application, you can:

• Cross-tabulate a virtually unlimited number of variables.

• Tabulate variables created "on the fly" under program control.

• Select the universe of tabulation.

• Tabulate values and weights.

• Tabulate simple counts and percents .

• Tabulate mean, median, mode, standard deviation, variance, n-tiles, proportions, min, max .

• Perform table cell manipulation after tabulation .

• Define detailed table formatting .

• Save tabulations in several file formats.

• Copy tables to spreadsheets or word-processing documents.

• Produce tables by geographic area.

See also: Create a New Tabulation Application

Data Dictionary

A data dictionary describes the overall organization of a data file; in other words, it gives a description of how data are stored in a file. CSPro requires that a data dictionary be created for each file that is structurally different (i.e., files that would require separate dictionaries to describe their layout). A data dictionary file uses the **.dcf** extension.

In the data dictionary you can:

- Define simple or complex hierarchical file organization.
- Define hierarchical levels, identification items, records, items (fields or variables), value sets (categories of values), and values.
- Create descriptive notes for documentation.
- Define multiply-occurring items.
- Produce reports of file organization.

See also: Creating a Dictionary for a New File, Creating a Dictionary for an Existing File

Forms Design

You can create an unlimited number of forms (screens) for data entry. These can be designed independently or as part of the data entry application.

- Forms may be any size. CSEntry will scroll as necessary.
- Forms may contain fields from different physical records.
- Physical records may be split among different forms.
- Forms may contain individual fields or rosters.

There is usually one forms file (.fmf) per application, but there may be multiple forms files. Each forms file contains one data dictionary file (.dcf) that represents the primary data file that is being created or modified.

See also: Introduction to Forms Design

Tool List

To run a tool, open the **Tools** menu and select one of the tools listed below. There is a user's guide for most of the tools.

Data Viewer

View the contents of a CSPro data file, displayed in tables based on the dictionary contents. The tool allows you to view, but not modify, all of the cases in the file. You can filter the cases based on a search key. In addition to showing the data described by the dictionary, the tool also display the notes and other metadata saved for each case.

Text Viewer

View the contents of any text file up to a maximum of 32,000 characters wide and up to two gigabytes in size. You can copy, save, or print all or part of the contents of the text file. You can also find text in the file, identify line and character position in the file, and copy tabular reports to spreadsheet programs. The file cannot be modified within the Text Viewer utility.

Table Viewer

Examine, but not change, the contents of any CSPro tables file. A table file is produced by running tabulation applications or using the Tabulate Frequencies tool. You can copy, save, or print all or parts of the tables in RTF (for word processing programs), or HTML (for Internet), or TAB delimited (for spreadsheet) formats.

Table Retrieval

Retrieve and display tables, maps, and other previously prepared documents from a large database of documents based on geography, subject matter, and title. It is very useful as a data dissemination tool.

Deploy Application

Deploy one or multiple applications to a server so that they can be downloaded onto interviewer devices.

Pack Application

Pack all the files in a CSPro application into a ZIP file so the application can be moved to another computer or sent as an email attachment.

Tabulate Frequencies

Produce frequency distributions of all or some of the variables in a data file. You simply select the variables (value sets) you want to tabulate and provide the name of the data file. More than one data file can be tabulated.

Sort Data

Sort a data file by questionnaires or by records. The data may be sorted based on ID items, record types, or singly-occuring items.

Export Data

Export selected data records or parts of data records to tab- or comma- delimited files. These files can be imported into spreadsheets or databases. The tool also allows you to export data to SAS, SPSS, Stata, or R formats.

Reformat Data

Reformat data from one file format to another using an input and output data dictionary. Fields with corresponding names are copied from the input to output file. This is useful for reorganizing data records or lengthening data items.

Compare Data

Compare the contents of two data files and identify the differences. The data files must have the same structure, that is, the same CSPro dictionary must describe both data files.

Concatenate Data

Concatenate (join end-to-end) two or more CSPro or text files. You do not need a dictionary for this tool when working with text-based files.

Table Retrieval Setup

Create and modify a set of tables and other documents organized by geographic area, subject and title for use by the Table Retrieval tool.

Paradata Viewer

Display reports about the paradata collected during an application's run.

Paradata Concatenator

Combine multiple paradata logs into a single log.

Index Data

Generate indices for data files or identify duplicate cases in a data file.

Excel to CSPro

Convert data from Excel workbooks to CSPro data files. The tool can also create a CSPro dictionary for the data in an Excel worksheet, performing an analysis to determine the best CSPro format for the data.

PFF Editor

Edit and view all options available when running a CSPro application file via a PFF file.

Production Runner

Set up a series of CSPro processing tasks and then run them all at once.

Operator Statistics Viewer

View one or more operator statistics files. These files are created when keying in data and can be used to track the progress of a keying operation. Reports can be viewed for the whole operation or on a per-keyer or per-day basis.

Save Array Viewer

View and modify saved arrays. If batch editing a census or survey, this tool can be used to view and modify the hotdecks used in the edits.

Deprecated Tools

These tools are no longer distributed with CSPro but are instead available on www.csprousers.org.

Convert Dictionary

Convert IMPS and ISSA data dictionaries to CSPro dictionaries, or convert CSPro dictionaries to IMPS or ISSA dictionaries. You can also convert ISSA dictionaries to CSPro dictionary and data entry forms.

Form Viewer

View snapshots of the data collected during a data entry application.

Map Viewer

View, create, and manipulate thematic maps of data.

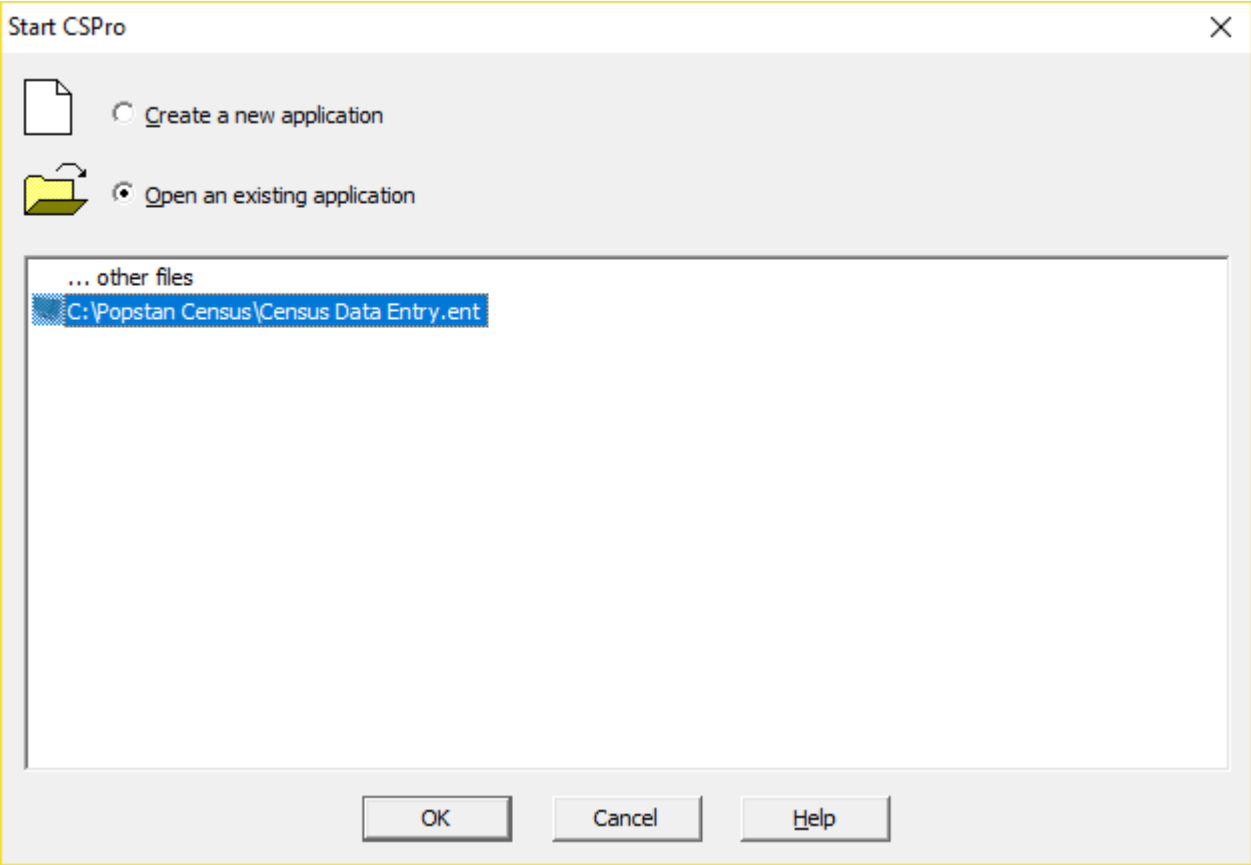
Convert Shape to Map

Convert ESRI ArcView or ArcInfo polygon shape files to a format for Map Viewer.

CSPro General Concepts

CSPro Initial Screen Layout

To open CSPro, click on the CSPro icon on your desktop. The screen will be subdivided into two parts: the left is reserved to display file trees; and the right window is reserved to display the actual application. Initially both windows are empty.



Create a new application: This allows you to create a new application when CSPro is launched. After you specify the names of the applications files, the new application is opened.

Open an existing application: This allows you to open an existing application either by selecting a recently used application from the list provided or to select, using other files, any CSPro application available on the computer or connected servers.

If you cancel the dialog box, CSPro will remain open so that you can use CSPro tools or at a later time open an application or create a new application.

Trees in CSPro

CSPro uses trees to present what files are currently loaded (such as dictionaries), and more importantly, what files are associated with the active application (dictionaries, forms, batch edits, or tables).

After a dictionary or application is opened or created, the "Files" tree will show the application, along with all files that belong to it. The dictionary tree ("Dicts") will also be present. Then, depending on the type of application opened or created, other trees will also appear.

There are five types of trees in CSPro:

- **Files** tree—shows all open applications; expand the tree for a given application and you will see all files that belong to it.
- Dictionaries tree ("**Dicts**")—shows all open dictionaries; expand the tree for a given dictionary and you will see its structure (level, records, items, etc).
- **Edits** tree—shows all open batch edit specifications; the order of the blocks and the fields within each block indicates the order the edits will be processed.
- **Forms** tree—shows all open form files; expand the tree for a given form file (indicated by the triple yellow boxes), and you will see all form(s) that belong to it, as well as the fields contained on each form.
- **Tables** tree—shows all open table specifications and their contents.

The **Files** and **Dicts** trees are always present. The other three trees are only present if the corresponding application is open.

The tabs at the bottom control which tree is displayed. Simply click on the desired tab to switch the tree view. Use **Ctrl+T** to toggle the tree view between full file names & labels (File tab) or unique names & labels (all other tabs). To switch the frame on the right side of the screen to the corresponding view, double-click anywhere on the desired tree.

See also: Data Dictionary Tree, Forms Tree, Batch Edit Tree

Windows

The window on the right side of the screen allows you modify the contents of a dictionary or application. Each different window has different functions associated with it. That is, you will see a different menu and toolbar with each different window.

Part of the toolbar to the left of the help button shown below allows you to switch between different types of windows: Dictionary, Forms Design, Batch Editing, and Tabulation.

To change the contents on the right side of the screen press the button of the type of window you want to view. If there is more than one window of that type, the most recent one viewed will be displayed.

If you need to select a particular window, from the **Window** menu, select the file name you want to view.

CSPro Workspaces

When using the CSPro Designer, the working space is generally divided into three areas:

- **Left**: a tree showing dictionary elements, form elements, etc.
- **Middle**: working space
- **Right**: a property panel (e.g., the dictionary property panel)

The term **workspace** refers to this middle working space. This workspace displays different content based on what you are editing:

- **Dictionary Editor**: High-level properties for the selected record, item, etc.
- **Form Designer**: Form content
- **Table Designer**: Table content

- **Logic Editor**: Logic and messages

CSPro General Functionality

CSEntry Data Source File Types

Overview

With the release of 7.7, CSPro now offers more data source types than ever to read from and write to; more exciting, with the 7.7 release, all non-CSPro/text file types will be written to natively. No more two-step processing to produce your R, SAS, SPSS, or Stata files. In addition, some new file types have been introduced, such as Excel.

The following **File Associations Matrix** provides a summary of the possible file formats found in the CSEntry Run Dialog:

CSEntry Allowable File Types	File Extension	External Data (dcf)	Input File	Impute Stat File	Impute Freq File	Freq File	Saved Array	Write File
Standard File Types								
None		Y	Y	Y				
CSPro DB	.csdb(e)	Y	Y	Y				
Comma-delimited	.csv	Y	Y	Y				
Tab-Delimited	.tsv	Y	Y	Y				
Excel	.xlsx	Y		Y	Y	Y		
HTML	.html				Y	Y		
R	.Rdata	Y		Y				
SAS	.xpt & .xpt.sas pair	Y		Y				
SPSS	.sav	Y		Y				
Stata	.dta	Y		Y				
Table (CSPro)	.tbw				Y	Y		
Text	.dat, .lst, etc	Y		Y	Y	Y		Y
Special File Types								
Multiple Files			Y					
Saved Arrays (.sva)	.sva						Y	
Semicolon-delimited	(any)			Y				

If you prefer to jump to a specific Data Source Type, click on one of the links below; otherwise, pick up reading just below this list:

None	SPSS
CSPro DB	Stata
CSPro DB Encrypted	Table (CSPro)
Comma/Tab-Delimited	Text
Excel	CSPro (Exported)
HTML	Multiple Files
R	Saved Arrays
SAS	Semicolon-Delimited

Standard File Types

Standard file types are those found within the Source Type dropdown of the **CSEntry** run dialog. A description of the file

types supported by CSPro, with the default (or required) file extension that can (or must) be used shown in parentheses. Example connection strings are given for several Source Types below; see the Export Attributes page for a full explanation of the available options and required syntax.

None

This is the one exception to the other entries in this matrix, for in fact, no file will be generated. This option is available for Input, Impute Stat Files, and External (i.e., lookup) files, as well as the file generated from any `impute` function calls that use the `stat` option. "None" appears in the Source Type dropdown if it is available for the given file. It can also be manually entered in the **.pff** file (if allowed). The following is how your .pff file would appear if no Input Data file was selected:

```
InputData=|type=None
```

CSPro DB (.csdb)

.csdb is the default file type for Input, Output, and External (i.e., lookup) files, as well as the file generated from any `impute` function calls that use the `stat` option. Listing files can also be written as .csdb files, though the option must be chosen from the dialog's Source Type dropdown

Through the use of connection strings, you can choose what you want to write to your output files. For example, suppose you just want one record written out, not all records; this is possible with the following parameters:

```
population_rec.csdb|record=POP_REC&type=CSProExport
```

When this happens a CSPro dictionary will also be generated; it will have the same filename, with **.dcf** appended to it. Using the above example, you would see **population_rec.csdb.dcf** had been created. If you wish to control the dictionary's name, you can do so via the "dictionary" Export Attributes as follows:

```
population_rec.csdb|type=CSProExport&record=POP_REC&dictionary=pop_rec.dcf
```

Note how the "type" and "record" arguments were switched above. After the pipe symbol has been given, connection strings can be given in any order; they just need to be combined with the ampersand symbol (&).

CSPro DB Encrypted (.csdbe)

For any source file entry that allows **.csdb** files, CSPro's encrypted file type (**.csdbe**) can also be used. For Input and existing Lookup files, this means the file has already been given a password, which must match the file's password or the run will fail. You will be prompted for each file's password after pressing "OK" on the **CSBatch** file association dialog. Alternatively, you can provide the password along with the filename by using a connection string. For example, `"MyOutputFile.csdbe|password=s@feW0rd"`. Be sure to place the pipe symbol immediately after the filename, followed by "password=" and your desired password. See the Export Attributes page for a full explanation of the available options and required syntax.

Comma-Delimited (.csv)

This option is available for Output, External (i.e., lookup), and Listing files. It can also be used for the file generated from any `impute` function calls that use the `stat` option. It is a text-based file, readable by Excel or any ASCII editor (such as Notepad or Wordpad). Only non-repeating records will be output to the file; repeating records will be ignored.

Tab-delimited (.tsv)

This option is available for Output and External (i.e., lookup) files. It can also be used for the file generated from any `impute` function calls that use the `stat` option. However, only non-repeating records will be written to the **.tsv** file; repeating records will be ignored.

Excel (.xlsx)

This option is available for all file associations except the Input, Write, and (Saved) Array Files. This will create a native

Excel file, which handles commas and single and double quotes correctly. When creating an Excel file, a separate tabsheet will be produced for each record type found.

Additionally, with the usage of connection strings, you can further control what is written to your Excel file. For example, if you only want the population record, and would also like to see columns for both the values (e.g., 1, 2) and corresponding labels (e.g., male, female) for each dictionary item found, the following Export Attributes should be appended to the file name:

```
.\POP_REC.xlsx|record=POP_REC&header=labels&values=codes-and-labels
```

HTML (.html)

This option is available for Listing, Freq, and Impute Freq Files. It is a text-based file, viewable by any web-browser (such as Firefox or Opera) or any ASCII editor (such as Notepad or Wordpad). Only non-repeating records will be output to the file; repeating records will be ignored.

R (.Rdata)

This option is available for External (i.e., lookup) files. It can also be used for the file generated by `impute` function calls that use the `stat` option. Both single and multiply-occurring records will be written to the file.

SAS (.xpt & xpt.sas pair)

This option is available for External (i.e., lookup) files. It can also be used for the file generated by `impute` function calls that use the `stat` option. Both single and multiply-occurring records will be written to the file.

SPSS (.sav)

This option is available for External (i.e., lookup) files. It can also be used for the file generated by `impute` function calls that use the `stat` option. However, only non-repeating records will be written to the **.sav** file; repeating records will be ignored.

Stata (.dta)

This option is available for External (i.e., lookup) files. It can also be used for the file generated from any `impute` function calls that use the `stat` option. However, only non-repeating records will be written to the **.dta** file; repeating records will be ignored.

CSPro Table (.tbw)

The CSPro table file type is available for Freq and Impute Freq Files. CSPro's **.tbw** files are viewable in any ASCII editor, tho they are meant to be viewed by CSPro's Table Viewer. There, you will see the familiar tree structure on the left (containing a list of all tables in the file) with the right side of the window devoted to the table display, presented in an Excel-like format.

Text (.lst, .dat, user's choice)

Text is the default file association for Listing, Freq, and Impute Freq Files, tho it can be used for all other file designations, except the (Saved) Array file (which must use the **".sva"** extension). The generated file is ANSI-based, viewable by any web-browser (such as Firefox or Opera) or any ASCII editor (such as Notepad or Wordpad).

Depending on the file being specified, a default file extension may or may not be given. For example, Listing and Impute Freq Files will be given an **".lst"** extension by default, although no default file extension is given to Freq Files. If a non-text file designation was used for Input or Lookup Files and the file type is changed to Text, a default file type of **".dat"** will be given. (For example, MyCensus.csdb will change to MyCensus.dat if the Source Type is changed from .csdb to Text.) No default extension is given for Write Files. Whatever extension is used, you can change it to whatever you prefer (for example, **".txt"**).

Special File Types

The following Source Types are not selectable in CSEntry's Run dialog; instead, they will only appear based on logic found within your application, or Export Attributes provided for the file.

Multiple Files

This Source Type can only occur for Input Data Files. For it to appear, 2+ data files must be referenced, at which point CSPro will automatically change the source type to "Multiple Files". You will not find this source type in the file association dialog dropdown, it is not a selectable type.

For example, if these files were listed in the File Associations entry for the Input Data File:

```
".\C001.dat" ".\C002.dat" ".\C003.dat"
```

you would see the "Source Type" change to "Multiple Files". However, if you were to add this manually in the **.pff** file, it should appear on separate lines as follows:

```
InputData=.\C001.dat
InputData=.\C002.dat
InputData=.\C003.dat
```

As mentioned above, whenever the **type=CSProExport** option is used, a dictionary will be generated for each **.csdb**, **.dat**, or other CSPro file type being generated.

Saved Array (.sva)

Saving out arrays between runs has many uses, especially when working with hot decks. If an `array` declaration appears anywhere in your program with the `save` keyword, **CSEntry** will add a file entry to the file associations dialog.

Semicolon-delimited (user's choice)

This option is available for the file generated from any `impute` function calls that use the `stat` option. However, only non-repeating records will be written to the file; repeating records will be ignored. In order for CSPro to know you want a semicolon-delimited file, you must use the "type" connection string as follows:

```
myFile.out|type=Semicolon
```

When this parameter is added, CSPro will change the Source Type to "Semicolon-delimited". Note when this option is used for Impute Stat files, a dictionary will be created with the same name.

Connection String

A connection string is a text string that details how to open a data source. Generally a connection string is just a file name. Connection strings are used in PFF files as well as in logic when data sources are specified, such as when using `setfile`. A connection string is made up of several components:

```
file_name | property_name1=property_value1 & property_name2=property_value2
```

The ***file_name*** is the file name of the data source. It can be specified as a relative or absolute path. If only the file name is specified, then the connection string is complete. However, if any properties are specified, then the properties must be separated from the file name by a pipe | character.

Multiple properties can be specified, with each grouping separated by the ampersand & character. The property name is listed, followed by an equals = sign, and then the property value is given. Most property values can be defined in human-readable text, but if the value contains special characters such as & or = characters, it must be percent-encoded.

The property "type" is used by CSPro to specify the type of a data file. Generally this is not necessary to define because

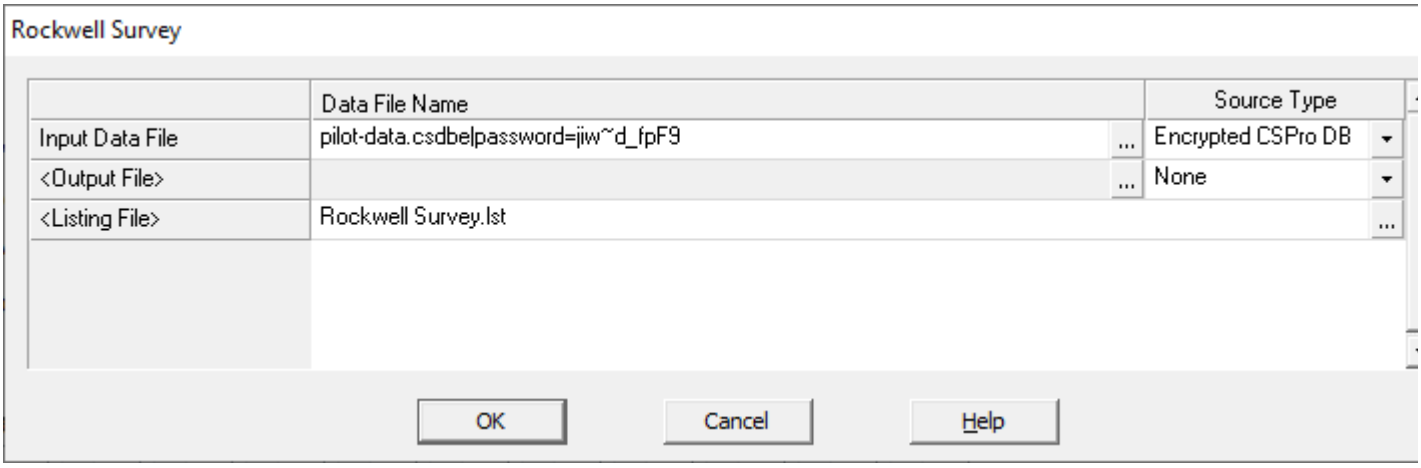
the data file type can be deduced from the file extension, but in one case it is necessary. This, for example, is how the none data source is specified in a PFF:

```
InputData=|type=None
```

Other properties are documented in the topics describing the data sources. An example of one property is the "password" property for encrypted data files. It can be specified in logic:

```
string pin_number = prompt("Enter your PIN number:", password, numeric);

setfile(HOUSEHOLD_DICT, "Household.csdbe|password=" + pin_number);
```

Or specified for the PFF:



Properties List

Properties include:

Property	Description
password	The password that opens an encrypted data file.
cache	Indicates to CSPro that cases should be cached in memory. This may be useful to advanced users who want to optimize programs that do a lot of case lookups.

See also: CSEntry Data Source File Types

Encrypted Data

Overview

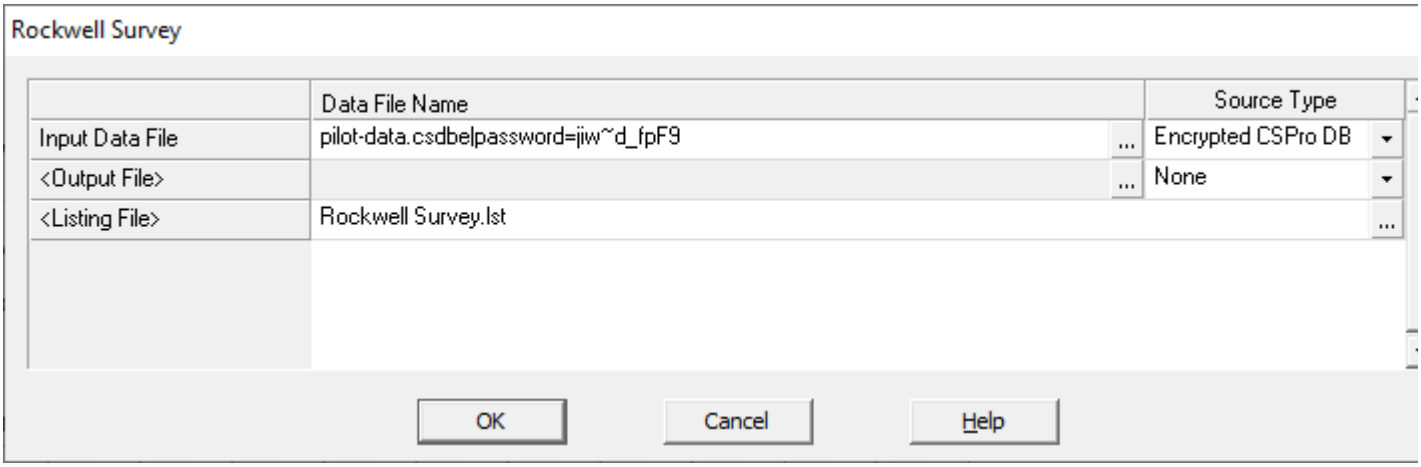
Using the Encrypted CSPro DB data source, you can read from and write to a data file that is protected by a password. The file cannot be opened without the password, meaning that it is important to implement a sufficient password management policy to ensure that you do not lose data during the data collection process.

An Encrypted CSPro DB file is like a CSPro DB file and can be used in any CSPro application. The only difference is that, upon opening the file, CSPro requires the specification of a password. There are two ways to specify a password:

Password entry: A dialog box will appear allowing the user to enter the password. The password must be at least four characters. If the data file does not exist and is being created for the first time, the user must enter the password twice to ensure that the password is entered correctly. You can reduce the number of times that a user must enter the password by allowing the password to be cached for a specified duration on the machine. When opening an existing file, if the password is not correct, the user will be prompted to enter the password again.

Connection string: The password can be specified in a connection string used in a PFF file or by `setfile`. If the

password is specified in the connection string, then the user will not be prompted to enter a password. If the data file does not exist and is being created for the first time, the specified password will be used to encrypt the file. When opening an existing file, if the password is not correct, the opening of the file will fail. For example, the following connection string opens the file *pilot-data.csdbe* with the password *jiw~d_fpF9*.



While the data in the file is encrypted, it is also a good idea:

- To encrypt the entire drive where the CSPro application and data files reside, using encryption such as Windows BitLocker or Android's full-disk encryption.
- Not to store passwords in plaintext anywhere (such as in a CSPro logic file or in a PFF file, as is done in the above image).
- To synchronize your data using a secure protocol (such as CSWeb over https). If you encrypt your data file on a tablet but then transfer the data over http using `syncdata`, that defeats much of the purpose of encrypting.

Technical Details

Encrypted CSPro DB files are SQLite files encrypted using the SQLite Encryption Extension (SEE) using "AES-256 in OFB mode." The specified password is not used as the key input to SEE but is instead hashed to create a 256-byte key that is used to encrypt the file. If allowed, this hash, not the password, is cached on the machine. A fixed salt is used during the hashing process because there is no suitable place to store a dynamic salt. This means that the same password will always result in the same encryption key.

See also: Dictionary Security Options, CSEntry Data Source File Types, Connection String

Unicode Primer

Beginning with version 5.0, CSPro is Unicode compliant.

What is Unicode?

Unicode is a widely adopted system for representing characters for all languages currently in use. Early computer programs generally used only one byte to represent a character, which led to a limit in the number of characters that could be displayed on screen and used in computations. This limit of 256 characters was used effectively by people who only required English characters, or characters from most European languages, but it could not represent the languages used by more than a billion people, including most notably, many Asian languages.

The Unicode standard now defines more than 100,000 characters, but many of these characters represent extinct languages. For that reason, Windows only provides native support for a subset of Unicode characters. In a Unicode program, a character is represented by two bytes, which allows for up to 65,536 characters.

Whereas in English it is very clear what makes up one character—one keystroke—in other languages it is not as straightforward. For example, in Chinese, typing two characters, "天津," requires seven keystrokes ("Tianjin"). In Bangla, the character ■■ requires two keystrokes (■ + ■■), which combine to create one character. Both the Chinese and

Bangla examples require four bytes in memory and return a string length of 2.

What is UTF-8?

With each character stored as two bytes in memory, there are several ways to write characters to a disk. One way is to write two bytes for every character, but this is costly for most users, the bulk of whom only use characters that can be expressed properly with either the ASCII or ANSI encoding systems. The computer world has settled on using UTF-8, a variable-length encoding scheme that uses between one and four bytes to represent a character. ASCII characters are represented using one byte, other ANSI characters are represented using two bytes, and most Asian characters are represented using three bytes. To identify a file as encoded in UTF-8, a three-byte BOM (byte order mark) is placed at the beginning of a text file. For example, here is the UTF-8 representation of: "I am François from 法国."

Character	(BOM)			I		a	m		F	r	a	n	ç		o	i	s		f	r	o
Hex	EF	BB	BF	49	20	61	6D	20	46	72	61	6E	C3	A7	6F	69	20	20	66	72	6F
ANSI	ï	»	¿	I		a	m		F	r	a	n	Ã	§	o	i	s		f	r	o

With a UTF-8 encoding, a file's size is not equal to the number of characters in the file. An empty file is three bytes because of the BOM. (The `fileempty` function can be used to determine whether a file is actually empty.) An ASCII file converted to UTF-8 will generally be the size of the original file plus three bytes. An ANSI file that uses non-English characters will be even larger. The name François takes eight bytes to represent in ANSI but uses nine bytes in UTF-8.

How Will CSPro Work with Unicode Files?

Versions of CSPro up to 4.1 used ANSI characters for data files and all specification files. Starting with version 5, CSPro uses the UTF-8 encoding scheme for representing Unicode characters. CSPro will always be able to read both ANSI and UTF-8 files, so any old files will work in the new version, but when files are modified and saved the files will be rewritten in UTF-8. All new data files and listing reports will be created in UTF-8. Because earlier versions of CSPro only supported ANSI files, any file created in CSPro 5.0 or later, including data files, is no longer automatically backwards compatible with older versions of CSPro.

How Will UTF-8 Affect My File Sizes?

If you only use ASCII characters in your specification and data files, your new files will be only three bytes larger than the CSPro 4.1 equivalents. These three bytes reflect the size of the UTF-8 BOM. If you use many accented characters that are valid in ANSI (like à and û), then your files will increase in size, though only slightly, because the vast majority of characters in any CSPro file are still digits and Latin characters. In general your files will increase in size by an inconsequential amount.

What if I Want to Use a Data File With an Older Version of CSPro?

The CSPro installation includes a tool, the Unicode Text Converter, that allows you to change the encoding of text files. This utility allows you to convert ANSI files to UTF-8, though this is not necessary because CSPro 5.0 or later does this automatically. You will more likely use the tool to convert UTF-8 files to ANSI. If you use non-ANSI characters in your specification or data files, these characters will be converted to question marks during the conversion to ANSI. For instance, "I am François from 法国." will become "I am François from ??." Converting files from UTF-8 to ANSI is not necessary a lossless conversion, so it should be done only if absolutely necessary of if you are confident that you did not use any Unicode-only characters.

Some text editors, including Notepad, allow a user to change the encoding of a text file. These editors can be used to convert text files from UTF-8 to ANSI. If using Notepad, open the file and then select Save As. At the bottom of the dialog box is an option to change the encoding.

What about CSPro's Binary Files?

The important binary files created by CSPro are compiled data entry applications (.pen), file indices (.idx), and intermediary tabulation files (.tab and .tai). These formats have changed and CSPro can no longer read binary files written by CSPro 4.1 or previous versions. CSPro can determine if a compiled data entry application is of the correct format and will present an error message to an operator who tries to open an old version in CSEntry. CSPro can also determine if an old index is being used and will automatically generate a new index. If you use intermediary tabulation

binary files (e.g., if you run tables in parts), you should delete and regenerate them when upgrading to a Unicode version of CSPro. The Unicode Text Converter will not work correctly on binary files.

Can I Still Write and Save to ANSI Format?

Outside of the Export Data tool, CSPro no longer supports writing to ANSI format, though it can still read files encoded in ANSI. CSPro automatically converts files from ANSI to UTF-8 formats as needed. CSPro will convert only files that may be modified by a program. For example, if you use an external file in your application but only call fileread and never filewrite, CSPro will keep the input file in its original format. However, if there is a filewrite statement and the file is encoded in ANSI, CSPro will convert the file to UTF-8 before opening it. The old ANSI file will be sent to the Recycle Bin in case you want to recover it.

Distributing CSPro Data Files

Most applications support Unicode so you may not have a problem using external applications with your data or specification files. However, if you must use an application that is not Unicode compliant, run the Unicode Text Converter on the necessary files.

If you are distributing files to a broad audience and if your data file does not contain any non-ANSI characters, you might consider converting it to ANSI for maximum compatibility. To illustrate an example of what might happen with a non-Unicode program reading a UTF-8 file, imagine a CSPro data file that has two record types, P (population) and H (housing). The ANSI file might look like this:

```
P<case id><data>
P<case id><data>
P<case id><data>
H<case id><data>
```

The UTF-8 file might look like this:

```
<BOM>P<case id><data>
P<case id><data>
P<case id><data>
H<case id><data>
```

An older version of CSPro would regard the first row of data as an invalid record because the BOM was in the place of the record type. It would thus register the household as having two members instead of three. If any non-ASCII characters existed in the data file, it would further shift the data and then items would not be read correctly.

If you release a UTF-8-encoded data file widely, you may want to include a documentation file with text similar to the following:

> **Warning: This CSPro data file is encoded in UTF-8. UTF-8 is a text encoding format widely used to represent characters from any language. If you use this data file with a software package that does not support Unicode and UTF-8, you must first convert the data file to an ANSI encoding. Without this conversion there is a risk that the software package will misread parts of the data file.**

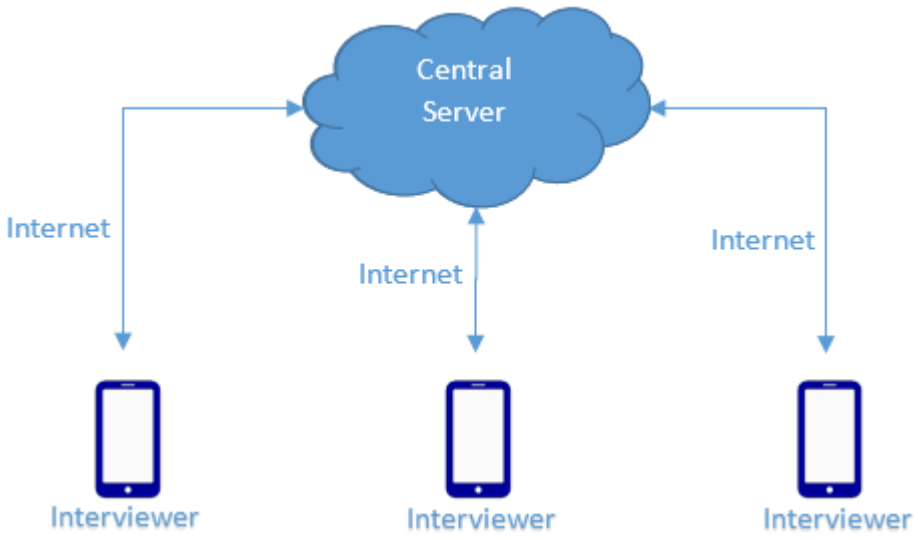
Synchronization Overview

Synchronization Strategies

When using mobile devices for a survey or census it is important to be able to transfer data collected in the field back to the head office. This allows for faster processing and analysis as well as better monitoring of the progress of the field operation.

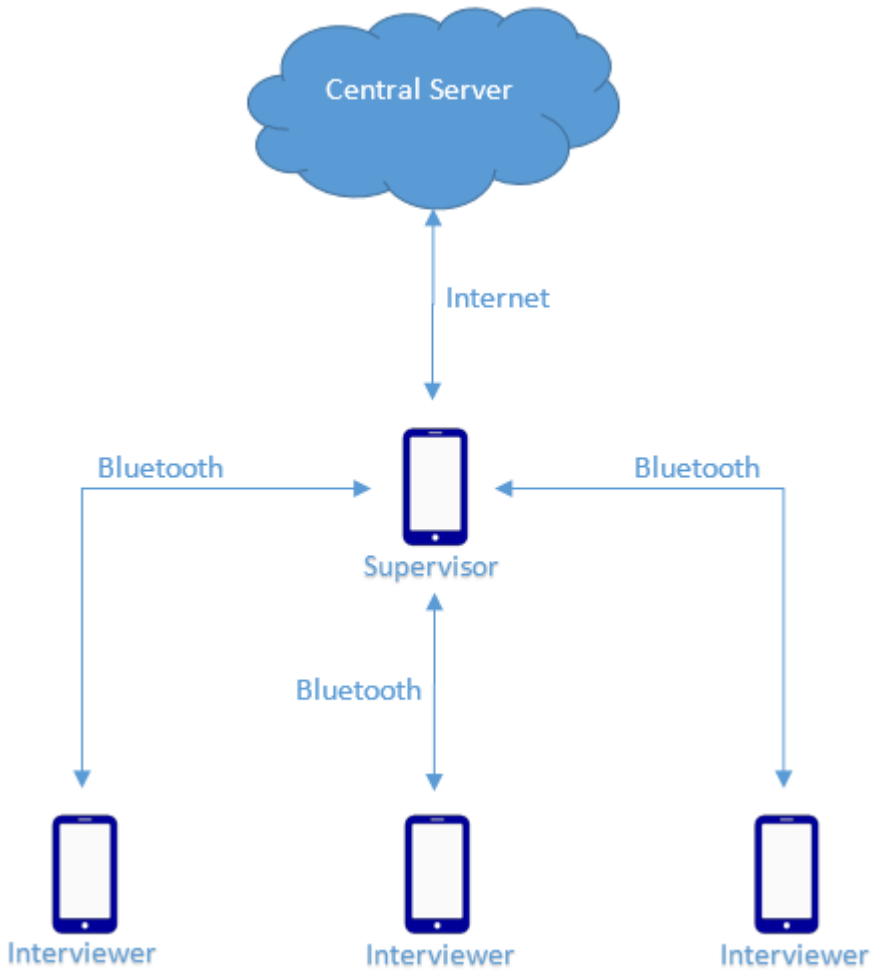
In CSPro, transferring data between devices in the field and the head office is referred to as synchronization. CSPro supports data synchronization over the Internet between interviewers' devices in the field and a central server. Interviewers use CSEntry to collect data on tablets, phones, or laptops and then use the Internet to synchronize the data on their devices with a server at the head office or in the cloud. In CSPro, the data collection itself may be done entirely offline,

with no Internet connection. When an interviewer is able to connect to the Internet they may synchronize with the server and transfer any data collected since the last synchronization. Synchronization may be done using Wi-Fi or a mobile data connection (2G/3G/4G).



Direct synchronization between interviewers and central server over the Internet

In situations where an interviewer rarely or never has access to the Internet, CSPro supports peer-to-peer synchronization between devices using Bluetooth. Bluetooth synchronization does not require an Internet connection. It is a direct connection between two devices that are in close proximity. Using Bluetooth, an interviewer may synchronize their device with a supervisor's device, transferring their data to the supervisor. Later, the supervisor travels to a location where they are able to connect their device to the Internet in order to synchronize with the central server. In this scenario, a supervisor might visit multiple interviewers in order to synchronize with them and later upload their data to the server at the head office.



Synchronization with supervisor over Bluetooth when interviewers do not have Internet access

CSPro supports both one-way and two-way data synchronization. This means that is possible to send data from the interviewer's device to the server as well as to download data from the server to the interviewer's device. You can configure your synchronization to only send data to the server, to only receive data from the server, or to both send and receive data.

It is possible to synchronize both CSPro data files and other non-data files such as application files, images, and text files. It is possible to update data entry applications in the field by downloading the latest .pen and .pff files from the server. This way, modifications to the application at the head office can easily be distributed to interviewers in the field. It is, however, simpler to deploy and update application files on mobile devices using the Deploy Application tool. This tool packages application files for deployment, uploads them to a server and allows interviewers in the field to install and update them on a tablet by choosing Add Application from the menu on their mobile device.

Synchronization Server

For synchronization over the Internet, a central server is required. CSPro supports three types of servers: CSWeb, Dropbox, and FTP.

- **CSWeb**: CSWeb is a web server running the CSPro synchronization server software. It is best for very large surveys and censuses. The server software is written in PHP and can be run on any web server that has PHP and the MySQL database software installed. The server can be set up on a computer at the head office that is connected to the Internet or it may be set up on a hosted website or virtual server in the cloud. Setting and maintaining a server for CSWeb requires knowledge and experience with web server maintainence and cyber security. Users that do not have these skills should consider using Dropbox or FTP instead. For more information about CSWeb servers, see the CSWeb help documentation.
- **Dropbox**: Dropbox is a free cloud based synchronization service. It is ideal for those doing small and medium survey operations. Dropbox requires no server setup or maintenance and avoids the cost and difficulty of setting up a CSWeb server.
- **FTP**: CSPro synchronization can be used with a FTP (file transfer protocol) server. For those doing small and medium survey operations who do not want to use Dropbox and are not able to configure a server to use CSWeb, FTP can be used for synchronization.

When synchronizing CSPro data files, synchronization is done at the case (questionnaire) level. CSPro keeps track of which cases have been added or updated and only sends cases that are new or have been modified since the last synchronization. This significantly reduces the amount of data transferred and therefore reduces bandwidth and the cost of air time. It also reduces the chance that two interviewers overwrite each other's work by both syncing to the same data file on the server. As long as two interviewers do not modify the same case at the same time, they may both synchronize to the same server without overwriting each other's data.

Once interviewers in the field have used data synchronization to upload data to the server, you can use the Data Viewer tool to download the data. When using CSWeb for data file synchronization, data is stored in a MySQL database on the server. When using either Dropbox or FTP, data will be stored on the server in a set of files in the directory *CSPro/DataSync*. In both cases, the data on the server cannot be opened directly by CSPro. In order to retrieve the data that has been uploaded to the server, use the Data Viewer's download function. This will download all the data on the server into a single CSPro data file that can be used by other CSPro applications and tools.

Simple Synchronizations

Basic synchronizations can be set up by specifying a few options documented here.

Advanced Synchronizations

For more advanced synchronization scenarios, CSPro has logic functions that can be used to implement data and file synchronizations. Using logic, it is possible to synchronize data files, including those associated with external dictionaries, and non-data files over either the Internet or Bluetooth.

Synchronization from Logic

There are several logic functions used for synchronization with a server:

- syncconnect is used to connect to the server and initiate the synchronization session.
- syncdisconnect is used after synchronizing to end the session.
- syncdata is used to synchronize cases in CSPro data files with the server.
- syncfile is used to upload/download non-data files to/from the server.
- syncparadata is used to synchronize paradata to/from the server.
- syncapp is used to update an application on the device from a new version on the server.
- syncmessage is used to send to and receive string messages from a Bluetooth server.
- synctime returns the time of the last synchronization of a data file or a specific case.

While it is possible to use syncfile to synchronize CSPro data files, it is more efficient to use syncdata since it will only send/receive cases that have been modified whereas syncfile always sends the entire file.

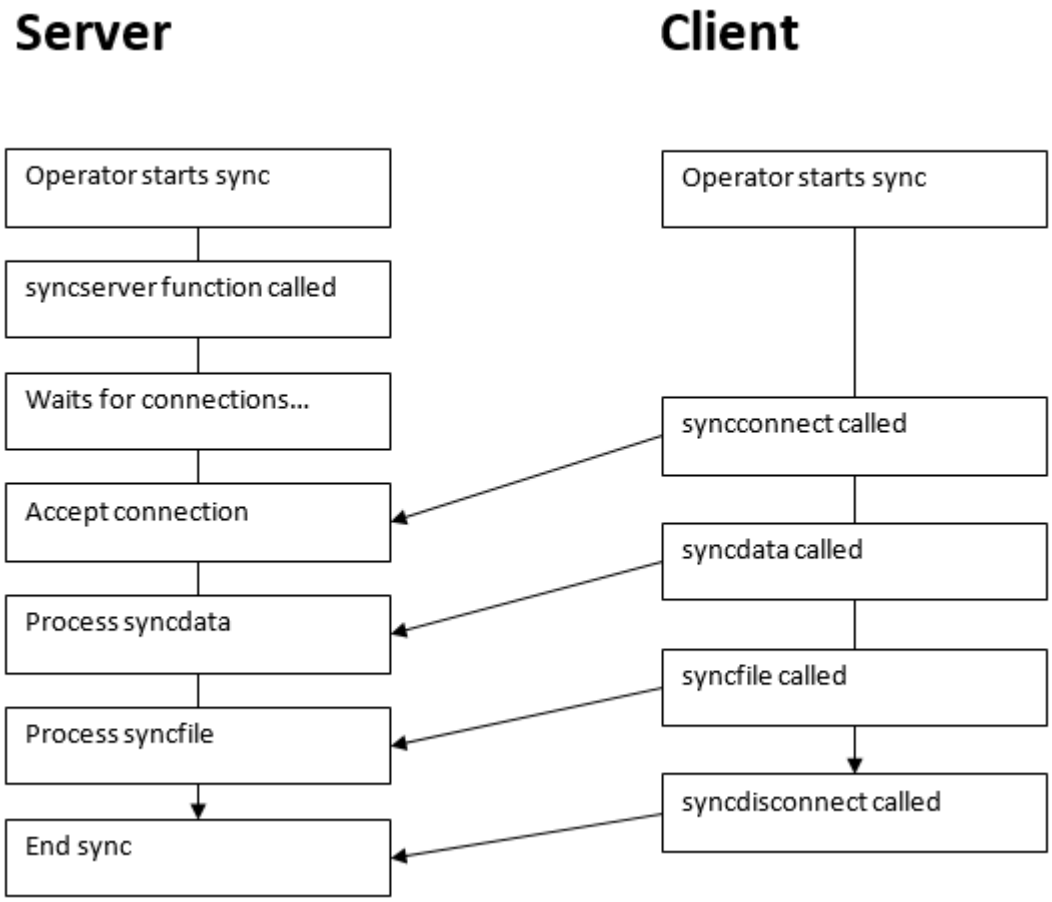
You will generally call syncconnect once to start the session followed by multiple calls to syncdata, syncapp, and syncfile, followed by a call to syncdisconnect. In the example below, we connect to the server, download the latest application files and upload cases from the data file associated with the data dictionary SURVEY_DICT.

```
string ServerUrl = "https://www.myserver.org/api";

// connect to the CSWeb server
if syncconnect(CSWeb, ServerUrl) then

    // send the latest cases to the CSWeb server
    syncdata(PUT, SURVEY_DICT);

    syncdisconnect();

endif;
```

Bluetooth Synchronization from Logic

Bluetooth synchronization is done between two devices. One device, called the client, has logic similar to what is used for web synchronization. The second device plays the role of server. With Bluetooth synchronization, the server is passive. The device acting as the server runs only a single logic function, syncserver, which waits and processes commands from the client device. Only the client device runs syncconnect, syncdisconnect, syncdata, syncfile, syncparadata, syncapp, and syncmessage.

The client and server devices must be in close physical proximity in order to connect to each other. The server device must first execute the syncserver function which causes it to wait for connections from nearby devices. Then the client device executes syncconnect which initiates the connection between the two devices. The client then executes one or more calls to syncdata and/or other sync functions, followed by a call to syncdisconnect, which ends the session. This requires that the two operators using the devices coordinate to start the synchronization routines on the two devices at roughly the same time.



The following example shows the logic that would be written for the client and for the server:

Client Logic:

```
// connect to supervisor via Bluetooth
if syncconnect(Bluetooth) then

    // send the latest cases to the supervisor
    syncdata(PUT, SURVEY_DICT);

    syncdisconnect();

endif;
```

Server Logic:

```
// start peer-to-peer Bluetooth server
syncserver(Bluetooth);
```

Application Deployment over the Internet

The synchronization functions all assume that your CSPro application is already installed on the interviewers device. This means that it is not possible to use the above methods to download a new application for initial installation. While it is possible to simply copy the application to the interviewers device by connecting it to a PC via USB, in some cases it is preferable for the interviewer to download applications over the Internet. To download an application onto a device, use Add Application from the menu in the *Entry Applications* screen in CSEntry on an interviewer's mobile device. To package and upload an application to a server from where it can be downloaded, use the Deploy Application tool. In addition, once an application has been installed on a mobile device using the Deploy Application tool, it can easily be updated by re-deploying the application to the server and then using Update Installed Applications from the menu on the mobile device or by using `syncapp` in application logic.

Troubleshooting

Many errors that you encounter during synchronization are caused by lack of network connectivity. An easy way to check if an error is a problem with your CSPro program rather than a general network error is to try to connect to the sync server in the web browser on the device you are trying to synchronize. For example, if you are trying to synchronize data on a tablet using Dropbox, try to connect to https://www.dropbox.com in the web browser on your tablet. If you are unable to connect in the web browser then there is a problem with your connection to the Internet rather than a problem with your CSPro application.

To get additional details on synchronization errors you can look at the sync.log file. This files lists all synchronization oprtations along with any errors encountered. This includes errors in uploading/downloading applications in Application Deployment and data download in Data Viewer. On Android devices you can find the sync.log file in the CSEntry directory on your tablet or phone. On Windows you can find the file in the directory *%AppData%\CSPro*. On Windows you can also locate the sync.log file from the Help menu in CSPro Designer under Troubleshooting. When asking for technical support with synchronization issues, please attach the sync.log file from your device to your email or forum post.

See also: Simple Synchronizations, SyncConnect Function, SyncData Function, SyncApp Function, SyncFile Function, SyncParadata Function, SyncMessage Function, SyncTime Function, SyncDisconnect Function, SyncServer Function

Paradata

Introduction to Paradata

CSPro can automatically collect paradata for your application. Paradata is defined as "data about the process by which the survey data [was] collected." In CSPro this includes information about fields and values entered, error messages encountered, and the state of the data collection device.

The paradata log, with the extension .cslog, is not a typical data file but is instead a SQLite file. Advanced users with an understanding of SQL can query the file from tools outside of CSPro. Alternatively, there is a tool, the Paradata Viewer, which allows you to look at the contents of paradata logs. If you have several paradata logs coming from multiple devices, you can combine these using another tool, the Paradata Concatenator. You can also query the paradata log from within CSPro using the `paradata` or `sqlquery` functions. Reports can be generated from paradata by creating a templated report. Paradata can be synchronized between devices or servers by using the `syncparadata` function.

CSPro collects information about many different kinds of events, which are described in more detail in the Paradata Properties page. You may wish to collect information about only a subset of possible events.

Enabling Paradata Collection

To turn on paradata collection, you must:

1. Open your application and then select **Options** -> **Application Properties**. You will then be able to modify the Paradata Properties.
2. Indicate that you want to collect paradata events and specify any additional options. By default, when creating a new system-controlled data entry application, paradata will be collected, though with a limited set of options.
3. The action in step #2 only indicates that CSPro *may* collect paradata. To actually turn on the collection, you must specify a filename for the paradata log when running your application. CSPro may suggest a default filename, which you can modify.

Paradata can be collected for data entry or batch edit applications. The paradata log is flexible and can be used simultaneously by multiple applications or users.

Uses of Paradata

If you conduct a pilot census or survey, you may find it useful to collect paradata on all devices. By analyzing this data, you may be able to improve and streamline operations for the actual census or survey. You can use paradata to answer certain questions about your application, such as:

- Which questions in my survey take the longest to collect? Should I to change the wording of the question to make it simpler?
- What out-of-range values are interviewers entering? Should these be valid values?
- What error messages are being triggered the most frequently?
- Where were interviewers located at a given time?
- How long did interviewers work?
- Which tablets had the best battery life?

This is just a sampling of questions that can be answered based on the events that are stored in the paradata log. In addition to storing previously unavailable data, the paradata log also contains all of the information that was traditionally saved to the listing file, the operator statistics log (.log), and the imputation frequencies listing file.

See also: Paradata Properties, Paradata Function, SyncParadata Function, LogText Function

Multiple Language Applications

CSPro supports the creation of applications that target multiple languages. The main areas where you can customize text include:

- Question text: In a data entry application, the question text for each field can be defined in multiple languages.
- Dictionary labels: Dictionary labels can be specified in multiple languages. When a field's screen text is linked to the dictionary item, then the form text will also change when a data entry application's language changes.
- Messages: Messages displayed or created using functions such as `errmsg` and `maketext` can be defined in multiple languages.
- Logic string literals: Using the `tr` function, string literals can be be specified in multiple languages.

In addition to the above functionality, there are several logic functions for working with multiple language applications, including `getlanguage`, `setlanguage`, and `OnChangeLanguage`.

The default starting language can also be specified in the application's PFF file using the Language parameter.

Mapping

CSBatch and CSEntry support displaying interactive maps on both Android and Windows devices. There are two ways to integrate maps into an application:

- Use the `map` object in your application logic to display a map with markers and buttons that can be customized using map functions.
- Display the case listing on a map, where each case in the data file is represented by a marker on the map.

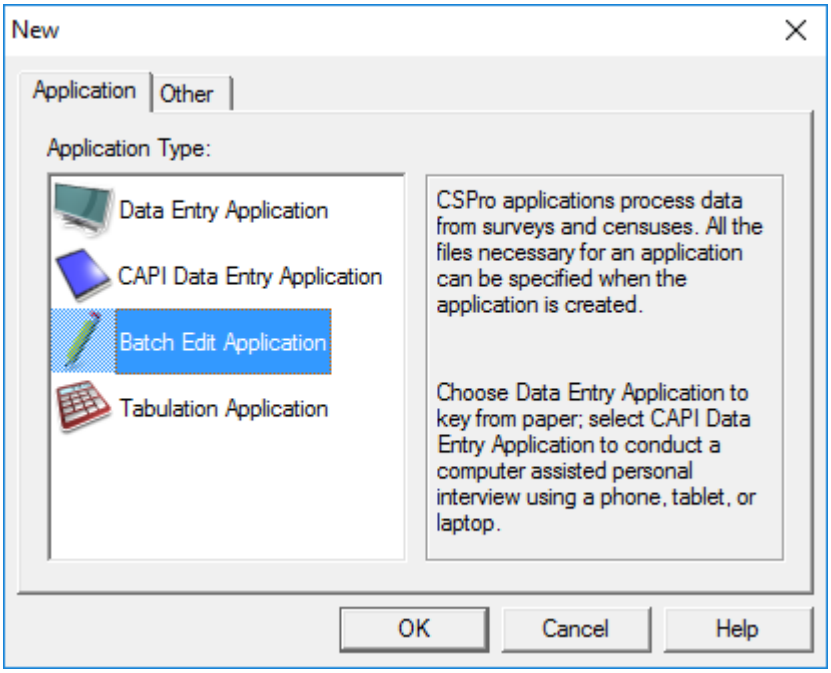
Maps can use Google Maps for base maps if the device has an Internet connection or you can supply an offline map file to use if there is no connection.

See also: Map Object, Mapping Properties, Base Map Specification, Offline Maps, Display the Case Listing on a Map

How To ...

Create a CSPro Application

Click on the toolbar; or from the **File** menu, select **New**; or press **Ctrl+N**. The following dialog box will appear:



Select the type of application you want to create and press **OK**. You can create one of the following applications:

- Data Entry Application: To enter data, generally from a paper questionnaire.
- CAPI Data Entry Application: To enter data, generally by using computer-assisted personal interviewing (CAPI).
- Batch Edit Application: To detect and correct errors in data file.
- Tabulation Application: To produce publication ready tables.

You will then be prompted to enter the file name of the application. Enter the file name of the application and press **Create**.

The File Associations dialog will be displayed. Specify the names of other files that make up the application, such as the data dictionary file and any additional external lookup dictionaries.

Open an Existing Application

Click on the toolbar, from the **File** menu select **Open**, or press **Ctrl+O**. Select from the "Files of type" at the bottom of the dialog box. In CSPro you can open either an application or a file. Select the name of the file you want to open. Each time you open or create an application or file, it is added to the files tree. Any other files belonging to the application or file will also be opened and added to the appropriate trees.

You may open a data dictionary and make changes to it, even if it already belongs to an application. Be aware that if you later open an application to which it belongs, CSPro will automatically make necessary adjustments in other files. For example, if you delete or rename a dictionary item, then later open an application that contains the data dictionary, any corresponding fields on forms will be deleted.

You may open a forms file and make changes to it, even if it already belongs to an application. However, you will not have access to the associated Logic file and you will not be able to run it.

Any changes you make to applications and files are not made permanent until you save the file or application that you modified.

Change the View

In the File Tree

The file tree can display the label or name of the file(s) and the contents of the applications currently open. To toggle between labels and names in trees, open the **View** menu select **Names in Trees**, or press **Ctrl+T**. A check mark appears next to the **Names in Trees** menu item when names are displayed instead of labels. The setting affects all the trees.

If you would like to view both the name and the label for each item in the tree, select the **Append Labels to Names in Tree** option.

Screen

To toggle between trees on left (i.e., split screen) and full screen, open the **View** menu and select **Full Screen**, or press **Ctrl+J**. A check mark appears next to the **Full Screen** menu item when the display is in full screen mode. This setting affects all applications.

Designer Font Preferences

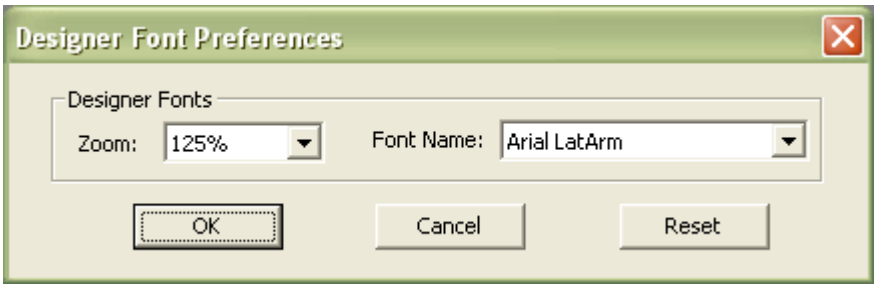
ANSI Font Handling

Versions of CSPro prior to 5.0 supported some languages that used non-Latin alphabets such as Armenian and Russian. With the switch to Unicode support in CSPro 5.0, this feature has been removed. Old applications that used these language specific ANSI fonts will no longer look correct in the CSPro Designer. For example, this is a value set in an Armenian dictionary:

N	Value Set Label	Value Set Name	Value Label	From	To	Special
	ê»é	SEXVT				
			îÕ³Ù³ñ¹ (m)	1		
			ÎÇÝ (f)	2		
			âÉñ³óí³Í	0		NotAppl

To view these old applications correctly, you can manually change the font used to render the text in the dictionary grid, the dictionary and form tree, and, if the font is monospaced, the application's logic.

With no application open within the CSPro Designer, select **Tools** -> **Preferences** -> **Fonts**.



In this example, an Arial Armenian font has been selected, at an increased zoom rate. After this selection, the dictionary editor displays the contents better:

N	Value Set Label	Value Set Name	Value Label	From	To	Special
☐	Սեռ	SEXVT				
☐			Տղամարդ (m)	1		
☐			Կին (f)	2		
☐			Չլրացված	▯		NotAppl

Ultimately you will want to convert your application to Unicode, as these font preferences are disregarded by CSEntry. Many tools exist to convert ANSI language scripts to their Unicode equivalents.

Font Display Within CSPro Designer

Some users may find that the CSPro Designer renders the Unicode characters for their language at a very small size. By choosing a zoom factor while leaving the font name blank, it will increase the characters to the magnification level chosen.

Change Windows

Cascade

Use this command to arrange multiple opened windows in an overlapping fashion.

Tile Top-to-Bottom

Use this command to arrange multiple opened windows one above the other in a non-overlapping fashion.

Tile Side-by-Side

Use this command to arrange multiple opened windows one beside the other in a non-overlapping fashion.

1,2,…

View displays a list of currently open files at the bottom of the Window menu. A check mark appears in front of the name of the file in the active window. Activate a window by choosing the name of its file from this list.

Add Files to an Application

You may add dictionaries and form files to a data entry application. Additional dictionaries represent data files used by the application, such as lookup files. Multiple forms files are sometimes used in advanced applications. You may not add files to tabulation applications.

To insert a file, from the **File** menu, select **Add File**. Select an existing file or enter the name of the file to be created.

See also: Drop Files from an Application

Drop Files from an Application

To drop one or more files from an application, on the **Files** menu, select **Drop Files**. Select the files you want to drop from the application and press **OK**. These files will be dropped from the application, but will not be deleted from the disk.

See also: Add Files to an Application

Change the Print Page Setup

To change the page headers, footers, or margins within the dictionary or a tabulation application, click Page Setup Button on the toolbar; or, from the File menu, select "Page Setup". The changes will remain in effect until you change them again.

Header

Edit the text to be placed at the top left, top center, and top right of each page. You can use the Date, Time, File, and Page buttons to insert the current date, time, file name, and page number.

Footer

Edit the text to be placed at the bottom left, bottom center, and bottom right of each page. You can use the Date, Time, File, and Page buttons to insert the current date, time, file name, and page number.

Margins

Change the size of the top, bottom, left and right margins. Your printer may not allow margins below certain values.

Page Orientation/Size

To change the page orientation or size, open the File menu and select "Print Setup". In the print setup dialog box make changes to orientation (portrait or landscape) and paper size.

Print All or Part of a Document

To print an entire document click on the toolbar; or from the **File** menu, select "Print"; or press **Ctrl+P**.

To print part of a document select the text you want to print then click on the toolbar; or from the **File** menu, select "Print"; or press **Ctrl+P**.

To preview the printing, click on the tool bar; or from the **File** menu, select "Print Preview."

Save an Application

Click on the toolbar; or, from the **File** menu, select **Save**; or press **Ctrl+S**.

The file associated with the current frame (right side of the screen) will be saved. If that file belongs to an application that is open, the entire application will be saved. If the file belongs to more than one application, CSPro will ask you which one you want to save. In that case, select the file or files you wish to close or save and click on **OK**.

To choose all of the files, click on the **Select All** button. To choose several files, hold down the **Ctrl** key and click on files you wish to select.

See also: Open an Existing Application

Close an Application

From the **File** menu, select **Close**.

The file associated with the current frame (right side of the screen) will be closed. If that file belongs to an application that is open, the entire application will be closed. If the file belongs to more than one application, CSPro will ask you which one you want to close. In that case, select the file or files you wish to close or save and click on "OK."

To choose all of the files, click on the **Select All** button. To choose several files, hold down the **Ctrl** key and click on files you wish to select.

See also: Open an Existing Application

Save an Application with a New Name

To save an application under a new name:

- Make sure the application is showing in the current frame (right side of the screen).
- From the **File** menu, select **Save As**.
- Enter the file name for the new application in the file open dialog box.
- Using the File Associations Dialog specify the names of other files in the application.

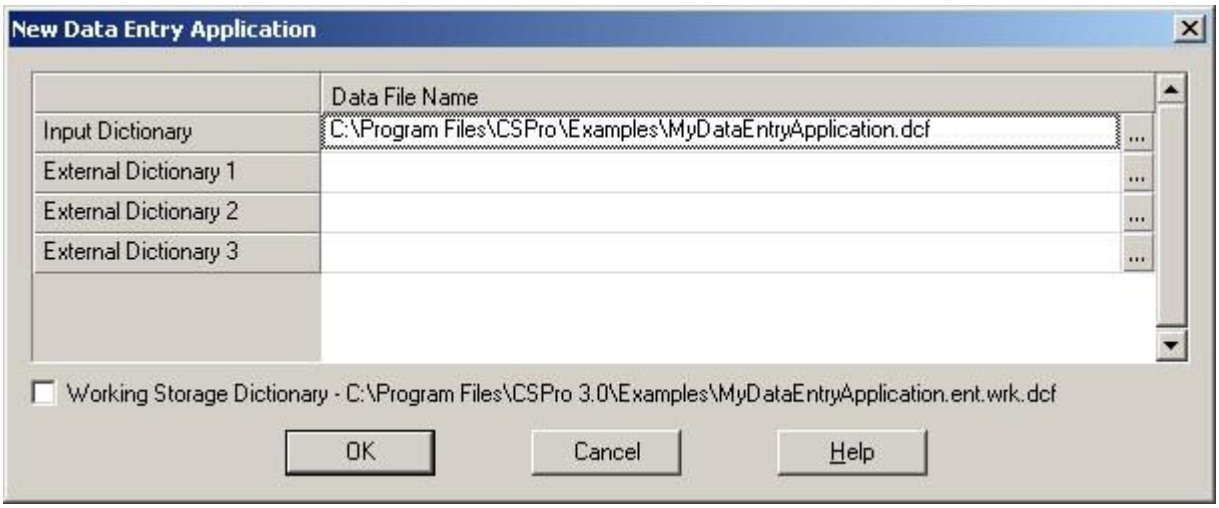
When the Save As is complete you will be editing the new dictionary.

See also: Save Dictionary as New File

Specify Application File Names

New Application

When a new application is created, a files dialog box like the one below is displayed.

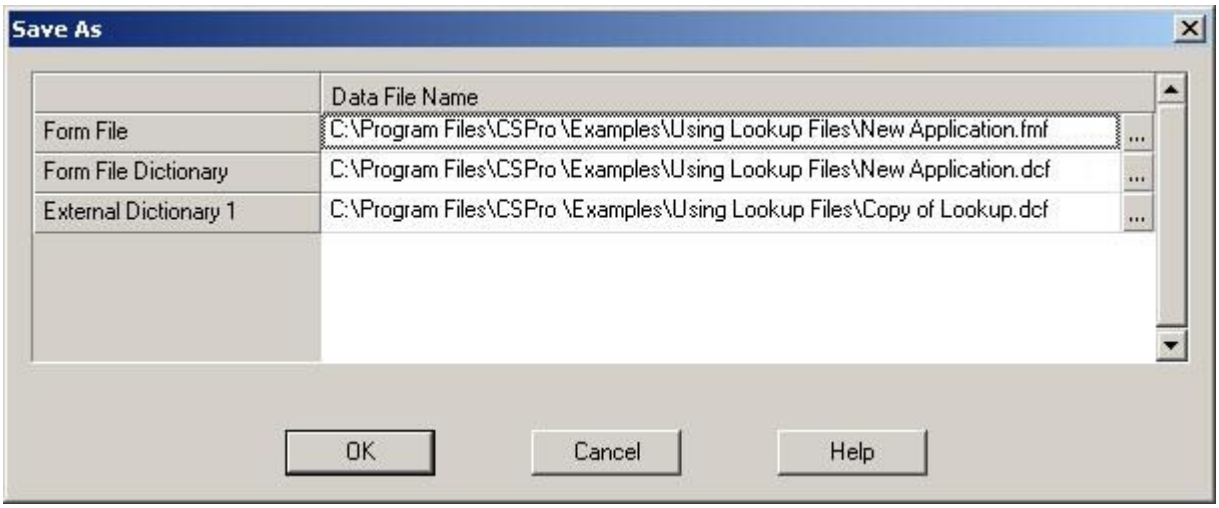


Only the name of the input dictionary must be supplied. A default input dictionary file name is given, but it may be changed. Other external dictionary file names are specified as needed.

If any of the files names specified already exist, those files will be attached to the application. If any of the file names specified are new, those files will be created.

Save As Application

When an application is saved under a new name, a files dialog box like the one below is displayed.



Names for all the files requested must be supplied. Default file names are given, but they may be changed.

If any of file names specified are new, those files will be created for the new application. If any of the files names is the same as the name in the original application, those files will be shared with the new application. If any other of the files names specified already exists, those files will be overwritten.

See also: New Application, Save As Application

Pack an Application

There are instances when it is helpful to collect all the files in an application to:

- Move them to another computer (for example to move data entry applications to all your data entry computers).
- Give the application to a colleague to use.
- Send the application when requesting help.

There is a CSPro tool to perform this function. This tool copies all the files in the application into a ZIP file. To use this tool go to the **Tools** menu and select **Pack Application**.

See also: Pack Application

CSPro Settings

The CSPro Settings dialog is accessed from the CSPro Designer or from CSEntry by selecting **CSPro Settings** from the **File** menu. The dialog has the following options:

Saved Credentials: By selecting Clear Credentials, any cached credentials saved to secure storage will be removed. These credentials include access tokens to CSWeb servers and Dropbox, passwords to FTP servers, and hashed password keys used to open encrypted data files.

Get Help

To contact the CSPro development team with comments, questions, or to report problems, please contact:

International Programs
Population Division
U.S. Census Bureau

4600 Silver Hill Road
Washington, DC 20233

Phone: +1 301-763-1451

Support email: cspro@lists.census.gov

Official website: www.census.gov/data/software/cspro.html

CSPro Users forum: www.csprousers.org/forum

When you contact us, please mention that you are using **CSPro 7.7.2**.

Collaborating on CSPro Development

CSPro 7.7 is not an open source project but it is in the public domain, meaning that it can be used and distributed at no cost. However, if you are interested in collaborating on the CSPro development, there are currently three projects, changes to which are distributed with CSPro.

Multiple Language Runtime Messages

The runtime, system-level messages that CSPro displays are provided in English. Partial translations for these messages are also provided in a few languages, including Chinese, French, Portuguese, Russian, Spanish, and Vietnamese. If you would like to provide message translations, we welcome your support. Please contact cspro@lists.census.gov for details.

The complete list of system-level, English-language messages used by CSPro can be found in the installation folder, ***C:\Program Files (x86)\CSPro 7.7***. The English language message file is named **CSProRuntime.en.mgf**. The other language message files installed with CSPro are named as follows:

Name	Language
CSProRuntime.zh.mgf	Chinese
CSProRuntime.fr.mgf	French
CSProRuntime.pt.mgf	Portuguese
CSProRuntime.ru.mgf	Russian
CSProRuntime.es.mgf	Spanish
CSProRuntime.vi.mgf	Vietnamese

Note: If you add an external message file to your application that begins with "CSProRuntime," the messages in that file will override the system messages that CSPro uses.

Help Documentation

The help documentation that you are currently reading is created using a custom tool, Help Generator. The tool, along with the documentation, is hosted on GitHub:

https://github.com/CSProDevelopment/helps

The repository has all the tools needed to develop the help documentation. The Help Generator tool is built using Visual Studio. Microsoft distributes a free version, the Community edition, that you can use to create the tool. If you do not have Visual Studio, but want to collaborate on help documentation, email cspro@lists.census.gov and you will be sent the Help Generator executable and a guide on modifying the help documentation.

CSPro Examples

The CSPro Examples folder is also hosted on GitHub:

https://github.com/CSProDevelopment/examples

If you want to modify any existing example, or want to add a new example that you think will be useful to the broader CSPro community, you can collaborate on this project.

Data Dictionary Module

Introduction to Data Dictionary

A Data Dictionary describes the overall organization of a data file. In other words, it gives a description of how data are stored in a file. CSPro requires that a data dictionary be created for each different file being used.

Each dictionary will allow you to give text labels for all levels, records, items, and value sets in the file. It also allows you to describe the organization of each record in the file and the characteristics of each item in the record: names, position in the data record, type of data, length, number of decimal places, valid values, and other documentation.

Before you convert the information from a questionnaire to computer-readable form, you usually create a data dictionary. You can also create a data dictionary for an existing data file if you have a description of its contents showing the location of each item.

CSPro requires that a Data Dictionary be created for each different file being used.

This section contains the following information:

- Organization
- Dictionary Concepts
- Levels
- Records
- Items
- Value Sets
- Values
- Relations
- Data Dictionary Application
- How to ...

Organization

Questionnaire and Dictionary Organization

Questionnaire: Form

A questionnaire is a collection of information relating to the same unit of observation (such as a household, person, or factory). A typical questionnaire consists of an identification section followed by other sections grouped by topic. Each section includes a set of related questions, each of which is associated with a list of response values. A questionnaire usually constitutes a case.

Questionnaire: Section

Any type of questionnaire will have an identification section that uniquely identifies the form, as well as one or more sections on different topics. Some sections may occur once per questionnaire while other sections are repeated many times. For example, in a typical housing and population census, a questionnaire would contain a section for the housing questions, and a section for the population questions. The questions in the housing section will be answered once per questionnaire [household], while the questions on the population section will be answered by every person in the household. If the census is collecting information on vacant housing units then the questions on the population section will not be answered. In a school survey, for example, the questionnaire would have an identification section and only one section to collect basic information for each student. The questionnaires for the different students are not related.

Questionnaire: Identification

The identification section identifies the questionnaire, usually with numeric geographic codes. The combination of identification codes (such as province, district, village, household) on a questionnaire uniquely identifies the form. These are the codes you would need to locate a specific questionnaire.

Questionnaire: Questions

The basic element of the questionnaire is the question. Each section of the questionnaire contains a set of one or more questions being asked for this census or survey.

Questionnaire: Responses

The valid options in response to a question are usually listed in the questionnaire. Some responses are quantitative, such as "size of farm" or "age of person," and some are qualitative, such as "relationship to head of household" or "crop grown." Responses can be numeric or alphanumeric. Most descriptive responses are equated to numeric codes that are placed on the questionnaire. However, some descriptive responses remain as alphabetic text.

Data Dictionary: File

In the data dictionary a topic section is usually equivalent to a record. A record includes data items (questions) that are associated with one or more value sets (response values). Records with the same identification codes (i.e., questionnaire IDs) comprise a single questionnaire.

Data Dictionary: Records

Similarly, a data dictionary may have records that occur once and records that repeat many times. The typical housing and population census will have one housing record and as many population records as people in the household; the housing and population records will equate to one questionnaire, and these records are related. If our study permits vacant housing units, then the data file will not include a population record for an unoccupied housing unit. In the school survey each questionnaire will only have one record, and there is no relationship between records in the data file. This type of data file is known as a "flat data file." The records for the different sections will most likely have different

structures. Using the data dictionary module you can identify each record structure using a record type name and code.

Data Dictionary: File Identification

Similarly in the data dictionary, you first define the identification items to uniquely identify the questionnaire. These data will appear on every record in a data file, as they are "common" to all of the records. In a data file, if a group of questionnaire ID fields uniquely identify the unit under observation, then those records make up one questionnaire. In the case of the student survey the student identification number could serve as the questionnaire identification.

Data Dictionary: Data Items

In the data dictionary, the data item contains the response to a question, and is therefore the most basic element of a questionnaire; `AGE`, `INCOME`, and `CROP_CODE` are all examples of items. Related items should be placed in the same record. And, just like records and levels, data items possess properties (such as a unique name, label, etc.). Items in data files must be fixed format, that is, items must have the same starting position and length in every record where they occur.

Data Dictionary: Value Sets

In the data dictionary, the responses in the questionnaire are defined as value sets. A single value set can contain one or more values. The valid values can be defined as individual values or ranges of values.

See also: Record Description, Item Description, Value Sets Description

Data File Type Structure

There are two basic types of data file structures: Those that contain single-record questionnaires, or those that contain multiple-record questionnaires. The following is a brief description of these two types.

- **A single record type per questionnaire**
  In a single-record data file, each line of data from the data file equates to a distinct questionnaire. This means there is no relationship between records in the data file—each record stands on its own and is distinct from another.

  One usage for a single-record questionnaire would be a student survey at a university. In this scenario, a single record would be created based on the student. The student identification number could serve as the questionnaire identification. A data file produced from this type of dictionary is known as a **flat data file**. Example:

  ```
  00011122122    ↵ (first student)
  00021122122    ↵ (second student)
  00031122122    ↵ (third student)
  ```

  `blue` text refers to the student identification number.
  `black` text describes the individual data items for each specific record.

  Notice there is no need to have a record type identifier.

- **Multiple record types per questionnaire**
  In a multiple-record data file, several lines of data (and therefore several records) from the data file equate to one questionnaire. This means there is a relationship between records in the data file—and information identifying them as such in the form of Questionnaire Ids will be needed.

  For example, in a typical housing and population census, a questionnaire might consist of the following records:

  - one housing record
  - multiple (zero or more) population records

  For a given questionnaire there would be one or more population records for one household record, dependent on the number of people in the household. However, if you allowed vacant housing units, then those questionnaires would not have any corresponding population record.

A sample (and recommended) file structure could be as follows (not all fields are defined for this example):

```
11010011122122    ↵ (household with 3 persons)
2101001120109196138
2101001212105196732
2101001311707199207
11010031211212    ↵ (vacant household)
11010021111121    ↵ (household with 2 persons)
2101002110716193069
2101002220812192871
```

In the example above:

Red	text refers to the record type. In our example, 1 is a household record, and 2 is a population record.
Blue	text refers to the (Id Items). Note that the numbers are unique for each questionnaire: the 101001 household contains three people whereas the 101002 household contains two people.
Black	text describes the individual data items for each specific record.

A questionnaire designed for an agricultural census might consist of the following records:

- one farm household record
- multiple (one or more) crop records
- multiple (one or more) farm worker records

A questionnaire for a reproductive health survey might consist of the following records:

- one record for data on the woman
- multiple (zero or more) children-ever-born records
- one contraceptive use record
- one immunization record

Dictionary Hierarchy

A data dictionary is structured in a hierarchical order. The top hierarchy is the case, followed by the level, then record.

Case

A case is the primary unit of data in the data file. A case usually corresponds to a questionnaire. However, some complex applications might have a hierarchical set of questionnaires, or many **levels**. For example, the main questionnaire may consist of a household roster and other household information, and there may be a separate questionnaire for each woman in the household. The data entry application may then contain two levels — one for the household and one for each woman in the household. The set of forms corresponding to the household make up level one. The set of forms corresponding to each woman make up level two. Each case would consist of two type of questionnaires: a single level one and a variable number of occurrences for level two.

Most applications consist of a single level.

Level

A level is a type of questionnaire. By default, all new dictionaries have one level. This is normally sufficient to describe, for example, a population or agriculture census. However, if you have a hierarchically-structured set of questionnaires, you will probably need to use additional levels. A level can have many **records** corresponding to different record types.

Record

A record usually corresponds to a section of a questionnaire, and consists of a group of related data **items**. For example, data items related to housing would form a housing record; data items related to individuals would form the population records; data items related to production of a particular crop would form the crop record, and so on. If a dictionary contains more than one record, then you must use the record type item to identify one record from another in the data file.

See also: Level Description, Record Description

Dictionary Concepts

General

Labels

Labels are descriptive text used to identify the dictionary and its elements. Labels are required for the dictionary and most of its elements. Labels can contain any printable character and spaces and can be up to 255 characters long.

The dictionary tree displays either the labels or names of dictionary elements. You can press **Ctrl+T** or from the **View** menu, select **Names in Trees** at any time to toggle between labels and names.

Names

Names identify the dictionary and its elements when they are referenced in CSPro procedures. Names are required for the dictionary and most of its elements. Names consist of upper case letters (A-Z), digits (0-9), and embedded underscores (_). The first character of a name must be a letter; the last character cannot be an underscore. Names can be any length but must not be CSPro reserved words.

Examples: `SEX, RELATIONSHIP, MOTHER_ALIVE, Q102_AGE_CHILD`

The dictionary tree displays either the labels or names of dictionary elements. You can press **Ctrl+T** or from the **View** menu, select **Names in Trees** at any time to toggle between labels and names.

Names cannot be duplicated within a dictionary. However, the same name can be used in different dictionaries, and in some cases, it maybe desirable to do so. For example, when using lookup files, you may prefer to use the same identifier name across dictionaries, rather than having 2+ spelling permutations for the same variable (i.e., ID_PROVINCE in the main dictionary versus LU_PROVINCE in the lookup file). This also facilitates cutting and pasting the variables (and their accompanying value sets) from one dictionary to the other, without having to worry about renaming the variables.

Note: When repeating variable names among dictionaries, you will have to **fully qualify** the variable names when referring to them in logic by prefacing them with their unique dictionary's name. This is necessary so that CSPro knows which variable you are referring to. Failure to do so will generate a compiler error. For example, using the ID_PROVINCE and LU_PROVINCE names above, logic that would appear as follows:

```
LU_PROVINCE = ID_PROVINCE;
```

would need to change to:

```
LOOKUP_DICT.PROVINCE = MAIN_DICT.PROVINCE;
```

Notes

Notes document the dictionary and its elements. The designer may create notes for the dictionary as a whole and/or any of its elements: levels, records, items, value sets, values. Notes may contain any printable character and spaces, and can be up to 65,000 characters in length.

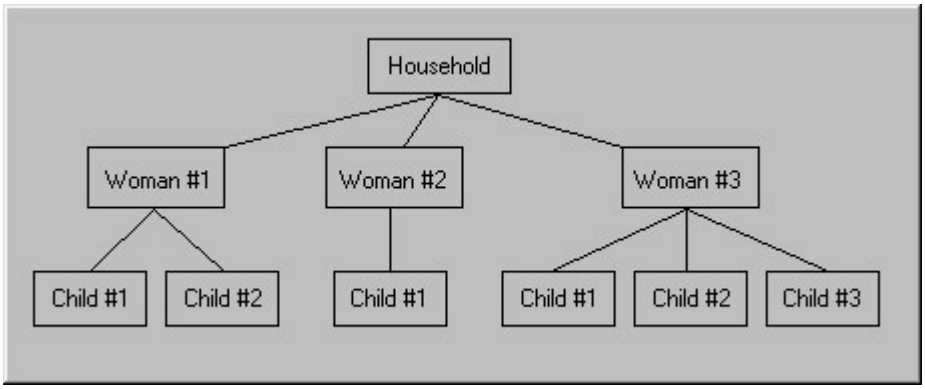
Levels

Level Description

A level is a type of questionnaire. By default, all new dictionaries only have one level. The maximum number is three. A good use for a three-level dictionary might be a reproductive health survey that has the following questionnaires:

- A housing questionnaire
- A questionnaire for each woman of reproductive age in the household
- A questionnaire for each woman's child in the household

A pictorial representation of this scenario is as follows:



In this example, you would want each child to be associated with its mother, rather than the household record. If you were to structure your dictionary in a single level, there would be no way to easily identify which mother and child(ren) belonged together during data entry or during tabulation. To accomplish this, you would want to design your dictionary with three levels, each level containing a single type of record, as follows:

Level 1
Household Record
Level 2
Woman of Reproductive Age Record
Child Record (one for each child)

In the Forms Designer you will be required to place each record's data on different forms. However, this facilitates the desired data entry behavior. You will first be asked to enter information from Level 1, i.e., the household. After completing the household form(s), you will then enter information for the first woman and her children (Level 2). When data entry is finished for this woman record the keyer will advance to the child record, and enter information for each child (if any) for this women.

If there are no further children (or no children at all for this female), finish the level by pressing **F12** (EndLevel Occurrence) and resume entering information for the second woman and her children. Continue in this manner until all women and their children have been entered for the household—when finished, press **Ctrl+F12** (EndLevel) from the Woman Form to complete data entry for this case.

Keep in mind that, when using more than one level, there are implications with respect to the order of executing logic in a data entry application or in a batch edit application.

See also: Dictionary Hierarchy, Level Properties, Add or Modify Levels

Level Properties

Level properties are visible when the dictionary has been selected in the tree tab. To reflect your intended usage for a level we suggest you change the level properties pressing **Ctrl+M**, which will activate the appropriate entry in the right-hand screen.

Property	Meaning
Label	A descriptive text label which identifies this level.
Name	The name given to this level for use in the CSPro language procedures.

See also: Dictionary Hierarchy, Level Description, Add or Modify Levels

Records

Record Description

A record is a group of related data items. In the process of creating a record to define (a portion of) the questionnaire, you will also be defining the physical layout of the data file. For example, suppose your (very simple) population record looks like the following (only item name, starting position, and length properties are shown; starting positions show that ID items occupy the first 9 positions in the record):

```
Item Name     Start Pos  Length
RELATIONSHIP  10         1
SEX           11         1
AGE           12         2
```

If an operator had keyed a questionnaire for a 35-year-old female (Sex = 2) head of household (Relationship = 1), you would see a line in the data file, corresponding to the population record defined above:

```
12345678901234567890 (position)
--------------------
         1235        (line in data file)
```

In deciding on a file structure, there is often the choice of defining a record type which occurs once within a questionnaire but contains repeating sets of data, or to define a record type which occurs multiple times within a questionnaire, each with a single occurrence of the data. The application designer should take into consideration the amount of information that recurs and the probable number of occurrences.

A common example in a Housing and Population Census is information about deaths in the household during the 12 months prior to the census. If this information (usually sex and age at death of the deceased) is collected during enumeration, the expectation is that 95% of households will have no more than one, or at most two, deaths during the previous 12 months. With this volume of information, it would be practical to have one record type that occurs once within the questionnaire and allows for repeating occurrences of the data, since it is unlikely that even the maximum number of occurrences, multiplied by the number of positions occupied by each occurrence, will exceed the length of the already-existing household and population records.

However, in the case of an agricultural survey, a section on crops may include questions about acreage planted, yields, etc., whose cumulative length for each crop mentioned may be quite large in relation to other records in the file. In such a case, it would be more practical to define a record type that occurs multiple times within the questionnaire. Within each occurrence of the record would be found the information relating to one specific crop.

See also: Dictionary Hierarchy, Record Properties, Add or Modify Records

Record Properties

You can view a record's properties by selecting the questionnaire to which it belongs (via the dictionary tree tab). You may change the default record properties by positioning the cursor in the right window and pressing **Ctrl+M**.

Property	Meaning
Label	A descriptive text label which identifies this record.
Name	The name given to this record for use in the CSPro language procedures.
Type Value	The record type value (code) that identifies this kind of record.
Required	Must a questionnaire contain this kind of record? (Yes/No)
Max	The maximum number of times this type of record can appear in any one questionnaire.

See also: Dictionary Hierarchy, Record Description, Add or Modify Records

Record Type

The Record Type is an alphanumeric item that uniquely identifies a dictionary record, and therefore helps describe your data file's organization.

If your dictionary contains more than one record, CSPro needs to be able to differentiate one record from another in the data file. The Record Type provides the means for doing this.

For example, a typical Housing and Population census data file would most likely have a housing record (describing details of the living unit) and a person record (to describe details on each individual in the household). You could assign a Record Type of '1' to the Housing record and '2' to the Person record to distinguish between them.

If your dictionary contains only one record, you do not need to use a Record Type. Therefore, you can 'reclaim' the location that was set aside for the Record Type as follows:

1. Select the (ID Items) set or the one-and-only record your dictionary contains from the dictionary tree.
2. In the view on the right, you'll notice the first line is (record type). Only three values are used, Starting Position, Length, and Data Type. Of these three values, you can only modify the start position and length. Change the length to 0. This will effectively "remove" the record type. (You can always reinstate it later by resetting the start position and length to non-zero values).

Similarly, if you would like to modify the length of the Record Type, proceed as above.

The record type value is always alphanumeric. Upper- and lowercase letters are distinct Record Type values (i.e., 'A' is not the same as 'a'). Blank is also a valid Record Type value.

See also: Record Description, Record Properties

Required Record

One of a record's properties is whether or not the record is **required**; the options are **Yes** or **No**. If a record is required (**Yes**), it means that for a given questionnaire, there must be at least one occurrence of this record. If there is not at least one occurrence of this record, the questionnaire will not be complete and the system will issue an error message to inform the keyer. If the record is not required (**No**), the questionnaire may or may not contain an occurrence of this record. The questionnaire can be considered complete without an occurrence of this record.

Suppose you are designing a dictionary for a census. You'll probably have at least two types of records: one for the household, and one for each person in that household. You can have four scenarios:

- If you allow vacant housing units (i.e., you collect information on unoccupied housing units), then the household record **is** required and the person record **is not** required.
- If you allow homeless people, then the household record **is not** required and the person record **is** required.
- If you allow homeless persons and vacant housing units, then neither the housing record nor the population record will be required record types, but because the two conditions will never occur simultaneously, you will never have a questionnaire without one of the other type of record.
- If you allow neither homeless persons nor vacant housing units, then both the housing record and the population record will be required record types. This means that a valid questionnaire will always have one housing record and at least one population record.

See also: Record Description, Record Properties

Maximum Number

This record property specifies, for the given record, the maximum number of occurrences of that record allowed in one questionnaire.

For example, suppose you are designing a dictionary for a census. You will probably have at least two types of records: one for the household, and one for each person in that household. There should be only one occurrence of the household record, but for the person record you will of course need more than one occurrence, as there will likely be more than one person in a household. Thus, the maximum for the person record could be 25, if limiting yourself to a family unit, or larger, if enumerating group facilities (military barracks, hospitals, mental institutions, etc.).

The maximum number of occurrences that may be specified for any record is 9,999. However, for greater program efficiency we recommend that you never have this many occurrences and that you keep the maximum to the lowest value that is appropriate for your particular application.

See also: Record Description, Record Properties

Items

Item Description

A dictionary item describes the response to a question, or can help identify areas of the questionnaire. The dictionary item is the most basic element of a questionnaire: age, income, and crop code are all examples of items. Generally, related items should be placed in the same record; for example, questions appearing in a single table/roster that are asked of all household members would be candidates for inclusion in the same, repeating, record (possibly named PERSON_REC). Likewise, questions regarding the characteristics of a residential living quarters might all be placed in the same record (possibly named HOUSE_REC).

While data items have some of the same properties as records and levels (unique name, descriptive label), dictionary items also have several additional properties which are unique to items.

Items can also be redefined into subitems; for example, breaking up a date of birth question from a single, 8-digit field into its constituent parts (day, month, and year).

See also: Identification Items, Item Properties, Add or Modify Items

Identification Items

Identification items (also referred to as **ID items**) are those data items that uniquely identify the questionnaire/case. If more than one dictionary level is used, they define the hierarchical levels at the level of definition and below. ID items usually include geographic items, such as Province, District, or Enumeration Area, as well as other unique identifiers, such as Survey ID or Household number. These data will appear on every record in a data file, as they are "common" to all of the records. The minimum number of ID items is 1, and the maximum number of allowable ID items is 15.

You will typically want to structure your dictionary so that the ID items begin in column 2; in this way they will precede all records' data items, and follow the data file's record type identifier (as column 1 is usually reserved for the type).

For a pictorial representation of a data file's layout, press **Ctrl+L** from within its dictionary (L=layout). Pressing **Ctrl+L** a second time will toggle this view off.

See also: Item Description, Item Properties, Add or Modify Items, Subitems

Subitems

Subitems allow items to be broken up into smaller pieces, or across broad categories. In this respect, they let you redefine data items and refer to the same data field in several different ways. The start position of a subitem must be within its parent item.

One useful application of subitems involves date and time fields. A date item, for example, could be referred to as a single 8-digit entity: YYYYMMDD. However, this does not allow you to easily manipulate or refer to a portion of the date (such as the year, month, or day itself). Suppose you had the following definition for date (for demonstrative purposes, only selected item properties are shown):

Item Label	Item Name	Start	Len	Item Type
Date of birth	DOB	20	8	Item

To redefine this item into subitems, you only need to add the following subitems:

Item Label	Item Name	Start	Len	Item Type

Year of birth	DOB_YEAR	20	4	Subitem
Month of birth	DOB_MONTH	24	2	Subitem
Day of birth	DOB_DAY	26	2	Subitem

Another reason for using subitems is to make data references available across larger categories. Censuses and surveys often have items of three or four digits in length representing categories such as industry, occupation, or ethnicity. For example, looking at occupation codes:

- the full value refers to a very detailed occupation, such as bus driver
- The first digit alone refers to the 'major' division, such as 'public service'
- The first two digits together refer to a more detailed 'major' division, such as 'public transportation'

It may be useful to test the ranges with the CSPro language at the item level. In tabulation applications, tables can be made at the major (1- or 2-digit) or minor (3- or 4-digit) divisions. The following example could represent part of an economic survey:

Item Label	Item Name	Start	Len	Item Type
Occupation	OCCUPATION	45	4	Item
Occupation, Major	OCC_MAJOR	45	1	Subitem
Occupation, Sub-major	OCC_SUBMINOR	45	2	Subitem
Occupation, Minor	OCC_MINOR	45	3	Subitem

Note: Identification items cannot have subitems.

See also: Item Description, Identification Items, Item Properties, Add or Modify Items

Item Properties

You can view an item's properties by selecting the record to which it belongs (via the dictionary tree tab). When creating an item, the following fields must be set:

Property	Meaning
Label	A descriptive text label that identifies the item. It is used as default field text in data entry forms and in default titles in tabulation.
Name	The name given to this item for use in CSPro language procedures
Start	Indicates the starting position of the item within the record.
Len	Indicates the length of the data item (i.e., the number of characters necessary to represent the values for the item).
Data Type	Indicates the type of data (numeric or alphanumeric) that will be found in the item.
Item Type	Indicates whether the item is or is not subordinate to, or part of, another item. If the item is part of another item, it is considered a "subitem". If not, it is identified as an "item". **Identification items cannot have subitems.**
Occ	The number of times this item will repeat within the record. The default value is "1". **Identification items cannot have multiple occurrences.**
Dec	The number of decimal places (if any) in the item. The default number of decimals is "0". **Identification items cannot have decimals.**
Dec Char	This specifies whether the item should be stored in the data file with an explicit decimal character. This applies only to items or subitems which have been defined with the "Dec" property greater than zero (i.e., Dec >= 1).
Zero Fill	This item property states whether the numeric data item should be stored with leading zeros or blanks. By default, all numbers are positive unless defined otherwise. Positive numbers have no sign. Negative numbers will have a leading minus (-) sign. The minus sign will be displayed in the data file immediately to the left of the value. If the item is zero-filled, the minus sign will be displayed in the left-most position.

Press the **Esc** key to quit modifying without making changes. Press **Ctrl+Enter** to finish making changes. Use undo if you completed the modification incorrectly. There is no limit on the number of items within a record.

See also: Item Description, Subitems, Identification Items, Add or Modify Items, Dictionary Property Panel

Starting Position

This item property indicates the starting location of a data item. In conjunction with the Length property, it specifies the location of the item in a record. In absolute positioning mode, you cannot give a starting position that will cause the item to overlap with another item.

The start position of a subitem must be within its parent item (the previous item).

See also: Select Relative or Absolute Positioning, Item Properties

Length

This item property indicates the total length of the data item (i.e., the number of characters necessary to represent the values for the item). In conjunction with the Start property, it specifies the location of the item in a record. In absolute positioning mode, you cannot give a length that will cause the item to overlap with another item.

The maximum length of a numeric item is 15 digits. The maximum length of an alpha item is 255 characters.

See also: Item Properties

Data Type

This item property specifies the type of data (numeric or alphanumeric) that will be found in the item. The default item type is numeric.

- **Numeric** items can contain numbers or blanks, and they may be negative or positive in value. Numeric values will be right-justified and, if requested, zero-filled.
- **Alphanumeric** items can contain any combination of characters, letters, numeric digits, blanks, or special characters. These values will be left-justified and are blank-filled, whether or not zero-fill has been selected. Declaring M or F for SEX is an example of an alphanumeric value.

Some responses are quantitative, such as size of farm, and some are qualitative, such as relationship to head of household. Most descriptive responses, such as "head of household," are given numeric codes that are placed on the questionnaire. However, some descriptive responses remain as alphabetic text.

Numeric responses can be discrete values or continuous values. An example of a discrete value is sex, 1 (male) or 2 (female). An example of a continuous value is yearly income, which can range from zero to a value limited only by the number of digits permitted for the response. A discrete value may be used to represent a grouping of continuous values. For example, when asking income, one may be asked to select from a choice of ranges of incomes rather than specify the exact income. Thus, the possible responses to the income question could, for example, be a code between 1 and 10.

See also: Item Properties

Occurrences

This item property defines the number of consecutive repetitions of the item in the data record. The dictionary will reserve space equal to the product of the length of the item times the declared number of occurrences for the item.

For example: A census collects information on births and deaths, and each questionnaire can list the ages of up to a dozen household members who died during the past year. By defining an item "Age at death" with a length of 2 digits and 12 occurrences, the dictionary will reserve a location 24 characters in length for this item.

Be aware that if fewer than 12 people died in the household, then the unused portion of this item will be blank. If you have several items that use occurrences and they are often unused, you are increasing the size of your data file. Therefore, you should always specify the number of occurrences with care.

If an item has multiple occurrences, then its subitems may not have multiple occurrences. Conversely, if a subitem has multiple occurrences, then its parent item may not have multiple occurrences.

ID items cannot have multiple occurrences.

See also: Item Properties

Decimal Places

This item property lets you specify how many digits of the numeric item represent the decimal portion of the item. CSPro does not expect the decimal point to be in the data file; if your data file does contain the decimal point, you will need to set the decimal character property. Therefore, the length of the item is not affected by the number of decimal places.

For example: Suppose you had two data files, each containing an item in the format "##.##". One file has an implied decimal point, the other file physically contains the decimal point. Here are the two ways to define the item (using 12.75 as an example)

```
Length  Dec  Dec Char
4       2    No       (decimal implied; number would appear as "1275")
5       2    Yes      (decimal present; number would appear as "12.75")
```

NOTE: ID items cannot have decimals.

See also: Item Properties

Decimal Character

This item property applies to those numbers specified as decimal. If the number is a decimal value, this states whether or not the decimal point is present in the data file. Your valid choices are:

• **Yes** the data file contains a decimal point for this item, or

• **No** the data file does not contain a decimal point for this item.

Note that if your item does not have a "numeric" data type, the Data Dictionary will not allow any value other than **No.** You can set this option for all items by clicking on "DecChar Default 'Yes'" on the Option menu.

See also: Item Properties

Zero Fill

This item property controls whether numeric data item should be stored with leading zeros or blanks.

For example, during data entry a numeric item with a length of 3 is encountered. A value of "92" was keyed. How will this value be stored in the data file?

- If **Zero Fill** had been set to **Yes**, the value would appear as: "092"

- If **Zero Fill** has been set to **No**, the value would appear as: `" 92"`

You can set this option for all items by clicking on **ZeroFill Default 'Yes'** from the **Options** menu.

See also: Item Properties

Value Sets

Value Sets Description

Creating value sets in the dictionary controls the valid range (for numeric) or valid set of values (for numeric and/or alpha) that a dictionary item (or subitem) can have. Value sets play a key role in CSPro, in particular for numeric items, and should always be created unless there is a strong reason not to do so.

Some of the uses they serve include:

- During the design phase, as a tool of discussion with subject-matter specialists
- During data entry, to control what values can be entered
- During data entry, to present a list of valid responses from which the enumerator can choose
- Within batch edit applications, to enable usage of the impute function's **valueset** option
- Within tabulation applications, to facilitate table creation
- To generate code books, useful when exporting a datafile to other formats (e.g., R, SPSS, SAS, and Stata)
- As documentation to explain the characteristics of the data for future reference

As mentioned above, value sets are especially useful during data entry, in particular for numeric items. For example,

- A data item of field width 1 would allow values 0-9 to be entered—not likely useful for non-zero identification items
- A data item of field width 1 would allow values 0-9 to be entered—not likely useful for dichotomous responses such as Yes/No or True/False that expect 1 or 2
- A data item of field width 2 would allow values -9 to 99 to be entered—also not likely useful for continuous variables such as age that cannot be negative

Similarly, for alpha variables, and in particular for alpha variables of length 1 that are meant to capture A, B, C, etc. type of responses, failing to create a value set to list the specific letters allowed would end up allowing any character—blanks and punctuation characters included.

As can be seen in the first example below, each dictionary item can have more than one value set. In addition, if multiple languages are defined for the dictionary, there can be more than one value set for an item—though note this does not actually create distinct value sets, as they will share the same unique value set name. It only allows you to create value sets in different languages in order to switch between/among them.

Example 1

For example, suppose you have a survey that needs to classify people's ages three different ways: by discrete value, by 5-year cohorts, or by category, such as "Child," "Adult," etc. This is easily done by adding value sets for the AGE data item:

```
Value Set Label  Value Set Name  Value Label         From  To
Age              AGE
                                                     0     98
                                 Not Reported        99
Age by 5 years   AGE_5YRS
                                 0 to 4 years        0     4
                                 5 to 9 years        5     9
                                 10 to 14 years      10    14
                                 15 to 19 years      15    19
                                 20 to 24 years      20    24
                                 25 to 29 years      25    29
                                 30 to 34 years      30    34
                                 35 to 39 years      35    39
                                 40 to 44 years      40    44
                                 45 to 49 years      45    49
                                 50 to 55 years      50    54
                                 55 to 59 years      55    59
                                 60 years and over   60    98
Age by Category  AGE_CATEGORY
                                 Infant              0     0
                                 Child               1     12
                                 Teenager            13    19
                                 Adult               20    59
                                 Senior              60    98
```

The AGE item now has three defined value sets: AGE, AGE_5YRS, and AGE_CATEGORY. The first value set defines the acceptable range for data entry, while the second and third value sets give a breakdown as you might want to see the data tabulated.

Each new value set will always be added to the end of the item's value set listings. If you add to the wrong place, press the **Esc** key to stop the add. Use undo if you added at the wrong place. Likewise, you can also just cut (Ctrl+X) the value set and paste (Ctrl+V) it to the desired position. If a value set is not in the first position but you need it to be, you can make it the primary (first) value set by right-clicking on the Value Set Label/Value Set Name row and selecting the "Make Primary Value Set" option.

Example 2

Continuing with the above value set, suppose the AGE_5YRS ranges were already entered elsewhere such as in an Excel file. You could simply copy the labels and ranges and paste it directly into the value set screen, saving you the effort of retyping it.

Example 3

Suppose you needed to tabulate age by single years: 0 to 99. It would be very tedious to create that value set. Instead, you could use the Generate Value Set tool to create this numeric value set.

See also: Value Set Properties, Value Set Images, Value Description, Implications of Data Dictionary Value Sets, Multiple Language Dictionaries

Value Set Properties

You can view a value set's properties by selecting the item to which it belongs (via the dictionary tree tab). When creating a value set, the following must be set.

Property	Meaning
Value Set Label	A descriptive text label for a collection of categories of an item. This is used in tabulation applications in table titles.
Value Set Name	The name of this item for use in the CSPro language procedures.

See also: Value Sets Description, Value Set Images

Value Set Images

In addition to the response values for a field, you can associate an image with some or all of the values. In this way, an the interviewer can select a response based on the picture. This can be helpful, for example, in CAPI situations where an interview must select units of measure or other things that may best be described visually. This is also useful for asking sensitive questions of people who may be illiterate (e.g., "What political party do you favor?").

From	To	Special	Image	
1			potato.jpg	...
2			yam.jpg	...

To associate an image with a value, click on the "..." button at the end of the value row and then select the image. Images in five file formats are supported: .jpg, .gif, .bmp, .png, and .tif.

If you are deploying a data entry application to another computer or an mobile device, you must remember to distribute the images along with the CSEntry data entry application. If you want CSPro to automatically bundle the images with the data entry .pen file, you may want to put the images in a special folder and then denote that folder as a resource folder.

See also: SetValueSet Function

Values

Value Description

A value set contains one or more **values**. A value can represent a single response, or a range of responses, as shown below (note that the final two value columns, 'Special' and 'Image', are not depicted):

Example 1: Dichotomous Values

Value Set Label	Value Set Name	Value Label	From	To
SEX	P03_SEX			
		Male	1	
		Female	2	

Example 2: Continuous Values

Age allows responses 0-95, with no break in the sequence. There is no need to provide a label to the 0-94 range, as the meaning of the response is clear. Only with "95+", which means all persons age 95 years and older, do we need a label.

Value Set Label	Value Set Name	Value Label	From	To
AGE	P04_AGE			
			0	94
		95+	95	

Example 3: Categorial Values

Not all variables are continuous—many will have breaks in their responses. The classic example is Industry and Occupation, with a very small portion of Occupation shown below. Notice how "DB & Network Admins" spans two lines. To make these disparate values be considered a single value, press the space bar in the "Value Label" field for the second portion of the response (the 151244-151245 row). This will join the responses into a single value. You will know this is successful by the lack of a notes box at the beginning of the second value line.

Value Set Label	Value Set Name	Value Label	From	To
OCC	P10_OCC			
		DB & Network Admins	151241	
			151244	151245

New values will always be added to the end of the existing value set listings. If you add to the wrong place, press the Esc key to stop the add. Use undo if you added at the wrong place.

See also: Value Properties

Value Properties

Property	Meaning
Value Label	The descriptive text for a single value or range of values. This label is used when showing value sets or by the tabulation applications module when creating column headings and stubs.
From	This is the single value, or starting value of a range associated with the value label. To add multiple ranges to a value, enter one or more spaces as the value label on the next value(s), the values which follow become part of the previous value. Multiple ranges are indicated by the lack of a "notes" box at

	the beginning of the value line.
To	This value is the upper limit of the range of values being defined. It must always be greater than the "From" value on the same line. Where only a single value is associated with the "value label," the "to" value may be blank.
Special	A numeric data item can be assigned one of four special values in the data dictionary. These are: `missing`, `refused`, `notappl`, and `default`.
Image	An image that is displayed along with the value label when running a CAPI application.

See also: Value Description

Relations

Relation Description

Relations provide a way of linking one multiple record or item to one or more multiple records or items. For example, suppose a questionnaire contains two record types, child records and mother records. Each child record contains a data item that gives the sequence number of the mother record of the child's mother. A relation can be defined which links child records to mother records so that when data items from a child record are processes, the corresponding mother record data items are available for processing. Relations work much like database joins.

Relations have one primary multiple record or item. Each instance of the primary element in a case is processed one at a time. A relation has one or more secondary records or items. The corresponding secondary elements are linked to the primary element during processing.

There are four types of linking that can defined by relations.

Type	Description
Occurrence to Occurrence	Corresponding occurrences of the primary record or item and secondary record or item are linked, that is first occurrences are linked, second occurrences are linked and so on.
Item to Occurrence	The value of an item on the primary record is a pointer to the occurrence of the secondary record.
Occurrence to Item	The value of an item on the secondary record is a pointer to the occurrence of the primary record.
Item to Item	The value of an item on the primary record is compared to the value of an item on the secondary record. If the values are equal, the records are linked.

Relations can be used in `for` statements in batch programs and in the Export Data tool.

See also: Relation Properties, Relation Statement

Relation Properties

Property	Meaning
Relation Name	The name of this item for use in for statements in the CSPro language.
Primary	This is the name of multiply occurring record or item. Items in the secondary are linked to items in the primary.
Primary Link	This is either (occ) or the name of an item. If the primary is a record, then this is the name of an item within the primary record. If the primary is an item, then this is a the name of a subitem within the primary item.
Secondary	This is the name of multiply occurring record or item. Items within the secondary are linked to item in the primary. The secondary cannot be the same as the primary.
Secondary Link	This is either (occ) or the name of an item. If the secondary is a record, then this is the name of an item within the secondary record. If the secondary is an item, then this is a the name of a subitem within the secondary item.

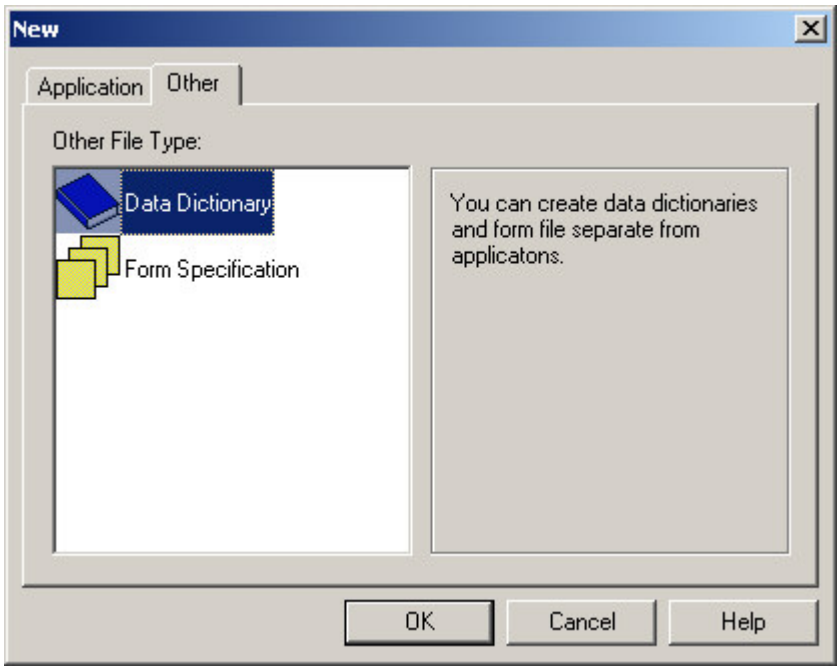
See also: Relation Description

Data Dictionary Application

Creating a Dictionary for a New File

To create a data dictionary application do the following:

• Click on the toolbar, or from the **File** menu, select **New**; or press **Ctrl+N**



• In the Object window select Data Dictionary

• Provide a name for the new dictionary (you need not provide the dictionary extension (.dcf); it will be automatically appended to the name).

• Select the folder where the dictionary (object) is to be stored. You can press the "**Browse**" button to locate an existing folder. If the dictionary file already exists, it will be used. If the dictionary file does not exist, it will be created.

• Press "**Next**" to advance to the Summary Screen and review your choices.

• If everything looks correct, press **Finish** to complete the operation.

If you are creating a dictionary to describe an existing data file, you may want to use absolute mode, in the event there are any "holes" in the data file(s). Or, if you want to use only a subset of the data file's information, using absolute positioning allows you to define only those data items of interest to you. If you are defining a dictionary for a new data file, you should be in relative mode, as this does not allow "holes" in your data.

If you have a long questionnaire, you can split the job and have several persons create data dictionaries for different sections. Later, you can copy and paste the items into one dictionary, making sure that the record type identification is unique.

Creating a Dictionary for an Existing File

To create a dictionary from an existing file you will need written documentation concerning the organization of the data on the file. This is usually presented as a set of record descriptions. These record descriptions identify the different kinds of

records, the items on each record, the starting position and length of each item, type of data contained in each item, the values that can appear in each item and the significance of those values. Further, if you only want to use a subset of information in the data file, using absolute positioning allows you to define only those data items of interest to you.

Once you have the record descriptions for the data file you are ready to create the dictionary.

- With CSPro create a new Data Dictionary file.

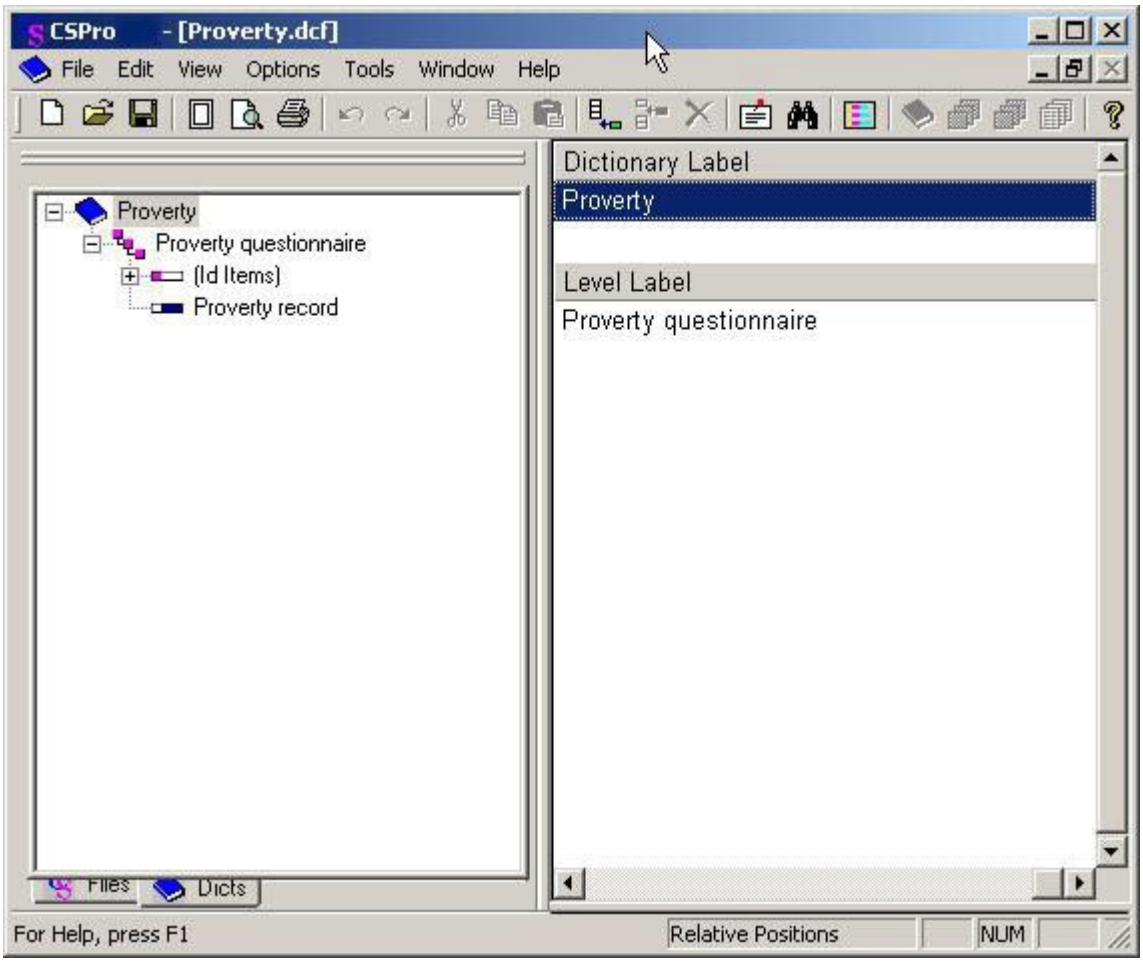
- In the Turn Options menu, uncheck the option for "Relative Positioning" so that you can position each data item according to the written specification.

- In the new data dictionary, define the records, items, and values from the record description.

See also: Select Relative or Absolute Positioning

Data Dictionary Screen Layout

When finished creating the data dictionary application, you will see the screen divided into two windows. The screen on the left displays the dictionary tree:



CSPro created the dictionary shown (Poverty) with one level (Poverty questionnaire), and that level contains a set of ID Items (Id Items) and one record (Poverty record).

The screen on the right displays detailed information for the highlighted object in the left-hand screen. For example, if in the left-hand screen the focus (cursor or highlight) is on the first line ( ), the right-hand screen will display information about the dictionary as a whole. If the focus in the left-hand screen is moved to the second line ( ) questionnaire level, the right-hand screen will display information about the questionnaire (case), which is the basic element of the file. As the focus is moved down the dictionary tree, the right-hand screen changes to reflect the different items of interest at each successive level.

Note the tabs at the bottom of the left-hand view marked "Files" and "Dicts". Clicking on either of these tabs will bring up

the appropriate tree. Within each tree, **Ctrl+T** toggles the view. For the File Tree, the two views will show either the internal or external names of the files; for the Dictionary Tree, the two views will show either the unique dictionary name or the dictionary label of each element.

The following is a good strategy for building your dictionary to reflect the data to be collected from your questionnaire:

- The first thing we suggest you do is to change the level properties to reflect your intended usage.
- Next, determine what ID items are necessary—remember that ID Items set will appear on each record in the current level, as well as all records in any lower levels.
- Third, create any additional records for the level, before changing the record properties as needed (e.g., setting the maximum number of occurrences for each record, stipulating whether the record is required or optional, and adding a record type, should you have 2+ records).
- Fourth, you're ready to add in dictionary items—these generally have a 1-to-1 correspondence to each question on your questionnaire.
- Finally, you're ready to make a final pass to add in value sets and values for each data item. This is a critical step, especially for numeric items, and should not be ignored. It determines the valid range of responses for each data item.

See also: Dictionary Property Panel

Dictionary Property Panel

The dictionary's property panel is launched from the **View** menu option **Property Panel**, or by pressing **Alt+Enter**. To access the property panel for specific dictionary elements, many can be viewed by simply selecting an element on the dictionary tree. However, for some elements you will need to select the "parent" level or record from the tree, and then select the "child" record or item on the workspace in order to see the associated properties.

General Properties

This property block applies to all dictionary elements. However, the Name and Alias properties are not applicable for dictionary values.

- **Type**—the underlying type of the dictionary element. Allowable types are dictionary, level, record, item, subitem, value set, and value. This property is not editable by the user.
- **Label**—the label given to the dictionary element. This property is not editable within the property panel, it must be changed within the workspace.
- **Name**—the unique name given to the dictionary element. This property is not editable within the property panel, it must be changed within the workspace. This property does not apply to dictionary values, as they are not given unique names.
- **Aliases**—allows one or more aliases to be created for the dictionary element. Aliases can be typed or pasted into the field. When adding more than one alias, separate each alias with a comma. If the name of an alias being added already exists elsewhere in the dictionary, it will not be added, as duplicate names are not allowed. A list of all aliases defined in the dictionary can be seen by going to the **View** menu and selecting **Aliases**. Aliases can also be created in logic using the `alias` statement. This property does not apply to dictionary values, as they do not have a unique name.
- **Note**—in addition to adding notes from the workspace, notes can be added within the property panel as well.

Occurrence Labels

This property only applies to repeating dictionary records or repeating items.

- **Occurrence Labels**—in addition to being able to set occurrence labels from the workspace, this property can be set within the property panel as well.

Item Properties

This property block only applies to dictionary items and subitems.

- **Default Capture Type**—this will set the default capture type associated with an item. When the item is dropped onto a form, it will use this capture type when set. It will also be used by Data Viewer to more intelligently display data, such as when checkboxes are chosen. During data entry, if a capture type is specified for a field, that setting will override the dictionary setting.

Numeric Item Properties

This property block only applies to numeric dictionary items and subitems.

- **Zero Fill**—in addition to being able to set the zero fill property from the workspace, this property can be set within the property panel as well.

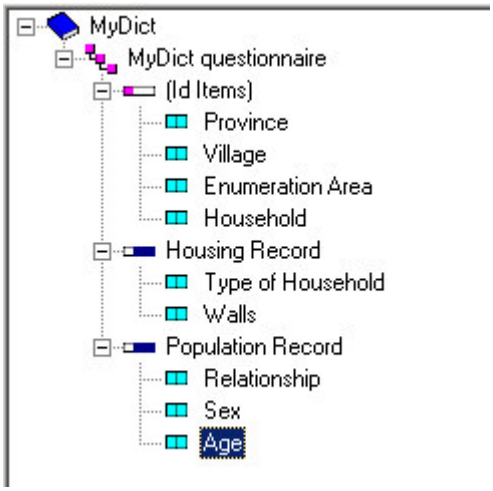
Value Properties

This property block only applies to dictionary values.

- **Image**—allows an image to be associated with a specific dictionary value, which will be displayed alongside the label during data entry. Provide the full path and name of the image file. Be sure to include the image file with the application when deploying to another device.
- **Text Color**—allows you to color your value labels. By clicking on the field, a small dialog will appear that displays basic colors (green, blue, red, etc.). You can select a color from the dialog, or type in its name. You can also click on **More Colors** to see a full palette of colors.

See also: Change Form Properties, Change Field Properties

Data Dictionary Tree



The Data Dictionary Tree contain the following items:

• **Data Dictionary File:**
This if the highest node, the identification of the data dictionary file.

• **Level:**
This is the second-tier tree node and identifies the level or questionnaire or case

• **Identification Item(s) or Record:** or
Represent the identification, or the record

• **Field:**

Represent the individual items. If the item has more than one value set, then the field will show the symbol "+" indicating that the item can be expanded. Value sets are always designated with the symbol . If the item contains sub-items, these will be designated with the symbol .

Relative and Absolute Mode

The mode refers primarily to the start positions of your data items. You can create a dictionary either in **relative** or **absolute** mode.

- **Relative mode**
  - Use relative mode when designing a new data file; you do not normally want "holes" (unused space between items) in your data files, as this will increase the size of the file.
  - The record type, if present, is always the first item in every record.
  - ID items, if any, are always located after the record type (and other ID Items defined at a higher level).
  - Each data item will be placed after **all** defined ID items (even those defined at a higher level than the record) in the record.
  - There are no gaps or "holes" between items.
  - As items are added, inserted, modified, or deleted, other items are automatically moved as needed to maintain the above arrangement.
  - Changing the starting position of an item will move it and any following items to give the implied relative arrangement.
- **Absolute mode**
  - Use absolute mode to create a dictionary from an existing data file.
  - The record type, ID items and data items can be positioned at any location in a record.
  - All items will remain in their assigned locations, unless specifically moved by the user.
  - When inserting or adding an item, there must be room (i.e., a "gap") for the item at the specified location.
  - When items are deleted, gaps may be created.
  - When an item's starting position or length is changed, room for the item must exist.

Dictionary Types

Every dictionary associated with an application has a type value that indicates how it is being used. For the primary dictionary (i.e., the one upon which your application was created), this will be your **main** dictionary. Other dictionaries (ones that are inserted either directly or secondarily via a forms file) can have additional properties, as explained below.

To see your dictionary's type, go to the Files tab, right-click on the dictionary in question, and select **Dictionary Type**. You will then see the following four choices (which may or may not be active, depending on their use):

Type	Description
Main	This is the principal dictionary upon which the application was built. You cannot give the dictionary another status as it will always be the primary dictionary for the application.
External	When you add a dictionary to an application, its type can either be "external" or "working." If it is an external dictionary, it must have an associated data file. When external dictionary variables are used in an application, their default values will be Not Applicable (Notappl).
Working	When you add a dictionary to an application, its type can either be "external" or "working." If it is a working dictionary, it does not need an associated data file. When working dictionary variables are used in an application, their default values will be blank (if the variable type is alphanumeric) or zero

(if the variable type is numeric).

Special Output This option is provided for backward compatibility with ISSA Batch Edit Applications. Only non-primary dictionaries used in Batch Edit Applications can have a "special output" type. Refer to the ISSA Manual for further instruction.

Reconciling Dictionary Changes

Whenever you make changes to a data dictionary, CSPro must reconcile the changes in applications that use the dictionary. If the application is open when you make the change, CSPro automatically makes the change in your application. If the application is not open, CSPro will attempt to make any changes the next time you open the application.

Under some circumstances CSPro will ask you to assist in the reconciliation process. You may asked whether you want to delete item from a form or rename the item, that is, use an item with a different dictionary name.

To rename the item, select "Rename" and then choose the new item name from the list presented. To delete the item, select "Delete."

How to ...

Open an Existing Dictionary Application

Click on the toolbar, from the **File** menu select **Open**, or press **Ctrl+O**. Select from the "Files of type" at the bottom of the dialog box. In CSPro you can open either an application or a file. Select the name of the file you want to open. Each time you open or create an application or file, it is added to the files tree. Any other files belonging to the application or file will also be opened and added to the appropriate trees.

You may open a data dictionary and make changes to it, even if it already belongs to an application. Be aware that if you later open an application to which it belongs, CSPro will automatically make necessary adjustments in other files. For example, if you delete or rename a dictionary item, then later open an application that contains the data dictionary, any corresponding fields on forms will be deleted.

You may open a forms file and make changes to it, even if it already belongs to an application. However, you will not have access to the associated Logic file and you will not be able to run it.

Any changes you make to applications and files are not made permanent until you save the file or application that you modified.

Move Around a Dictionary

Press To

Up Arrow Move up one line

Down Arrow Move down one line

Page Up Scroll up one screen (if possible)

Page Down Scroll down one screen (if possible)

Ctrl+Home Jump to first record, item, or value (from the view only)

Ctrl+End Jump to last record, item, or value (from the view only)

Ctrl+Left Arrow Scrolls left (if possible, and from the tree only)

Ctrl+Right Arrow Scrolls right (if possible, and from the tree only)

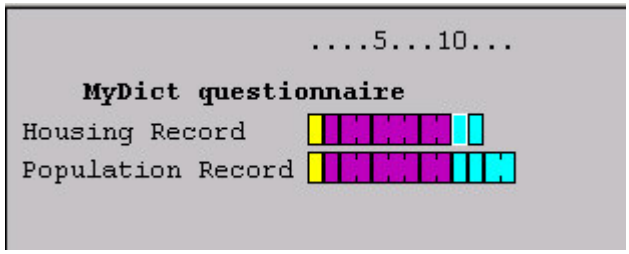
Ctrl+Up Arrow Multi-selects rows (from the view)
Scrolls up (if possible, and from the tree only)

Ctrl+Down Arrow Multi-selects rows (from the view)
Scrolls down (if possible, and from the tree only)

See also: Add, Modify, Delete Dictionary Elements. Toolbar Summary

View the Dictionary Layout

From the toolbar, press , or press **Ctrl+L**, or select Layout from the View menu. Any of these options will toggle the display of the current layout of the data dictionary. The display will appear in the lower half of the right-hand screen.



The layout shows you where, physically, each item in each record is located, how much space has been allocated to it, and if there are any gaps in your file (which is possible when the file's status is absolute). The color scheme used is the following:

denotes the Record Type

denotes Id Items

denotes record Items

denotes Subitems

• Click on the item on the layout window to move the cursor to the specific item in the dictionary window

• Single click on an item to move to the item's definition.

• Double click on an item to show its value set(s).

• Press **Ctrl+L** a second time to close the view.

Add Dictionary Elements

If you wish to add a level, record, item, value set, or value to a dictionary, first position the cursor in the location where you want to add the dictionary element, then click on the toolbar; or press **Ctrl+A**; or, from the **Edit** menu, select the **Add [Element]** option. You can add from either the tree or view. In either case, the dictionary's menu bar and popup menu listing (displayed if you right-click over a tree item or the view) are context-sensitive. If you add to the wrong place, press the **Esc** key to stop adding.

See also: Modify Dictionary Elements

Modify Dictionary Elements

You can modify any of the dictionary's items (i.e., a level, record, item, value set, or value). You can modify an item from either the tree or view. In either case, the dictionary's menu bar and popup menu listing (displayed if you right-click over a tree item or the view) are context-sensitive. Therefore, depending on what you've selected, your choice will be to modify the properties of a level, record, item, value set or value. The **Ctrl+M** key combination invokes the modify process; or, from the **Edit** menu, choose the **Modify [Element]** option.

See also: Add Dictionary Elements

Add or Modify Levels

Most of the applications will only use one level. If you need to add additional levels (recommended only for more complex censuses and surveys), do the following:

• From the dictionary tree, select any level within the dictionary

• Right-click to get the popup menu and select "Add Level", or press **Ctrl+A;** or select "Modify Level", or press **Ctrl+M**.

• Complete the level properties requested

• When you are finished entering the level(s) desired and wish to terminate data entry, press the **Esc** key.

After entering the second level, the 'add level' mode will continue for one additional level. To terminate with just two levels, press **Esc** when you reach the (third) new level entry. Additional levels will have exactly the same structure as the first one, i.e., an (ID Items) set and a record ('New Record'). You can undo modifications if you decide they were incorrect.

If you need additional records for this level, you should create them first. After selecting a level in the tree, you can press to initiate add mode. The level will always be added at the end of the dictionary.

See also: Dictionary Hierarchy, Level Description, Level Properties

Add or Modify Records

• From the dictionary tree, select any record (or (Id Items) set) within the level you wish to add or modify.

• Right-click to get the popup menu and select "Add Record", or press **Ctrl+A**; or select "Modify Record", or press **Ctrl+M**.

• Complete the record properties requested.

• When you are finished entering the record(s) desired and wish to stop adding, press the **Esc** key.

If this structure is sufficient for your needs you can begin adding identification items and data items to the record. The item will always be added at the end of the record.

There is no limit on the number of records within a level.

See also: Dictionary Hierarchy, Record Description, Record Properties, Record Type, Required Record, Maximum Number

Add or Modify Items

From the dictionary tree:

- Click on the ID item or record item you wish to add or modify.
- Press **Ctrl+A** to add or **Ctr+M** to modify the item (alternatively, right-click on the item from the tree and make your choice from the popup menu).
- Add/modify the item properties requested (when adding)/desired (if modifying).
- When you are finished entering data items and wish to stop adding, press the **Esc** key.

There is no limit on the number of items you can add within a record. The item will always be added at the end of the record. After selecting an item in the tree, you can press Add Button to initiate add mode.

Press the **Esc** key to stop adding. You can also use undo if you added at the wrong place.

Add or Modify Value Sets

From the dictionary tree:

- Select the item or subitem to which you wish to add a value set to.
- Right-click to get the popup menu and select "Add Value Set", or press **Ctrl+A**; or select "Modify Value Set", or

press **Ctrl+M**.

- Provide the Label and Name for the Value Set.
- Complete the value set properties requested.
- When you are finished entering values and wish to stop adding, press the **<Esc>** key.

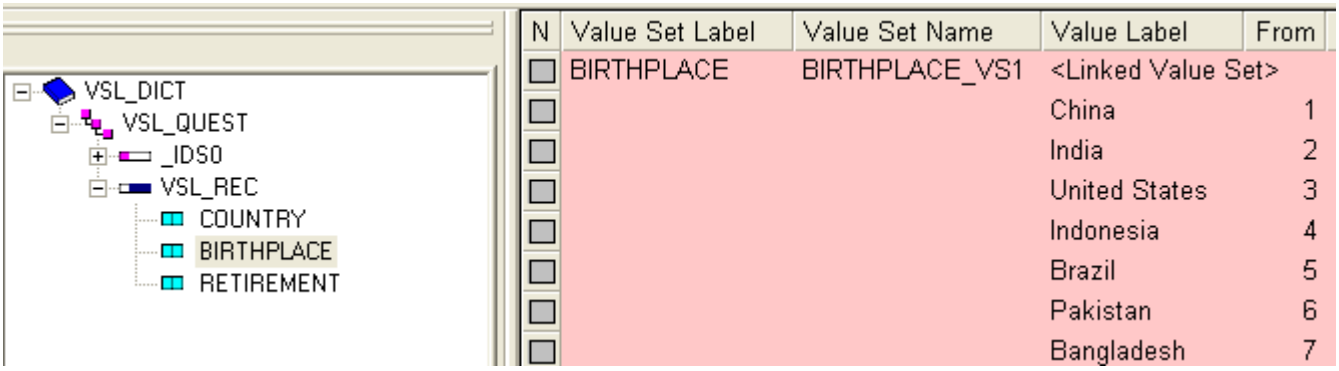
Linked Value Sets

There are times when several items in a dictionary share the same list of possible values. For example in a survey that asks for the country where someone lives as well as the country where that person was born, the value set for both items would be a listing of all countries. A user can enter the possible values for one item and then copy and paste them onto the second item, but the concept of linked value sets is a better approach to the problem. With linked value sets a user pastes a "value set link" which indicates to CSPro that the two items share the same possible values. Then if the user adds or modifies entries for one item, the values for the linked items are also modified. Users do not have to update multiple value sets to make changes.

Create a Value Set Link

Create the value set for one of the items and then copy the value set to the clipboard. Go to the value set window for the item to which you want to link the value set. Right-click on the window and select **Paste Value Set Link**.

After pasting the link the value set will turn pink as a way of indicating that the value set is shared across two or more items. Modify or add values to the value set as you would for a regular value set.



Remove a Value Set Link

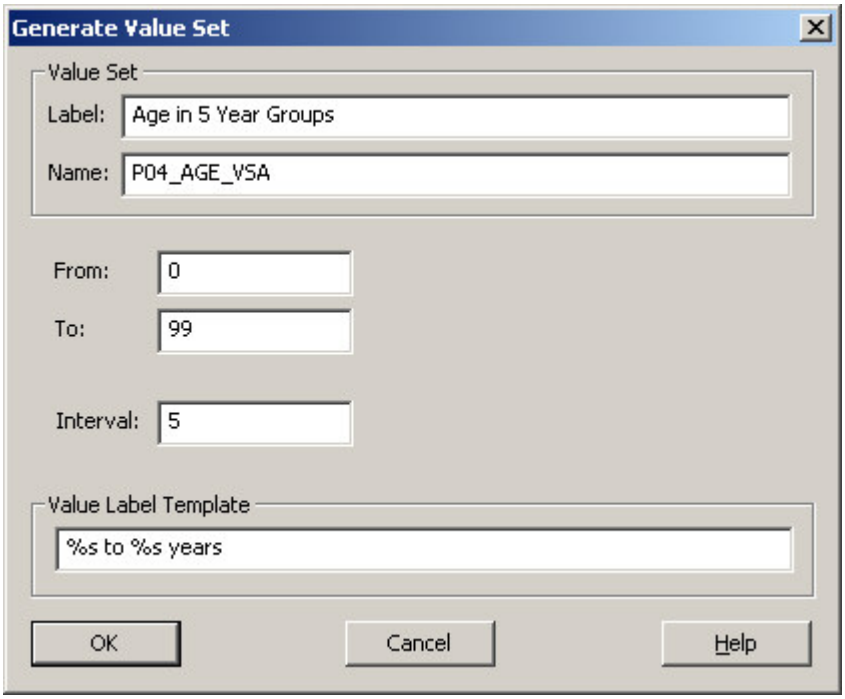
Right-click on the value set name and select **Remove Value Set Link**. If the value set is linked to more than two items, you can chose to remove the link for only the item that you right-clicked on, or you can remove all the linkages. When a link is removed the value set is no longer colored pink. The values remain but now modifying the value set does not affect

the value sets in the items to which it had previously been linked.

Generate Numeric Value Set

To generate a value set for a numeric data item:

- From the dictionary tree, select the desired item.
- Right click on the view on the right, or select the **Edit** menu.
- Select **Generate Value Set**. The following dialog box will appear.



Value Set Label: A descriptive text label for a collection of categories of an item. Used by the Tabulation module to select tabulation categories. This cannot be left blank.

Value Set Name: The name of this value for use in the CSPro language procedures.

From: The smallest value in the value set.
To: The largest value in the value set.
Interval: The size of the interval in each value range generated. For example, if 5 year age groups are desired, then the interval is 5. The interval must be greater than 0. If the interval is the smallest positive number that can be contained in the data item, then single values are produced.

Value Label Template: A model for descriptive text of each value or value range that is generated. The characters **%s** in the template are replaced by the **from** and **to** values for each value range. All other characters are output as entered.

Add or Modify Values

From the dictionary tree;

- Select the desired value set
- Select one of the value set's values in the view on the right
- Press **Ctrl+A** to begin adding a value
- Complete the value properties requested

- When you are finished entering values and wish to stop adding, press the **<Esc>** key

A single value set can contain one or more values. A value will always be added to the end of the current value set. If you wish to insert a value into an existing value set, use the insert feature. If you add to the wrong place, press the **Esc** key to stop the add. Use undo if you added at the wrong place.

See also: Value Description, Value Properties, Value Sets Description

Undo and Redo Changes

CSPro keeps track (in an "undo stack") of the 12 most recent changes you have made to your application. Please be aware that not all changes can be undone.

If you have made a mistake and want to undo it, press on the toolbar; or from the **Edit** menu, select **Undo**; or press **Ctrl+Z**. CSPro will try to restore your forms to the state previous to last change you made. To undo the next-to-last change, press the **Undo** button again.

Sometimes you may undo several changes and realize you have gone too far back. Press on the toolbar; or from the **Edit** menu, select **Redo**; or press **Ctrl+Y**. Redo is an "undo" of an undo.

Select Several Dictionary Elements

You can select several dictionary elements of the same type from the dictionary window.

Using the Mouse

1. Click on the line where you want to start selecting.
2. Hold the left mouse button down and drag the mouse up or down until your desired selection is highlighted. Note that the window will automatically scroll if necessary.
3. Release the mouse button.

Using the Keyboard

1. Using the cursor keys, move to the start of your desired selection, so that the blue highlight bar is on that line.
2. Press and hold the **Shift** key.
3. Use the **Up** and **Down** arrows to expand your selection. **PgUp** and **PgDn** will expand the selected lines a page at a time.

You can use cut and copy to move/copy your selection elsewhere within the dictionary, or to use in another open dictionary. You can delete multiple records, items, or values at the same time. Undo can be a useful feature when dealing with block operations.

Insert Dictionary Elements

If you wish to insert a level, record, item, value set, or value to a dictionary, first position the cursor in the location where you want to insert the dictionary element, then do one of the following: click on the toolbar; or press the **Insert (Ins)** key; or, from the **Edit** menu, select the "Insert [element]" option. You can insert from either the tree or view — in either case, the dictionary's menu bar and popup menu listing (displayed if you right-click over a tree item or the view) are context-sensitive. If you insert to the wrong place, press the **Esc** key to stop inserting. Use undo if you completed the insert to the wrong place.

Delete Dictionary Elements

If you wish to delete a level, record, item, value set, or value from the dictionary, first position the cursor in the location where you want to delete the dictionary element, then do one of the following: click ✕ on the toolbar; or press **Delete/Del**, or select **Delete** from the **Edit** menu.

If you delete the wrong object, click on the toolbar to undo the operation and recover the deleted material. You can select multiple lines by dragging the mouse over the desired lines or pressing down on the **Shift** key while you use the up or down arrow to adjust the selection.

Move Dictionary Elements

To move things in the Dictionary around use "cut," "copy," and "paste." "Cut" will delete the material from the dictionary and place it on the clipboard. "Copy" will simply place a copy of the selected material on the clipboard. "Paste" will place a copy of the material on the clipboard into the dictionary. You can paste cut or copied material to more than one location. You can also copy/cut dictionary elements from one dictionary and paste into another.

- **To cut things …**

  Select the material you want to cut, then click on the toolbar; or from the **Edit** menu, select **Cut**; or press **Ctrl+X**.

- **To copy things …**

  Select the material you want to copy, then click on the toolbar; or from the **Edit** menu, select **Copy**; or press **Ctrl+C**.

- **To paste things …**

  Select the place where you want the records, items, or values to be pasted then click on the toolbar; or from the **Edit** menu, select **Paste**; or press **Ctrl+V**.

Find Dictionary Elements

Press on the toolbar, or press **Ctrl+F**; or select **Find** from the **Edit** menu to open the Find dialog box. Names and labels of all dictionary entities (for example, levels, record, items, value sets) will be searched. The following options allow you to search for text:

Find What

Enter all or part of the text string to search for. Text used in previous searches is available by clicking on the down arrow and selecting from the dropdown list.

Next

Find the next occurrence of the text string, starting from the last one found. If it finds the item, it will be brought into focus in the view; otherwise you will receive a notification that it could not be found.

Prev

Find the previous text string starting from the last one found.

Match Case

If this option is checked, the string will match only if the letters are the same case (upper or lower) as in the string you entered as the search key. If this option is not checked, the search will ignore case.

Close

Close the Find dialog box.

Document Dictionary Elements

In the view screen on the right, select the element you want to document or for which you want to modify the documentation. Click on the toolbar; or from the **Edit** menu select **Notes**; or press **Ctrl+D**. You can also press the button at the beginning of the line of the selected element, or right-click and select Notes. You can use the **Enter** key to end a paragraph and begin a new one within the note. You can use **Ctrl+X**, **Ctrl+C**, and **Ctrl+V** to cut, copy, and paste text in the note.

Convert Items to Subitems

Select the items you want to convert to subitems. From the **Edit** menu, select **Convert to Subitems** or right-click on the item list in the view and select **Convert to Subitems** from the popup menu. Enter information about the item that will include these subitem(s).

To convert subitems back to items, delete the item. When asked if you wish to "Delete subitems too?" answer **No**.

Select Relative or Absolute Positioning

To toggle between Relative and Absolute positioning, select **Options** from the menu bar, then select **Relative Positions**. A checkmark indicates your file is in **Relative Positioning**; the absence of a checkmark indicates the file is using **Absolute Positioning**.

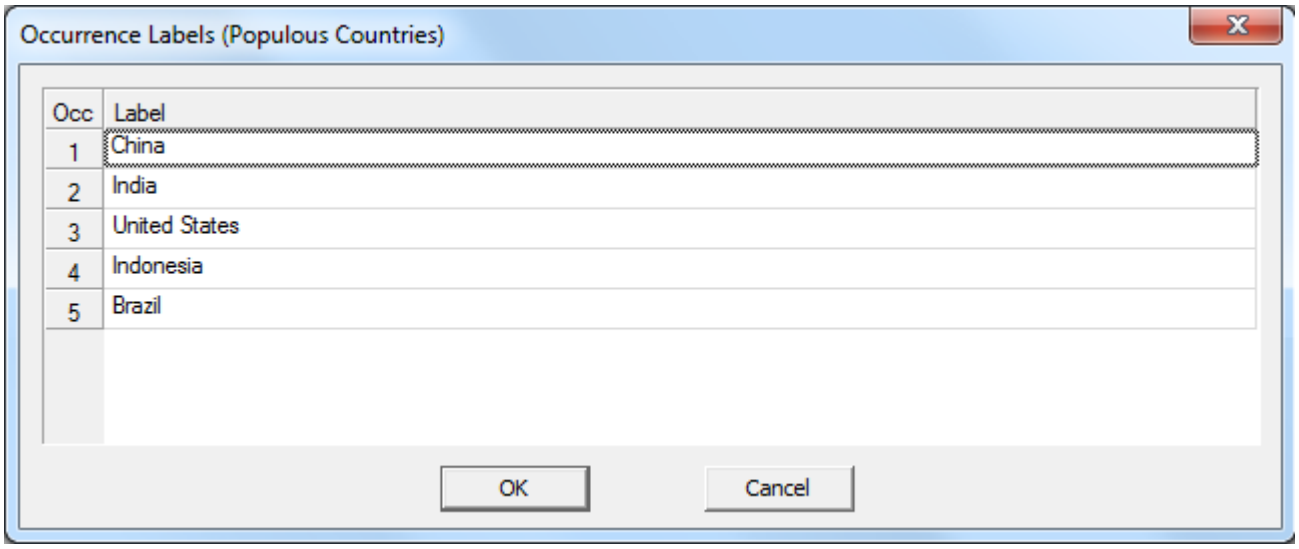
Create Dictionary with No Record Types

If your dictionary has only one type of record, there is no need for a Record Type item in the record. In order to remove the Record Type item, set its starting position to 0 and it length to 0.

N	Item Label	Item Name	Start	Len
	(record type)		0	0
☐	Province	PROVINCE	1	2
☐	District	DISTRICT	3	2
☐	Village	VILLAGE	5	3
☐	Enumeration area	EA	8	3

Occurrence Labels

Occurrence labels can be attached to a repeating item or a repeating roster by right-clicking on the dictionary element and selecting **Occurrence Labels**. Alternatively, that option can be found on the **Edit** menu. To edit an occurrence label, simply click on the appropriate row and add the label.



Occurrence labels can be used as row labels in a roster, in programmed logic, or as replacement text in the CAPI text window.

Add or Modify Relations

- From the **Edit** menu, select **Relations**.
- Press the **Add** button to add a relation at the end of the list or press the **Insert** button to insert a relation at the current highlighted line.
- Enter the Relation Properties across the line.
- Press the **Delete** button to delete the highlighted line.
- Press OK button when you have completed your edits.

Print the Dictionary File

Click on the toolbar; or from the **File** menu, select **Print**; or press **Ctrl+P** to open the Print dialog box and select among the following options:

• **Where**
You can send your output directly to a printer, or save it in a file.

• **Detail**
You can select **Complete** to print the data dictionary with the value sets and notes; or **Brief** to obtain a listing of the items only.

• **Order**
This option allows the user to place either the item name or the item label in the first column of the listing.

You can also print part of a document by selecting the desired text and then pressing **Ctrl+P**. To preview the printing click on the tool bar; or from the **File** menu, select **Print Preview**.

Save Dictionary as New File

To save a dictionary under a new name:

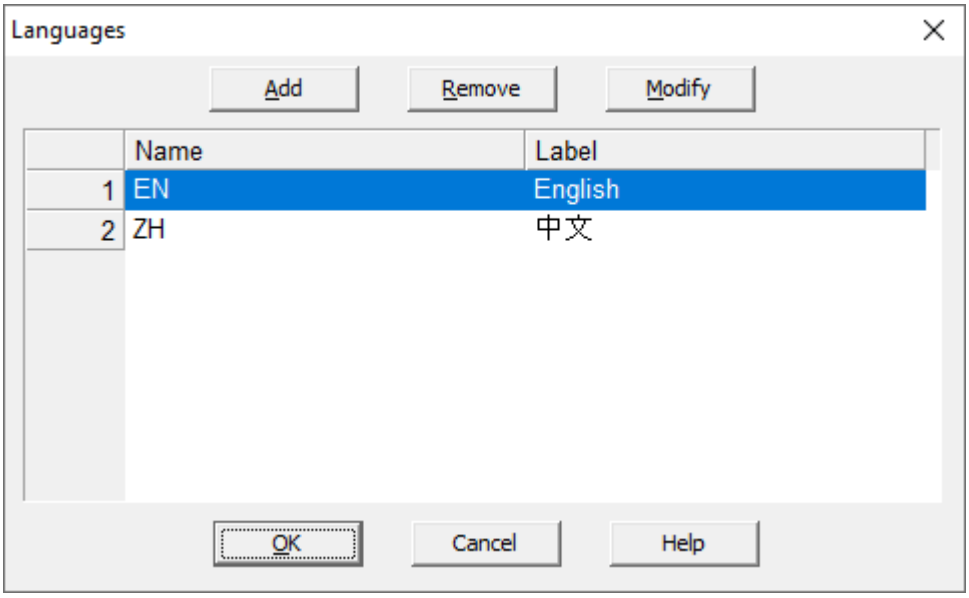
1. Open the dictionary alone (that is, not as part of an application).
2. From the **File** menu, select **Save As**.
3. Enter the file name for the new dictionary in the file open dialog box.

When the **Save As** is complete you will be editing the new dictionary.

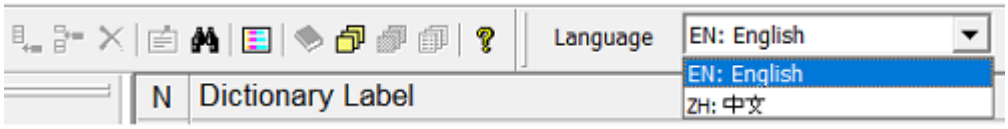
See also: Save an Application with a New Name

Multiple Language Dictionaries

You can create dictionary labels in more than one language. To do this, while editing your dictionary, go to the **Edit** menu and select **Languages**. You will be presented with the following dialog box, from which you can add (or remove or modify) languages. (This is similar to the way that you define CAPI languages.)



Once your application has more than one language, the dictionary toolbar will be updated to display the names of all defined languages:



Each language will have its own set of labels for all of the dictionary elements (record labels, item labels, value labels, occurrence labels, etc.). If a label is not defined for a given language, the label will default to the label for the first language. To simplify the editing of labels, you can press the keyboard shortcut **Ctrl + Alt + L** to switch between the languages.

When adding items to your data entry forms, you can specify that the field labels should be directly linked to the dictionary label. This default setting ensures that the field labels change when the user changes the language during data entry.

For your application to fully take advantage of multiple language dictionaries, you must define the languages that you would like to use in the CAPI language section. When running your data entry program, CSEntry will only list languages defined in the CAPI section. In this way, changing a language will change both the CAPI language text as well as the dictionary labels used.

Dictionary Macros

There are several specialized tasks that you can perform on a dictionary using the dictionary macros functionality. To access the options, either right-click on a dictionary in the tree and select **Dictionary Macros** or, with the dictionary editor active, select **Dictionary Macros** from the **Edit** menu. A dialog box will appear with the following options:

Dictionary Names and Labels

You can **Copy All** the names and labels to the clipboard for all attributes in your dictionary (levels, records, items, and value sets). If your dictionary has multiple languages, the **All Languages** option allows you to copy all of the labels, instead of just the primary label, to the clipboard. The **Item Lengths** option allows you to copy, in addition to the names and labels, the length of each item to the clipboard.

This can be useful if you want to edit the names, labels, and lengths in a different software tool, such as Microsoft Excel. The information is copied in a tab-delimited format.

If you edit the data in an outside program and want to bring the changes back to CSPro, you can use the **Paste All** functionality. CSPro will check that what is in the clipboard matches the contents of the dictionary, so you cannot add or delete items outside of CSPro.

You may find it useful to setup your dictionary parameters in CSPro and then have a subject matter specialist or clerk enter the labels in another software, particularly if working in multiple languages.

Value Sets

You can **Copy All** the names and labels to the clipboard for all value sets in your dictionary. As with dictionary names and labels, you can use this feature to edit the value sets in a different software tool. If your dictionary has multiple languages, the **All Languages** option allows you to copy all of the labels, instead of just the primary label, to the clipboard. If you select **Value Set Images**, the file names of value set images will also be copied.

After modifying the value set labels in an outside program, you can use the **Paste** functionality to bring the changes back to CSPro. You do not need to paste all value sets, so you can make changes to only the value sets that you want to modify. You can add and remove value set labels in the outside program, but you cannot change value set names.

If you want to remove all the value sets from your program, which can be useful if testing a data entry application that someone else designed (and for which you do not know the proper values), you can use the **Delete All** functionality. The change is permanent, however, so you may want to make a copy of your dictionary prior to removing all value sets.

Add Items to Records

This macro allows you to bulk add items to a record. This can be useful if you want to add many items to your dictionary, whose names and labels you will modify in a different software tool. Specify the **Number of Items** to add, and then select to which **Record** you want to add the items. The items will come with names such as `RECORD_NAME_ITEM001`, `RECORD_NAME_ITEM002`, etc.

Records

In a dictionary with many records, it may be useful to modify all records to make all required (**Set All to Required**) or to make none required (**Set All to Not Required**).

Generate Random Data File

Sometimes, particularly when testing a data entry or batch edit application, you may want to create a data file with some **Number of Cases** in it. If you do not have any preexisting data for your dictionary, this is a way to easily generate data without having to enter it manually. The output file will consist of cases with data based on your dictionary's specifications. For items with value sets, only valid values will be output.

If you want some number of blank values in your output file, you can modify the **Percent Not Applicable**. If you want some items to have values that are not in their value sets, than you can modify the **Percent Invalid**. When testing batch edit applications, this can be useful as a way to detect if your edits properly handle blank and out-of-range values.

Sample Data File

This macro provides an easy way to create a sample from an existing data file. For example, it is a common task to create a 1%, 5%, or 10% sample census data file, particularly when testing batch edit applications. Based on the provided **Percent**, you can create a **Random** or **Sequential** sample data file.

Suppose your file has 100 cases and you want to create a 10% sample. Creating a random sample means that all 100 cases have a 10% change of being selected for the sample. Based on randomness, that means that generally your sample file will have 10 output cases, but it could have fewer or more. You might also get some clumping of cases. On the other hand, a sequential sample means that the output file will consist of the 1st, 11th, 21st, ... cases. The output file will always have 10 cases. By modifying the **Start Position**, you can control which case is output first. For example, if that value was 3, then your output file would consist of the 3rd, 13th, 23rd, ... cases.

Data File Tasks

There are three options in this area:

- **Compact Data**: When working with text data sources as external dictionaries to a program, you can end up with a lot of deleted records. Deleted records appear in the file with a tilde (~) at the beginning of the line. This option will remove all deleted records from a file, reducing its size.
- **Sort Data in ID Order**: This will run the Sort Data tool to automatically sort a data file in ID order. Particularly after concatenating multiple data files, you may want to sort the data in some order, generally based on geographic ID items.
- **Create Notes Dictionary**: This will create a new dictionary, based on your current one, that can be used to read the .csnot notes file generated by a data entry application. If you want to export your data entry notes, you can use this macro and then open the newly created dictionary in the Export Data tool.

See also: Dictionary Analysis, Question Text Macros

Dictionary Analysis

While editing or viewing a data dictionary, you can generate several reports that analyze the contents of the dictionary by selecting an option from the **Dictionary Analysis** submenu off the **View** menu. The options include:

Items Without Value Sets: Determine which items have no value sets.

Numeric Items Without Value Sets: Determine which numeric items have no value sets.

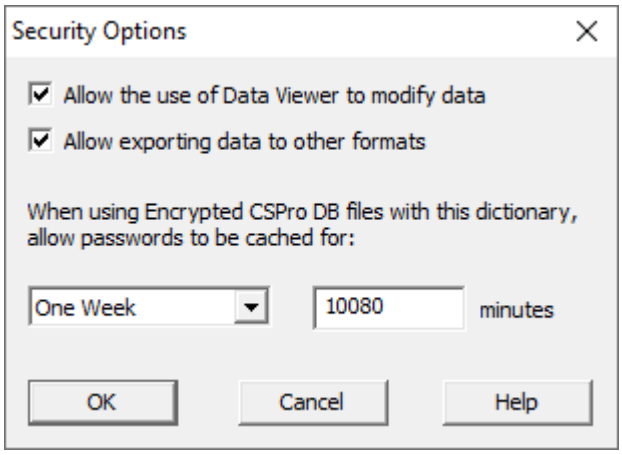
Numeric Items With Overlapping Value Sets: Determine which numeric value sets have discrete or range values that overlap.

Numeric Items With Mismatched ZeroFill or DecChar Options: Determine which numeric items have a zero fill or decimal character option that differs from the dictionary's default options. You generally want every item in the dictionary to use the same options.

See also: Dictionary Macros

Dictionary Security Options

Several security options can be defined on a per-dictionary basis. To change these options, select **Security Options** from the **Edit** menu. A dialog box will appear with the following options:



When checked, the first option, **Allow the use of Data Viewer to modify data**, means that a user can use Data Viewer to delete and modify cases.

When checked, the second option, **Allow exporting data to other formats**, allows a user to use Data Viewer, the Export Data tool, or the `export` statement to output data to another format.

The drop-down allows you to choose a time period to cache the password. Available choices are Never, One Hour, One Day, One Week, One Month, One Year, Forever, and Custom. If One Hour is chosen, 60 minutes will be prefilled; if One Day is chosen, 1440 minutes will be prefilled. If you enter 0 minutes, Never will be selected. If you enter a non-zero time period that does not correspond to one of the preset categories, the Custom option will be used. The Forever option must be manually chosen.

Encryption Options

Encryption options control whether or not passwords can be stored for data sources using that dictionary. These options only apply when using the dictionary to access encrypted data using the Encrypted CSPro DB data source. By default, passwords used to open such files are not cached on the machine, meaning that the password must be specified every time the file is opened (unless the password is specified in a connection string). If you want to allow the storage of passwords, specify the number of minutes that the stored password can be used to open the data file.

The password itself will not be stored on the device; instead, a key that can be used to open the data file will be stored. A clever hacker who accesses this key may be able to open the data but will have a hard time reverse engineering the key to get the original password itself. The key is stored in secure storage on the device. You can use the CSPro Settings dialog to clear any cached passwords.

See also: Encrypted Data

The CSPro Language

Introduction to CSPro Language

The CSPro language lets you write programming logic for your data entry and batch edit applications. In data entry applications you can write logic to control and check the keying or CAPI operation as it progresses. In batch edit applications you can write logic to identify and correct errors after data capture is complete.

This section contains the following information:

- Program Structure
- Declaration Section
- Procedural Sections
- Logic
- Language Elements

For a list of commands, see the CSPro Statements and Functions section.

Data Requirements

Data files appear in many different formats and structures. CSPro can work with several type data sources, but most people work with text files. If you are using data files created by another software package, you must save the data in a separate text file before you can use it with CSPro. Data files are limited to 2 gigabytes in overall size; the maximum length of any record in the file is 32,000 characters. CSPro encodes data files using UTF-8. Read more about Unicode text files at the Unicode Primer.

CSPro processes one case at a time. Each record must contain a unique questionnaire identification code in the same position in each record. This number must be the same for all records of the same case. If the file is a "flat" file, meaning that each questionnaire contains only one record, the questionnaire identification becomes irrelevant. In this case, any data item can be used as the questionnaire identification, but it should be unique for each record. CSPro uses the case identification values to determine where one case ends, and the next one begins. Records belonging to the same case must be contiguous within the data file, but there is no requirement that the data file be sorted by case identifier.

CSPro can handle a data file with multiple record types—for example, housing and population—but a record type code must identify the type of record. This code must be in the same position in each record. Within the same record type, each data field must be in the same position.

CSPro can process one input data file at a time, but it can access one or more external files. These files must also be described by a data dictionary.

In some survey data, especially where the total number of data items is great but only a few responses are expected, the user may choose a format in which each data field is preceded by a "source code" relating it back to the original document. By using this scheme, non-response fields (empty responses) need not be entered. With this type of format, each data field is not in a pre-defined location on the record. Before a file like this can be processed by CSPro, it must be reformatted so that the data fields are in fixed positions. Items in data files must be fixed format, that is, items must have the same starting position and length in every record where they occur.

See also: Data File Type Structure

CSPro Program Structure

CSPro logic consists of a collection of events defined as procedures. Each procedure performs the operations you specify using CSPro statements and functions written in the CSPro language. A CSPro program includes a declaration section and one or more procedural sections.

Declaration Section (PROC GLOBAL)

Declarations and definitions are defined in the **global procedure**. In this section you declare global variables, arrays, file handlers, and user-defined functions. The global procedure always appears at the beginning of the logic file and begins with the line "PROC GLOBAL." Except for within user-defined functions, there are no executable statements in this section. You can edit the PROC GLOBAL section by clicking on the topmost entry of the data entry edits tree or batch edits tree.

Example

```
PROC GLOBAL

// variables
numeric MinAgeDifferenceParent = 12,
        MaxAgeDifferenceMother = 55;

string personName;
array validRelationships(5);

// user-defined function
function numeric IsMotherValidByAge(numeric motherAge, numeric childAge)

IsMotherValidByAge = ( ( motherAge - childAge ) in MinAgeDifferenceParent:MaxAgeDifferenceMother
 );
end;
```

Procedural Section

This section contains executable and assignment statements that can be written before (preproc) or after (postproc) an event. Events always fall under the PROC section, which is followed by the name of the forms file, level, form, roster, or field. Statements are assumed to be in the postproc unless it is explicitly stated that they are in another procedure.

Data entry applications also have a forms file procedure. The form file preproc is executed before a data entry session begins. There are three other procedure types that may be useful in data entry applications: onfocus, killfocus, and onoccchange.

Example

```
PROC MYDICT_FF // form file procedure

preproc

    // statements

PROC MYDICT_QUEST // level procedure

preproc

    // statements

postproc

    // statements

PROC HOUSING_FORM // form procedure

postproc

    // statements

PROC INCOME // field procedure

    // statements (implicitly in the postproc)
```

See also: Proc Statement, PreProc Statement, OnFocus Statement, OnOccChange Statement, KillFocus Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

Programming Standards

It is a good idea to create a set of programming standards when writing CSPro code. If you work in an organization, you will want to discuss these standards with the other programmers in your organization. Code is more readable when multiple people use the same coding guidelines.

This is an example of a set of programming standards adopted by an organization. Your standards may differ, but the point is to at least have a set of standards.

1. Backup your code, and all components of the application, frequently. Create a system of version control (such as by using Git).
2. Compile code frequently while working on it as this will help you identify syntax errors early.
3. Use the help system (by pressing F1) and the reference window while writing logic.
4. Use `//` for line comments and `{ }` for multiline comments. Comment the code while writing it, not afterwards, as it will probably never get done in that case. Remember that it is likely that you will not be the only person who looks at your code.
5. Maintain indentations for code within conditional statements (`if`), loops (`do`/`for`), and functions.
6. Write CSPro statements in lowercase.
7. Write variable names from the dictionary in UPPERCASE.
8. Write globally declared variable names in mixed case, starting with a lowercase word (this is called camelCase). Make all global variables at least three characters long.
9. Declare all constants in `PROC GLOBAL`, not in the application-level `preproc`.
10. Use locally declared variables for loop counters and when only one procedure or function uses the value of the variable. Remember that the value of the local variable will be reset (to 0) each time the procedure is executed.
11. Use functions when an operation is generalizable or if the same operation will be called repeatedly from different procedures.
12. Use standard prefixes for variable names. For example:

- `cnt`: counts of an item (such as `cntPersons` and `cntHouseholds`)
- `ptr`: pointer to a record (for instance `ptrEldest` is the occurrence number of the eldest resident)

13. For batch applications, maintain an error message numbering scheme so that both subject matter specialists and programmers can understand where error messages occur. For example:
    - (1 digit) record type number
    - (2 digits) question number of the primary variable
    - (2 digits) error number for this variable
14. Do not use deprecated features as they may not be supported in a future version of CSPro.
15. Take advantage of free support from the U.S. Census Bureau at: cspro@lists.census.gov

See also: Define Coding Standards

Code Folding

The CSPro logic editor supports code folding, a way to collapse and uncollapse sections of code. When working with large amounts of code, it may be useful to collapse ("fold") code to see an overview of the code, and then to uncollapse ("unfold") the code to see details about an implementation.

To modify code folding options, from the **View** menu, select **Code Folding**. The **Level** options allow you to control what is folded:

- **None**: Turn off code folding.
- **Procedures and Functions**: Fold procedures and user-defined functions.
- **All**: In addition to procedures and functions, fold other block-like aspects of the language (e.g., `do`, `if`, `forcase`, `recode`, etc.).

The actions, all of which have shortcuts, are:

Action	Shortcut	Description
Fold All	**Alt + 0** (zero)	Folds the entire document, collapsing all sections at the current fold level.
Unfold All	**Alt + Shift + 0** (zero)	Unfolds the entire document, uncollapsing all sections at the current fold level.
Toggle Current Level	**Ctrl + Alt + F**	Toggles the current code fold level where the cursor is. That is, if the section is folded, it will be unfolded; if it is unfolded, it will be folded.

Debugging CSPro Applications

There is no structured debugger that works with CSPro. However, there are several techniques that can be used to find problems and solve them. One technique is to "comment out" code. By enclosing specific code between braces `{ }`, you can isolate specific areas of code that may be the cause of your problem.

Another technique is to use error messages with the `errmsg` function. The generation of an error message tells you that program control has passed to that point in your application. The error message can also include variable values to indicate the status of those variables at that point.

It may sound obvious, but it is good programming technique to **indent** your code. This will help you find problems caused by unterminated `if` or `do` statements, for example. If you consistently indent (or tab) the content within any control structure, finding a lost `endif` or `enddo` will be much easier.

You can also use the `trace` function to determine problems in your application by outputting messages to a separate window or file rather than have them appear onscreen as they would with error messages.

Declaration Section

Compiler Mode

The CSPro compiler operates with the requirement that variables are declared explicitly. You must declare all variables not defined in your dictionary; otherwise, the variables will be flagged as errors by the compiler. The advantage of this is that you do not have to worry about misspelled names.

Prior to CSPro 7.2, there was a compilation mode called implicit that allowed for on-the-fly variable declarations. This was removed because it could easily lead to mistakes based on programmers mistyping variable names. There were some purposes where implicit declaration mode could be useful, which can now be achieved by using the `ensure` statement.

Variables

In CSPro you can declare `numeric` or `string` variables. Variable names must contain only letters, numbers, or the underscore (_) character, and must begin with a letter. Names are case insensitive, that is, uppercase and lowercase letters are considered the same. (For example, `myvar`, `MYVAR`, and `MyVar` are all equivalent.) Variables of all types follow the same rules for names.

Numeric Variables

In CSPro, numeric variables are stored internally in floating point format. They can accommodate numbers of extremely small or large size, positive or negative. Numeric variables can be global in scope, meaning that you can assign or get the value of a variable from any other event, or local in scope, meaning that you can only access the value from within a single event. A numeric variable may be up to 15 digits in size. It is equivalent to a float or double variable.

String Variables

String variables in CSPro store alphanumeric data. You must declare a string variable using either the `alpha` or `string` statements. As with numeric variables, string variables can be either global or local in scope.

See also: Array Object, Text Strings, Logic Objects and Dot Notation

Alias Statement

Format

```
alias aliased_name : original_name;
```

Description

The `alias` statement allows for the creation of new names to alias, or to provide an alternative reference to, names in a program. This can be used to shorten, or lengthen, variable names, or to match names in a dictionary with preexisting code using a certain naming convention.

You can define aliases in logic, or directly in the dictionary editor.

Example 1: Shortening Dictionary Names to Quicken the Typing of Variables

```
alias   P14 : P14_AGE,
        P15 : P15_RELATIONSHIP,
        P16 : P16_SEX;
```

Example 2: Using Aliases to Standardize Variable Names so that Logic Can be Reused Across Programs

```
alias   AGE             : P14_AGE,
        RELATIONSHIP    : P15_RELATIONSHIP,
        SEX             : P16_SEX;
```

In both examples, `P14_AGE`, `P15_RELATIONSHIP`, and `P16_SEX` are the names of items declared in a dictionary. Once the `alias` statement has been specified (in `PROC GLOBAL`), the names `P14` and `P14_AGE` can be used interchangeably. For example, these two statements are identical:

```
if P14_AGE < 18 and P16_SEX = 2 then

if AGE < 18 and SEX = 2 then
```

Ensure Statement

Format

```
ensure numeric variable_name1[, ... , variable_nameN];
```

Description

The `ensure` statement allows for the conditional creation of numeric variables in the `PROC GLOBAL` section of logic. One or more variable names are specified, and if the name of the variable exists (because of being declared previously in logic, or defined in a dictionary), then it will not be created. If it does not exist, a temporary variable is created just as if it were a declared numeric variables. The initial value of the variable is 0.

This statement is useful when working with applications that reuse blocks of logic with different dictionaries. If a certain routine depends on a dictionary variable, you can use this statement to ensure that the logic compiles without issue. It is important to make sure that your logic works regardless of whether the variable is defined in a dictionary or from the ensure statement.

In versions of CSPro prior to 7.2, you could simulate the behavior of `ensure` by using set implicit mode.

Example

```
PROC GLOBAL

ensure numeric SUPERVISOR_CODE;

PROC DATA_TRANSMISSION

    // assuming that 0 is never a valid value for SUPERVISOR_CODE, then
    // you could have two synchronization routines in your code, one for
    // surveys where there are supervisors (and SUPERVISOR_CODE is defined
    // in a dictionary), and another for surveys where interviewers directly
    // send data to headquarters

    if SUPERVISOR_CODE = 0 then
        SyncWithHeadquarters();

    else
        SyncWithSupervisor();

    endif;
```

See also: Numeric Statement

User-Defined Functions

User-defined functions are coded in the declaration portion (`PROC GLOBAL`) of an application. Once defined, they can be used anywhere in an application. Functions are used to perform operations that are used in several different places in an application.

Functions are of the form:

```
return_value = function_name(parameter_list);
```

Functions may include a list of parameters, which can vary depending on the function call's requirements. This list may be empty (that is, it contains no parameters between the opening and closing parentheses) or it may contain one or more parameters. Each parameter specifies a variable or object that is used by the statements within the function.

Numeric, string, and alphanumeric variables are local to the function. That is, if a variable is passed as an argument, its value in the rest of the application will not be changed by actions within the function (this is called "pass by value"). On the other hand, objects (such as arrays and file handlers) passed as arguments refer to the source variable and interactions on the variable affect the source variable (this is called "pass by reference"). If you want to pass a numeric or string variable by reference, you can use the `ref` keyword to signify that changes made in the function should affect the source variable.

A user-defined function:

- Returns a single value, either numeric or alphanumeric.
- Can contain CSPro statements and functions, as well as other user-defined functions. If no return value is assigned to the function, the value `default` (or a blank string) is returned.

The `function` statement allows the creation of a user-defined function.

You can use the `invoke` function to call a function with the function name specified only at runtime. Additionally, you can execute user-defined functions while viewing HTML content by using the JavaScript interface.

See also: Function Statement, Additional Examples of User-Defined Functions

Array Object

In logic, an `array` is an object that contains a collection of elements of the same type, either numeric, alphanumeric, or string. An array can be of many dimensions, though each of these dimensions is of a fixed size. Arrays can be used in various processing operations, including defining a list of constant values (like month names) or working with hotdecks and DeckArrays.

Whenever an array variable is used in the application, a value or numeric expression for each dimension must be given. The initial array contents are zero (if numeric) and blank (if alphanumeric or string) until a value for each dimension is assigned. If using a numeric saved array, the initial array contents are `default`.

If you want the behavior of an array but without fixed dimensions, you can use the hashmap object, which is an associative array.

Functionality

An array is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`clear`	Resets all array values to the default value.

`length`	Returns the size of an array dimension.

Assignments

When an array is used as an argument to a user-defined function, it is passed by reference.

See also: HashMap Object, List Object

Audio Object

In logic, an `audio` object represents an audio recording. Audio recordings may be loaded from audio files, saved to audio files, played back or recorded. Audio objects are typically used to record portions of an interview either interactively or in the background. With interactive recording an audio recorder is displayed to the user who then controls when the recording is started and stopped. The user must complete the audio recording before moving to the next survey question. With background recording, the audio recorder is started and stopped using CSPro logic and the user can continue with the survey while the audio is recorded.

Audio recording is typically implemented by declaring a variable of type `audio`, calling either the `audio.record` or `audio.recordInteractive` function, and then calling the `audio.save` function to save the audio to a file.

Audio recording and playback are currently only supported on Android.

Functionality

An audio object is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`clear`	Erases the audio recording currently stored in the audio object.
`concat`	Appends an audio recording to the recording stored in the audio object.
`length`	Returns the length, in seconds, of the recording in the audio object.
`load`	Reads an audio file and places the contents in the audio object.
`play`	Launches an audio player to play back the contents of the audio object.
`record`	Starts recording audio in the background.
`recordInteractive`	Starts interactive audio recording.
`save`	Writes the audio recording to a file.
`stop`	Stops the current background recording.

In addition to these object functions, audio objects can be used as arguments to the `filename` function.

Assignments

Audio objects can be assigned to other audio objects, which will replace the recorded audio with the recording from the assigned audio object.

```
audio_name = another_audio_name;
```

It is also possible to assign audio objects to a document:

```
document_name = audio_name;
```

You can also make the reverse assignment, assigning a document to an audio object. A runtime error will occur if the document's data was not audio data.

```
audio_name = document_name;
```

When an audio object is used as an argument to a user-defined function, it is passed by reference.

Example

```
PROC AUDIO_QUESTION

onfocus

// Variable to store the recording
audio recording;

// Save recording to a file that includes case id-items to differentiate it from
// audio saved in other cases
string nameOfAudioFile = maketext("%v%v%vAUDIO_QUESTION.m4a", PROVINCE, DISTRICT, EA);

// If no audio is recorded for this question then the only option is to record,
// otherwise allow user to re-record, play or clear recording before continuing
valueset vs;
if fileexist(nameOfAudioFile) and recording.load(nameOfAudioFile) then
    vs.add("Re-record", 1);
    vs.add("Play recording", 2);
    vs.add("Clear recording", 3);
    vs.add("Next question", 4);
else
    vs.add("Record", 1);
endif;
setvalueset($, vs);

postproc

if $ = 1 then
    // Record/re-record
    recording.clear();
    string message = "Record the respondents answer to this question";
    numeric seconds = recording.recordinteractive(message);
    if seconds > 0 then
        recording.save(nameOfAudioFile);
    else
        errmsg("No audio recorded. Please try again");
    endif;
    reenter;
elseif $ = 2 then
    // Play back recording
    recording.play();
    reenter;
elseif $ = 3 then
    // Clear the recording
    filedelete(nameOfAudioFile);
    reenter;
elseif $ = 4 then
    // Continue to next question
endif;
```

See also: Multimedia Features

Document Object

In logic, a `document` object facilitates the storage of text or binary files. Unlike CSPro data files, or text files (accessed via `file`), the contents of a document cannot be directly edited.

Functionality

A document object is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
load	Reads a file and stores its contents in the document object.
save	Writes the contents of the document object to a file.
view	Displays the contents of the file held by the document object.
clear	Clears the document object's contents.

In addition to these object functions, document objects can be used as arguments to the filename and view functions.

Assignments

Document objects can be assigned to other document objects, which will replace the document's contents with the contents from the assigned document object.

```
document_name = another_document_name;
```

It is also possible to assign audio, geometry, and image objects to a document. These assignments will always succeed.

```
document_name = audio_name;
document_name = geometry_name;
document_name = image_name;
```

You can also make the reverse assignment, assigning a document to an audio, geometry, or image object. A runtime error will occur if the document's data cannot be converted to those objects.

```
audio_name = document_name;
geometry_name = document_name;
image_name = document_name;
```

When a document is used as an argument to a user-defined function, it is passed by reference.

Example

```
document manual;
manual.load("Farmers Markets Survey.pdf");
manual.view();
```

See also: Image Object

File Object

Most files in CSPro are data files, whose structure is defined by a data dictionary are are used in logic by referring to their dictionary names. However, it is also possible to write to text files without an associated data dictionary. Lines of text can be written to a file or read from a file, and these files can also be used in export statements.

The physical name of the file can be specified in the Define File Associations dialog when the application is run, in a PFF file, or by using the file.open, open, or setfile functions.

Functionality

A file is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
open	Associates the file object with an existing or a new file on the disk (equivalent to setfile).
close	Closes the open file (equivalent to close).
read	Reads one or more lines of text from the file (equivalent to fileread).

`write`	Writes one or more lines of text to a file (equivalent to `filewrite`).

In addition to these object functions, files can be used as arguments to functions such as `filecopy`, `filedelete`, `fileempty`, `fileexist`, `filename`, `filerename`, `filesize`, `open`, and `pathname`.

Assignments

When a file is used as an argument to a user-defined function, it is passed by reference.

Example

```
file login_times_file;
login_times_file.open("Login Times.csv", append);
login_times_file.write("%s, %s, %d", getdeviceid(), getoperatorid(), timestamp());
login_times_file.close();
```

Freq Object

In logic, a `freq` can be one of two things: an "unnamed" statement used to generate frequencies with a specific set of behaviors regarding how values are tallied, or a CSPro logic object with a name, which allows for the creation of frequencies with control over when the values are tallied. This page is only relevant if using named frequencies.

Functionality

A freq is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`clear`	Resets all tallies to 0.
`save`	Saves frequency tables to a file.
`tally`	Tallies the frequencies.
`view`	Displays frequency tables in an embedded web browser.

In addition to these object functions, freq objects can be used as an argument to the `view` function.

Assignments

Unlike most CSPro logic objects, freq objects cannot be assigned to other freq objects.

If the freq has only one variable as part of its frequency list, individual tallies of the freq can be accessed by using an index:

```
freq_name(value) = tally;

tally = freq_name(value);
```

When a freq is used as an argument to a user-defined function, it is passed by reference.

Example 1

```
freq hh_status_freq(HH_STATUS);

forcase LISTING_DICT where FIPS = 69 do
    hh_status_freq.tally();
endfor;

hh_status_freq.view()
                heading("Household Status - Northern Mariana Islands");

// show the detailed household status report when some households are incomplete
if hh_status_freq(1) > 0 then
    ShowDetailedHouseholdStatusReport();
endif;
```

Example 2

```
PROC GLOBAL

// create a freq object that will tally the sex and age of
// the first occurrence (the head)
freq head_sex_age_freq(P03_SEX(1), P04_AGE(1));

PROC CENSUS_DICTIONARY_FF

    // save the frequency, formatting the ages using a
    // 5-year value set
    head_sex_age_freq.save()
                      valueset(P04_AGE_5YEAR_VS);

PROC QUEST

    // only tally households that are not vacant
    if count(PERSON_REC) > 0 then
        head_sex_age_freq.tally();
    endif;
```

See also: Freq Statement (Unnamed)

Geometry Object

In logic, a `geometry` is an object that can be used to display and manipulate vector geometry: points, lines, and polygons. The `geometry` object can load and save data in GeoJSON format. You can display the geometry on a map, interactively trace polygons on a map, and compute the area and perimeter of polygons.

Note that displaying geometry on a map and polygon tracing are currently only supported on Android. These functions will do nothing when run on Windows.

Functionality

A geometry is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`load`	Read geometry from a GeoJSON file.
`save`	Write geometry to a GeoJSON file.
`clear`	Delete all data from the geometry.
`tracePolygon`	Allow user to draw polygon by tapping points on a map.
`walkPolygon`	Allow user to draw polygon by walking the perimeter and saving the locations using the GPS.
`area`	Compute the area of a polygon.
`perimeter`	Compute the perimeter of a polygon.
`maxLatitude`	Get the largest latitude value of the geometry.

maxLongitude	Get the largest longitude value of the geometry.
minLatitude	Get the smallest latitude value of the geometry.
minLongitude	Get the smallest longitude value of the geometry.
getProperty	Retrieve one of the geometry's properties.
setProperty	Set the value of one of the geometry's properties.

In addition to these object functions, geometry objects can be used as arguments to the `filename`, `map.addGeometry`, `map.clearGeometry`, and `map.removeGeometry` functions.

Assignments

Geometry objects can be assigned to other geometry objects, which will replace the vector geometry with the contents from the assigned geometry object.

```
geometry_name = another_geometry_name;
```

It is also possible to assign geometry objects to a document:

```
document_name = geometry_name;
```

You can also make the reverse assignment, assigning a document to a geometry object. A runtime error will occur if the document's data was not valid geometry data.

```
geometry_name = document_name;
```

When a geometry object is used as an argument to a user-defined function, it is passed by reference.

Example

```
// Declare a geometry
geometry mygeometry;

// Capture polygon
mygeometry.tracePolygon();

// Compute the area of the captured polygon
AREA = mygeometry.area();

// Save the geometry to a file
mygeometry.save("polygon.geojson");
```

See also: Map Object

HashMap Object

In logic, a `hashmap` is similar to an array but has dimensions that can be either non-consecutive numbers or strings. A hashmap is an associative array that can be used to store numbers or strings and can dynamically grow or shrink in size.

Functionality

A hashmap is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
clear	Removes all values from the hashmap.
contains	Returns whether a specified key exists.
getKeys	Fills a list with the hashmap's keys.

`length`	Returns the number of keys.
`remove`	Removes a key.

Assignments

Hashmaps can be assigned to other hashmaps, which will replace the initial hashmap with the values of the assigned hashmap:

```
hashmap_name = another_hashmap_name;
```

When assigning a hashmap to another hashmap, both must have the same value types, and the dimension types must be compatible.

Individual elements of hashmaps can be retrieved or set by using an index to specify the dimension keys:

```
hashmap_name(key_value1[, key_value2, ...]) = value_to_set_or_modify;

value_to_retrieve = hashmap_name(key_value1[, key_value2, ...]);
```

When a hashmap is used as an argument to a user-defined function, it is passed by reference.

Example

```
PROC GLOBAL

hashmap invalidValuesByPerson default(0);

PROC CENSUS_LEVEL

    list string person_list;

    invalidValuesByPerson.getKeys(person_list);

    do numeric counter = 1 while counter <= person_list.length()
        errmsg("%s had %d invalid values", strip(person_list(counter)),
                                           invalidValuesByPerson(person_list(counter)));
    enddo;

    invalidValuesByPerson.clear();

PROC SEX

    if not invalueset(SEX) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;

PROC AGE

    if not invalueset(AGE) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;
```

See also: Array Object, List Object

Image Object

In logic, an `image` object facilitates the storing and manipulating of images. Images can contain photos, signatures, or other kinds of visual media. CSPro supports reading and writing to JPEG, PNG, and BMP image formats.

Functionality

An image object is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`load`	Reads an image from a file and stores its contents in the image object.
`save`	Writes the contents of the image object to a file.
`width`	Returns the width of the image in pixels.
`height`	Returns the height of the image in pixels.
`resample`	Resamples the image to change the image's dimensions.
`takePhoto`	Takes a photo using a device's camera and stores the photo in the image object. (Android only.)
`captureSignature`	Allows the drawing of a signature and stores the captured signature in the image object. (Android only.)
`view`	Displays the image held by the object.
`clear`	Clears the image object's contents.

In addition to these object functions, image objects can be used as arguments to the `filename` and `view` functions.

Assignments

Image objects can be assigned to other image objects, which will replace the image's contents with the contents from the assigned image object.

```
image_name = another_image_name;
```

It is also possible to assign image objects to a document:

```
document_name = image_name;
```

You can also make the reverse assignment, assigning a document to an image object. A runtime error will occur if the document's data was not a valid image.

```
image_name = document_name;
```

When an image is used as an argument to a user-defined function, it is passed by reference.

Example

```
image roof_photo;

if roof_photo.takePhoto("Take a photo of the household's roof.") then

    // in case the device's camera takes photos with an unnecessarily
    // large resolution, resample the image to a more reasonable size
    roof_photo.resample(maxWidth := 1600, maxHeight := 1200);

    // save the image using the household key...
    string base_filename = path.concat(application, "Roof Photos", key(HH_DICT));

    // ...with 90 quality to prevent the JPEG from being too large
    roof_photo.save(base_filename + ".jpg", quality := 90);

endif;
```

See also: Document Object

List Object

In logic, a `list` is similar to an one-dimensional array but without a defined size. That is, a list is a collection of values, either numeric or string, that can grow or shrink in size.

Functionality

A list is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`add`	Adds a single value, or a list of values, to the end of a list.
`clear`	Removes all values from the list.
`insert`	Inserts a single value, or a list of values, at a given position in the list.
`length`	Returns the size of the list.
`remove`	Removes the value at a given position from the list.
`removeDuplicates`	Removes duplicate values from the list.
`removeIn`	Removes values from the list that are specified in an in list.
`seek`	Returns the index of a specified value.
`show`	Displays the list's values (similarly to `accept`) and returns the index of the operator's selection.
`sort`	Sorts the list's values in ascending or descending order.

In addition to these object functions, lists can be filled when used as arguments to functions such as `dirlist`, `keylist`, and `hashmap.getKeys`.

Assignments

Lists can be assigned to other lists, which will replace the initial list with the values of the assigned list:

```
list_name = another_list_name;
```

Individual elements of lists can also be modified or added by using a one-based index:

```
list_name(index) = modify_value;
list_name(list_name.length() + 1) = add_value;
```

When a list is used as an argument to a user-defined function, it is passed by reference.

Example

```
list string respondent_query;

do numeric counter = 1 while counter <= count(NAME)
    respondent_query.add(NAME(counter));
enddo;

numeric respondent_index = respondent_query.show("Who in the household is responding to
questions?");
```

See also: Array Object, HashMap Object, ValueSet Object

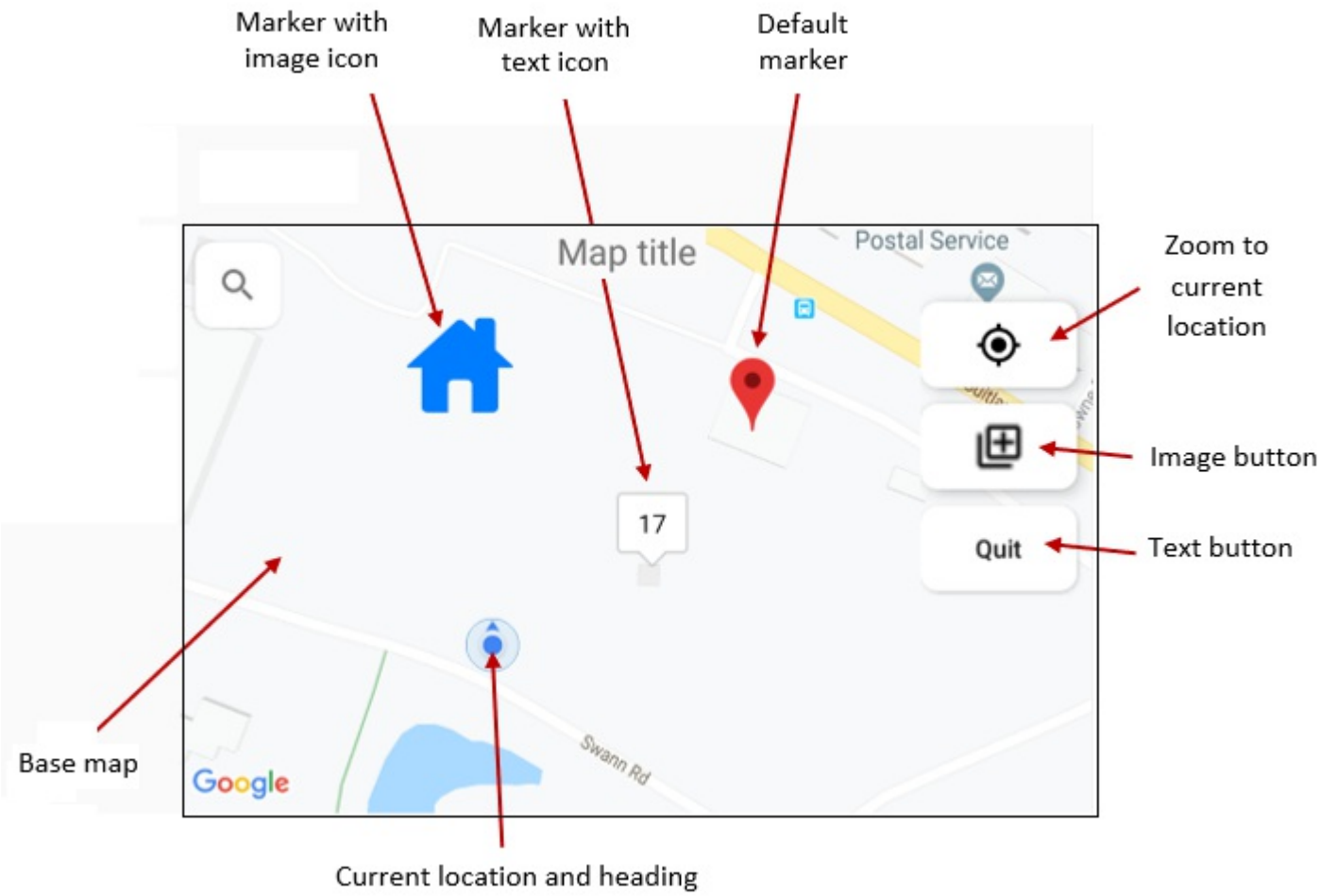
Map Object

In logic, a `map` is an object that can be used to display and control an interactive map. The map may be panned and zoomed with touch controls and may optionally show the user's current location. Using map functions you can add markers at geographic positions on the map, add custom buttons, and set the geographical area displayed. You can also specify user-defined functions in your logic to be called when the user taps on markers, buttons, and on the map itself. Together, these functions allow for rich map-based interactions such as showing households on a map and

launching an interview when the user taps a household.

To display a map, call the `map.show` function. The map will be displayed allowing the user to interact with it until the user taps the back button or the `map.hide` function is called from program logic.

You can call map functions to add markers, buttons, set the base map, and pan/zoom before showing the map. This will be more efficient than adding them afterwards, especially when adding large numbers of markers. You can also call the map functions after showing the map from within any of the user-defined callback functions for map, marker, and button clicks and drags.



You can add markers to the map using the `map.addMarker` function. You can replace the default marker icon with a custom image by calling `map.setMarkerImage` or with a text icon using `map.setMarkerText`.

By default, the map contains a button to zoom to the users current location. You can add additional buttons to the map by calling `map.addImageButton` or `map.addTextButton`.

You add vector geometry, such as polygons and polylines, to the map using the `map.addGeometry` function.

By default, the map will display a base map from Google Maps, which requires an Internet connection. With no Internet connection the base map will be empty. To display a map without an Internet connection you can copy an offline map file to your device and pass the file to the `map.setBaseMap` function.

Functionality

A `map` is a CSPro logic object that allows the following functions to be called via dot notation:

Function	Description
show	Display the map and allow the user to interact with it.
hide	End displaying the map if it is currently showing.

setTitle	Set the map title.
setBaseMap	Change the map source to an online or offline base map.
zoomTo	Pan/zoom the map.
showCurrentLocation	Display the current location (as retrieved from the GPS) on the map.
saveSnapshot	Save the currently-displayed map as an image.
setOnClick	Set the user-defined function that is called when the user taps on the map.
getLastClickLatitude	Retrieve the latitude of the last point on the map the user tapped.
getLastClickLongitude	Retrieve the longitude of the last point on the map the user tapped.
addMarker	Place a marker on the map at the specified latitude and longitude.
removeMarker	Delete a single marker from the map.
clearMarkers	Delete all markers from the map.
setMarkerImage	Set the icon displayed on the map for a specific marker.
setMarkerText	Set text displayed on the map for a specific marker.
setMarkerDescription	Set text displayed in a popup window when a specific marker is tapped.
setMarkerOnClick	Set the user-defined function that is called when the user taps a marker.
setMarkerOnClickInfoWindow	Set the user-defined function that is called when the user taps on a marker's popup info.
setMarkerOnDrag	Set the user-defined function that is called when the user a drags a marker on the map.
setMarkerLocation	Move an existing marker to a new position on the map.
getMarkerLatitude	Retrieve the latitude of a marker on the map.
getMarkerLongitude	Retrieve the longitude of a marker on the map.
addGeometry	Add vector geometry to the map.
removeGeometry	Delete vector geometry from the map.
clearGeometry	Delete all vector geometry from the map.
addTextButton	Create a button with a text label on the map.
addImageButton	Create a button with an icon on the map.
removeButton	Delete a button on the map.
clearButtons	Delete all buttons from the map.

Assignments

When a map is used as an argument to a user-defined function, it is passed by reference.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
mymap.addMarker(38.84839, -76.931098);

// Display the map
mymap.show();
```

See also: Display the Case Listing on a Map, Geometry Object

Pff Object

In logic, a `pff` is an object that contains information about a CSPro Program Information File (.pff). PFF files are used to run CSPro applications or tools in production mode. Using the pff object in logic, it is possible to dynamically modify the properties used when running an application or tool.

Functionality

A pff is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`load`	Loads the contents a PFF file from the disk.
`save`	Saves a PFF file to the disk.
`getproperty`	Gets the value associated with a PFF property.
`setproperty`	Sets the value associated with a PFF property.
`exec`	Executes the application or tool associated with the PFF.

In addition to these object functions, pff objects can be used as arguments to the `filename` function.

Assignments

Pff objects can be assigned to other pff objects, which will replace the PFF's properties with the properties of the assigned PFF:

```
pff_name = another_pff_name;
```

When a pff is used as an argument to a user-defined function, it is passed by reference.

Example

```
PROC INTERVIEWER_MENU

    // run the listing program in the interviewer's assigned cluster
    if INTERVIEWER_MENU = 1 then
        pff listing_pff;
        listing_pff.load("Listing.pff");
        listing_pff.setproperty("Key", maketext("%v", CLUSTER));
        listing_pff.exec();
    endif;
```

See also: Program Information File (.pff)

SystemApp Object

In logic, a `SystemApp` is an object that contains information about the parameters to be used when calling a system application. A system application is an application that is loaded on a device and is generally not part of the CSPro suite of applications and tools. On Android, the SystemApp object also stores any information that resulted from the running of the system application.

Functionality

A SystemApp is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`clear`	Clears any stored arguments.

exec	Executes the system application.
getResult	Gets a result returned from the system application.
setArgument	Stores the argument for use when executing the system application.

Assignments

When a SystemApp is used as an argument to a user-defined function, it is passed by reference.

Example

```
// play the training video in Windows Media Player
SystemApp windows_media_player;
windows_media_player.setArgument(pathconcat("../Videos/Census Training.mp4"));
windows_media_player.exec("wmplayer.exe");
```

See also: ExecSystem Function (Desktop), ExecSystem Function (Mobile), View Function

ValueSet Object

In logic, a `valueset` is an object that contains information about the possible values that are considered permissible for a field. Each possible value contains a label, a code (the valid value), and can contain a link to an image shown when the value set is displayed. The value set object, once passed to the `setvalueset` function, becomes an item's dynamic value set.

Functionality

A value set is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
add	Adds a single value, or all the values from an existing value set, to the value set.
clear	Removes all values from the value set.
length	Returns the size of the value set.
randomize	Randomizes the order of entries in the value set.
remove	Removes the value with the specified code from the value set.
show	Displays the value set's labels (similarly to `accept`) and returns the code of the operator's selection.
sort	Sorts the order of entries in the value set by either label or code.

In addition to these object functions, value sets contain two built-in list objects: `valueset_name.codes` and `valueset_name.labels`, which contain the codes and labels for the value set. These are read-only lists and can be used in most ways that list objects can be used.

Several functions accept value sets as arguments, including: `getimage`, `getlabel`, `invalueset`, `maxvalue`, `minvalue`, `randomin`, `randomizevs`, and `setvalueset`. Value sets can also be used with the `in` operator.

Assignments

Value sets can be assigned to other value sets, which will replace the initial value set with the values of the assigned value set:

```
valueset_name = another_valueset_name;
```

When a value set is used as an argument to a user-defined function, it is passed by reference.

Example

```
PROC RESPONDENT

preproc

    valueset respondent_valueset;

    do numeric counter = 1 while counter <= count(PERSON_REC)
        if AGE(counter) >= 15 and USUAL_MEMBER(counter) = 1 then
            respondent_valueset.add(NAME(counter), counter);
        endif;
    enddo;

    setvalueset(RESPONDENT, respondent_valueset);
```

See also: Array Object, List Object

Procedural Sections

Statements

A procedure contains a series of statements. Each statement is a complete instruction to the computer.

Executable Statements

Executable statements begin with a command and end with a semicolon (;). They are made up of a combination of commands, keywords, expressions, and functions. For example:

```
skip to Q103;
```

`skip` is a command, `to` is a keyword, and `Q103` is the name of a data entry field.

Assignment Statements

Assignment statements set a variable equal to the value of an expression and do not contain commands. If the expression is a string expression, then the variable must be alphanumeric. If the expression is numeric or conditional, then the variable must be numeric. For example:

```
AGE = 10;
Q102 = previousAge;
Y = sqrt(X);
NAME = "John Doe";
SEX_RATIO = MALES / FEMALES;
```

Proc Statement

Format

```
PROC PROCEDURE_NAME
```

Description

The `PROC` statement declares the beginning of the procedures for a data entry or batch processing element. The ***PROCEDURE_NAME*** must always be the name of an object in the forms or edit tree. If you are in the logic view and select a processing element from the tree, the logic view will automatically generate the "`PROC PROCEDURE_NAME`" heading.

If you plan to write logic for more than one procedure, the order of procedures must be as follows:

```
PROC ITEM_NAME

preproc

    // statements

onfocus

    // statements

onoccchange

    // statements (only valid for rosters and multiply-occurring forms)

killfocus

    // statements

postproc

    // statements
```

See also: PreProc Statement, OnFocus Statement, OnOccChange Statement, KillFocus Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

PreProc Statement

Format

```
preproc
```

Description

The `preproc` statement declares that the statements following it are executed at the beginning of a run, case, level, record, form, roster, or field.

In data entry applications, statements in a `preproc` procedure are executed when you move **forward** onto an object. Moving forward occurs when the execution flow moves the cursor onto the object, or when the user goes forward to the object by any means (mouse-click, tab key, arrow keys, etc.). If you move **backward** onto an object, `preproc` statements are **not** executed. Moving backward occurs if a keyer reenters a value, goes backward with a mouse click, or uses the Shift+tab or arrow keys to move backward. If you want to execute the statements when you move both forward and backward onto a field, code them in the `onfocus` procedure.

In batch edit applications, a `preproc` is used to execute logic at the beginning of a run, case, level, or record. For an item there is no difference between placing your logic in a `preproc` or `postproc`.

Remember, if you don't code a `preproc` or `postproc` in a `PROC`, all instructions are considered `postproc` statements by default.

Example

```
PROC INTERVIEW_DATE

preproc

    INTERVIEW_DATE = sysdate("YYYYMMDD");
```

See also: Proc Statement, OnFocus Statement, OnOccChange Statement, KillFocus Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

OnFocus Statement

Format

```
onfocus
```

Description

The onfocus statement declares that the statements following it are executed when a form, roster, or field becomes active.

Statements in an onfocus procedure are executed when you move onto the object in which they are coded. When moving **forward**, any preproc statements are executed before the onfocus statements. However, when moving **backward**, the preproc statements are **not** executed and only the onfocus statements are executed.

Example

```
PROC CHRONIC_ILLNESS

onfocus

    if curocc() = 1 then
        setvalueset(CHRONIC_ILLNESS,CHRONIC_ILLNESS_FIRST_VS);

    else
        setvalueset(CHRONIC_ILLNESS,CHRONIC_ILLNESS_SECOND_VS);

    endif;
```

See also: Proc Statement, PreProc Statement, OnOccChange Statement, KillFocus Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

OnOccChange Statement

Format

```
onoccchange
```

Description

The onoccchange statement declares that the statements following it are executed at the time that a group's current occurrence changes. The onoccchange statement can only be coded in a group: a roster or a multiply occurring form.

During data entry, whenever the user moves from one occurrence to another (forward or backward), the onoccchange of the group will be executed.

The onoccchange statement is not very common, but it may be useful in some CAPI applications, particularly to set question text fills.

Example

```
PROC PERSON_FORM

onoccchange

    if curocc() > 0 then
        personName = strip(FIRST_NAME(curocc())) + " " + strip(LAST_NAME(curocc()));
    endif;
```

See also: Proc Statement, PreProc Statement, OnFocus Statement, KillFocus Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

KillFocus Statement

Format

```
killfocus
```

Description

The `killfocus` statement declares that the statements following it are executed when a form, roster, or field **stops** being active.

Statements in an `killfocus` procedure are executed when you move off the object in which they are coded. When moving **forward**, any `killfocus` statements are executed before any `postproc` statements. However, when moving **backward**, the `postproc` statements are **not** executed and only the `killfocus` statements are executed.

The `killfocus` statement is not very common as most logic operations and checks only occur when moving forward in the application, in which case they can be coded in a `postproc`.

Example

```
PROC CEB_MALE

killfocus

    CEB_TOTAL = visualvalue(CEB_MALE) + visualvalue(CEB_FEMALE);
```

See also: Proc Statement, PreProc Statement, OnFocus Statement, OnOccChange Statement, PostProc Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

PostProc Statement

Format

```
postproc
```

Description

The `postproc` statement declares that the statements following it are executed at the end of a run, case, level, record, form, roster, or field.

In data entry applications, statements in a `postproc` procedure are executed when you **complete** an object; that is, flow off of it. When you click off a field, manually skip from a field, or move backward, the `postproc` statements are **not** executed. If you want to execute the statements in these situations, code them in the `killfocus` procedure.

In batch edit applications, a `postproc` is used to execute logic at the end of a run, case, level, or record. For an item there is no difference between placing your logic in a `preproc` or `postproc`.

If you don't code a `preproc` or `postproc` in a `PROC`, all instructions are considered `postproc` statements by default.

Example

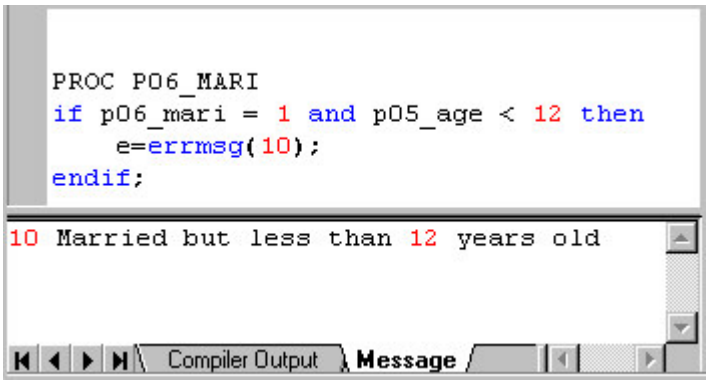
```
PROC MARITAL_STATUS

postproc

if MARITAL_STATUS <> 1 and AGE < 12 then
    errmsg("People aged less than 12 cannot be married.");
    reenter;
endif;
```

See also: Proc Statement, PreProc Statement, OnFocus Statement, OnOccChange Statement, KillFocus Statement, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

Logic

View Logic

To view the logic in a data entry application you can press icon; or in the **View** menu select "Logic"; or press **Ctrl+L**. To go back to the forms press ; or in the **View** menu select "Form"; or press **Ctrl+M**. From the Logic Window you can create or modify procedures that add logic to your application. The view is divided into two areas:



- Top: Text editor, where you write the logic statements

- Bottom: Message tab, where you create messages to be displayed during, or at the completion of, the execution of your application. This area is also used to display the results of the compiler after you compile a program. If you have errors in the logic, the compiler will display the error messages; otherwise it will display "Compile Successful".

Click on any element of the data entry or batch edit tree to see the logic which corresponds to that symbol. For example, if you click on a field, you see the logic for only that field. A group or level can also have logic associated with it. Click on the forms file node (usually the topmost node on form tree) to see the logic for the whole application. This is the way to see and enter logic for the global procedures.

See also: Create and Edit Logic, Order of Executing Data Entry Events, Order of Executing Batch Edit Events

Create and Edit Logic

You can use the CSPro language to write logic for virtually any part of your data entry or batch edit application: a level, roster, form, or field. In a data entry application you must make sure the screen has the logic view on the right and the data entry tree on the left, so you can click on the item for which you want to write logic. You can see the logic for the whole application by clicking on the forms file (usually the topmost node) on the data entry or batch edit tree.

Example: Programming a Message for the Keyer

Give a message for the keyer if there are married (or previously married) people under the age of 12. Click on the "Marital Status" field on the forms tree, `P06_MARITAL_STATUS` in our example. In the text editor, at the top of the logic view, you will see:

```
PROC P06_MARITAL_STATUS
```

Now enter the following:

```
if P06_MARITAL_STATUS <> 1 and P05_AGE < 12 then
    errmsg("Not 'never married' but less than 12 years old");
endif;
```

Note that this particular verification can be done only after data has been entered in both fields. If for some reason `AGE` is captured after `P06_MARITAL_STATUS`, then these instructions would be placed in the `AGE` field's logic.

Example: Programming a Skip

Program a skip after the marital status question to skip over "Age at First Marriage" if the person is never married. Click on the "Marital Status" field on the forms tree, `P06_MARITAL_STATUS` in our example. In the text editor, at the top of the logic view, enter:

```
PROC P06_MARITAL_STATUS

    if P06_MARITAL_STATUS = 1 then
        skip to P08_WORK_STATUS;
    endif;
```

Find and Replace Logic

Find

Press on the toolbar, or press **Ctrl+F**; or select **Find** from the **Edit** menu to open the Find dialog box. You can simply search for a text string by typing it in the dialog box and pressing Enter. You can then press the **F3** key to find the next occurrence of the text string. If you use the **Mark All** button, CSPro will show you each line that contains the text string by putting a yellow circle to the left of it.

Replace

Press **Ctrl+H**; or select **Replace** for the **Edit** menu to open the Replace dialog box. This is the standard Windows dialog box that allows you to replace one text string with another text string, either one at a time or all at once.

Compile Logic / An Application

Before CSPro can execute your program, it needs to check for errors in the logic. Errors include such problems as misspelling a command, not using proper command syntax, and putting logic in the wrong place. Warnings can also appear during compilation, and often occur from the use of deprecated features.

Compile Scope

You can choose to compile logic for a specific entity (such as a field, roster, or form) or for the entire application. To compile code for a specific entity, simply select that element from the form or batch edit tree. The associated logic for that item will be displayed in the logic view. If you are ready to compile the entire application, select the root node (the topmost entry of the data entry or batch edit tree) and compile.

How to Compile

Press on the toolbar; or from the **File** menu, select **Compile**; or press **Ctrl+K**.

Compiler Results

Error and warning messages will appear within the **Compiler Output** tab of the message view at the bottom of the workspace. A dot will appear to the left of the line of code containing the problem—red dots indicate an error, yellow dots indicate a warning. Double-clicking the error or warning message will move your cursor to the offending line within the logic view.

To facilitate code development, you should compile your logic on a regular basis to ensure not too many errors are introduced. Depending on the number and type of errors, it can cause CSPro to have difficulty determining the source of the error (for example, missing "end" or "endif" statements can cause a lot of problems). If there are unfinished code blocks that won't compile, they should be commented out until ready to complete.

Is My Program Ready to Run?

- If there are no errors or warnings, the system will display "Compile Successful" within the Compiler Output tab. You are ready to run your application.
- If there are no errors but some warnings, the system will present a message for each problem found within the Compiler Output tab. You can run your application, but should make note of the problems found and attempt to resolve them.
- If one or more errors are found, you will not be able to run your program until they are resolved.

Symbol Analysis

When viewing a data entry or batch application's logic, you can select **Edit** -> **Symbol Analysis** to view information about where each dictionary symbol, form symbol, and user-defined function is used in an application.

The analysis window shows a list of symbols used in logic, including:

- **Dictionary symbols**: dictionaries, records, items, and value sets
- **Form symbols**: forms, rosters, and blocks
- **Other symbols**: user-defined functions

After selecting a symbol, uses of that symbol are shown along with the line of logic where it is used. You can click on each use and a larger area of logic—not just the single line—will be shown.

Symbol Analysis can be used to quickly see where and how symbols are used throughout your application.

Language Elements

Delimiters

Delimiters separate elements in the CSPro language.

Delimiter	Symbol	Usage
Blank		Separate any language symbol.
Comma	,	Separate arguments within functions.
Quotation mark	" "	Specify the beginning and end of strings.
Apostrophe	' '	Specify the beginning and end of strings.
Semicolon	;	Specify the end of statements.
Colon	:	Separate the beginning and end of substrings.
Parentheses	( )	Specify the beginning and end of function arguments.
Brackets	[ ]	Specify substrings.

Comments

Comments make applications easier to understand. They are used to explain the purpose of specific statements or to temporarily disable statements to help find errors. Any text enclosed by braces `{ }` is a comment. The text within the brackets will be green. Comments can be placed anywhere in an application and are not checked for syntax errors. Comments can be nested, that is, comments within comments are allowed. Any text following double slashes `//` until the end of the line is also a comment.

The third line in the example below is a comment; a comment is also appended to the code in line five. It is highly recommended that the user document CSPro applications through the liberal use of comments.

Example

```
PROC HHDAY

    { Do not allow June to have more than 30 days }
    if HHMONTH = 6 and HHDAY > 30 then
        errmsg(1,"June",30,HHDAY); // if error, then display message
        reenter;
    endif;
```

Logic Preprocessor

Format

```
#if condition
#elseif condition
#else
#endif
```

Description

Before CSPro logic is compiled, it passes through a preprocessor that allows you to conditionally compile your code based on certain conditions. This is an advanced feature and is primarily useful when using external logic files where certain functionality may only successfully compile based on characteristics of the application.

The preprocessor uses the language's familiar `if`/`elseif`/`else`/`endif` keywords but with four differences:

1. The keyword must be preceded by a # character.
2. The # must be the first non-whitespace character on the line.
3. The whole condition must be specified on a single line.
4. A condition should not end with a `then` keyword.

An `#if` and `#endif` pair must exist, and optionally `#elseif` and `#else` can be included as part of a preprocessor block.

The preprocessor is activated before the rest of logic is compiled and thus has limited functionality. Simple math expressions and conditional checks are possible, and there are two functions that can be called:

```
AppType(app_type)
```

This function returns true if your application is of the type specified by ***app_type***. Supported types are `Entry`, `Batch`, and `Tabulation`.

```
exists(symbol_name)
```

This function returns true if a symbol exists at compile-time. The function checks for symbols that exist as part of an application (dictionary names, form names, etc.), and not symbols that are created during compilation (user-defined functions, objects, etc.).

Example 1

```
function string GetEAName(numeric province, numeric district, numeric ea)

    string ea_name = maketext("EA %04d-%02d-%03d", province, district, ea);

    // if we have access to the geocodes dictionary, add the EA description to the name
#if exists(GEOCODES_DICT)

    GEOCODES_PROVINCE = province;
    GEOCODES_DISTRICT = district;
    GEOCODES_EA = ea;

    if loadcase(GEOCODES_DICT, GEOCODES_PROVINCE, GEOCODES_DISTRICT, GEOCODES_EA) then
        ea_name = ea_name + " / " + strip(EA_DESCRIPTION));
    endif;

#endif

    exit ea_name;

end;
```

Example 2

```
function ValidateLiteracy()

    if HH_EDUCATION in 3:6 and HH_LITERACY <> 1 then

#if AppType(Entry)
        errmsg("Person with education level %d should be literate. Please
correct.", HH_EDUCATION);
        reenter;

#elseif AppType(Batch)
        errmsg("Person with education level %d should be literate. Literacy imputed to 1
(literate).", HH_EDUCATION);
        impute(HH_LITERACY, 1);

#endif

    endif;

end;
```

Variables and Constants

Data Items

Data items are defined in a data dictionary. You can assign a value to a data item, or get the value of a data item in any procedure. The following is an example of the use of data items:

```
PROC SEX

if AGE > 15 and NumberOfKids <> notappl then
    $ = 2;
endif;
```

However, in developing a data entry or batch edit application, it will frequently be necessary to define variables that do not exist in the data file(s) attached to the application. These variables may be used throughout the application, but only exist during the execution of the application.

See also: Introduction to Data Dictionary, Add or Modify Items, This Item ($)

This Item ($)

The dollar sign ($) is a short way of referring to a data item if used within that data item's procedure.

Pressing **Ctrl + 4** (4 is where the dollar sign is on many keyboards) will insert the name into the logic buffer.

Example

```
PROC AGE

    if MARITAL_STATUS > 1 then // ever married

        if $ < 12 then // the same as AGE < 12
            errmsg("Person too young (%d) to be married",$);
        endif;

    endif;
```

Subscripts

Items with multiple occurrences or in multiple records have one name (the item name), but can occur multiple times. In order to indicate the specific occurrence of the item, you may need to use an index or subscript. The subscripts are integers and are numbered from 1.

Imagine that the SEX is an item in the multiple record CHILD.

The expressions

`SEX(1)` refers to the sex of the first child.

`SEX(3)` refers to the sex of the third child.

`SEX(i)` refers to the sex of the **i**th child.

Subscripts can be numeric expressions as well as numeric constants. For example, the expression

```
SEX(curocc(CHILD));
```

refers to the current occurrence of CHILD. (curocc is a function that returns the current occurrence of a multiple record). When referring to multiply-occurring items within the scope of their repetition, you do not need to use subscripts, as the current occurrence will be assumed. For example, suppose you have a population record that has multiply occurrences, and belonging to that record are the three variables SEX, AGE, and FERTILITY. If your code is contained within any of these variables' procedures, you do not need to use subscripts. To illustrate:

Example 1:

```
{this will check the sex and fertility values for each person in the household}
 PROC SEX
   if $ = 1 then
     if fertility <> notappl then
       errmsg ("male found with fertility");
     endif;
   elseif $ = 2 then
     if age < 10 and fertility <> notappl then
       errmsg ("underage female found with fertility data");
     endif;
   else
     errmsg ("invalid sex code (sex=%d)", $);
  endif;
```

However, if you were to place the exact same logic elsewhere in your program, you would have to programmatically mimic the looping mechanism, and use subscripts. For example, if the above code were placed in the QUEST procedure, it would need to be adjusted as follows:

Example 2:

```
 PROC QUEST
   NumPeople=count (POP_RECS);
   do varying i=1 while i <= NumPeople
     if sex(i) = 1 then
       if fertility(i) <> notappl then
         errmsg ("male found with fertility");
       endif;
     elseif sex(i) = 2 then
       if age(i) < 10 and fertility(i) <> notappl then
         errmsg ("underage female found with fertility data");
       endif;
     else
       errmsg ("invalid sex code (sex=%d)", sex(i));
     endif;
   enddo;
```

Numbers

Numbers may be any positive or negative integer or decimal value. Negative numbers have a leading minus (-) sign. Positive numbers have no sign, but can have an optional leading plus (+) sign. Numbers can have up to 15 significant digits. Numbers must not have thousands separators. Decimal points can be either period (.) or comma (,) depending on the "Regional Options" setting of the computer.

Text Strings

A text string is any set of characters in the computer's character set enclosed between a pair of quotation marks (") or apostrophes ('). Any spaces enclosed within the quotation marks or apostrophes are considered part of the text string. Upper- and lower-case letters may be used. However, a text string 'a' is different from a text string 'A.' If you wish to have apostrophes (') embedded within your string, you must use the quotation marks (") to enclose it, and vice-versa. For

example:

```
MyString = 'That's great!';
```

This would set MyString to "that" and the trailing "s great!" would be considered outside the string, and would therefore provoke a compiler error. Thus, if you wanted to accomplish the above, you must write:

```
MyString = "That's great!";
```

Similarly, if you wanted to embed quotation marks within your string, you must write the string as follows:

```
MyString = 'The chair is 23" high';
```

Strings that are surrounded by quotation marks will appear in **magenta**. Strings that are surrounded by apostrophes will appear in **black**. We recommend using quotation marks ("), and if the text of the string does not turn magenta when you think you have completed entering it, it will be quickly apparent that you have not terminated your string properly.

Expressions

Expressions

An expression is a combination of operators and operands. Operands can be constants, items, variables, functions, or some combination thereof. Operators can be arithmetic (+, -, *, /), relational (=, <>, >, <, >=, <=) or logical (and, or, not). Every expression evaluates to a value and can therefore be used as a sub-expression of other expressions. There are three types of expressions: numeric, string, and logical.

• **Numeric**
They evaluate to numbers. The following are numeric expressions:

```
4
4 + 5
A / B
A*(B+C/D)
A + sqrt(B)
```

• **String**
They evaluate to strings. The following are string expressions:

```
answer="Yes";
concat(FIRST_NAME, " ", LAST_NAME);
edit("ZZZZ9", A + B);
```

• **Logical**
Logical expressions (conditions) evaluate to **true** (1) or **false** (0). The following are logical conditions:

```
KIDS > 5
SEX = 2 and AGE > 12
```

Substring Expressions

Format

```
string_variable[start_index:string_length]
```

Description

A substring expression lets you extract a part (substring) of a string. The ***start_index*** gives the starting character position of the substring within the string, and ***string_length*** gives the number of characters to include in the substring, including the starting character. If ***string_length*** is not given, then it is assumed to be to the end of the originating string. A negative ***start_index*** leads to the basing of the substring from the end of the string.

Example 1

Suppose the variable `STR` has the value `"ABCDEF"`:

```
STR[1]      "ABCDEF"
STR[3:1]    "C"
STR[3]      "CDEF"
STR[2:3]    "BCD"
STR[5]      "EF"
STR[4:7]    "DEF"
STR[-2]     "EF"
STR[-4:2]   "CD"
```

Both ***start_index*** and ***string_length*** can be numeric expressions as well as constants. For example, to obtain the last 3 characters of `STR` you could use the expression:

```
STR[length(STR) - 2:3]
```

In this example, if `STR` is not at least two characters long, you may get unexpected results. You could also write the above as:

```
STR[-3]
```

Example 2

Likewise, substring expressions can be performed on string arrays. Suppose the string array `crop` had the following definition:

```
PROC GLOBAL

array string crop(5); // 5 crop names

PROC MY_PROGRAM

preproc

    crop(1) = "maize";
    crop(2) = "wheat";
    crop(3) = "rice";
    crop(4) = "potatoes";
    crop(5) = "legumes";
```

The following substring expressions would yield the results as shown:

```
crop(1)[2]      "aize"
crop(1)[3:1]    "i"
crop(2)[3]      "eat"
crop(3)[2]      "ice"
crop(4)[5]      "toes"
crop(5)[1:3]    "leg"
```

Boolean Values

There are two numeric values in the CSPro language that translate to boolean values as used in other programming languages. The value `true` is equivalent to 1 and the value `false` is equivalent to 0. Note that you need to be cautious when using `true` in conditional expressions, as a conditional expression is true if it is neither zero nor special, but comparing something with `true` is true only if the compared value is 1. That is:

```
numeric value = 2;

if value then
    errmsg("This will be displayed");
endif;

if value = true then
    errmsg("This will not be displayed");
endif;
```

See also: Special Values

Special Values

There are four special values in the CSPro language: `missing`, `refused`, `notappl`, and `default`. A particular value of a data item can be mapped to one of these special values in the data dictionary. They have the following meaning and uses:

Missing

The value `missing` indicates that a data item was supposed to have a response and no response was given. Other terms for this are "not stated" and "non-response." To properly utilize this special value, you must create a value set for this item in the dictionary, setting one of the value set entries to the special value "Missing." For example, you could set 8 (or 88, 888, etc.) or 9 (or 99, 999, etc.) to `missing`. Finally, although you must associate a number with the special value missing, you can only use the = or <> comparison operators against the special value `missing`; i.e., you cannot refer to the numeric value you assigned it to in your dictionary value set.

Refused

The value `refused` is similar to `missing` but has several special attributes regarding how it is handled during data collection. The value indicates that a data item was supposed to have a response but the respondent refused to provide an answer. To properly utilize this special value, you must create a value set for this item in the dictionary, setting one of the value set entries to the special value "Refused."

Notappl

The value `notappl` indicates that a data item is blank. The item did not have a response because the question did not apply to this respondent. Fields that are skipped during data entry are assigned the value `notappl`. You can map this value to a number or blanks in the dictionary using the special value "NotAppl," but generally it is best to keep this value defined as blanks.

Default

The value `default` indicates that a data item or variable has an undefined value. This can result from various circumstances. For example, an invalid calculation in logic (such as a divide-by-zero error) will return the result `default`. Reading the value of a variable from a data file when the data type specified in the dictionary for the variable is numeric and the value in the data file contains non-numeric characters will cause the variable to become `default`. The same result occurs if the dictionary does not specify a decimal character for the variable being read but the value in the data file contains one. Additionally, if a numeric variable has numeric subitems, some of which are `notappl` (blank) then the value of the variable will sometimes be `default`. For example if the numeric variable `DATE` has 3 subitems `DAY=1`, `MONTH=notappl`, and `YEAR=2020`, then the value of `DATE` will be `default`. The value `default` can also be written to the data file when the value of the variable being written overflows the length specified for that variable in the dictionary (for example if the value of the variable is 999 but the dictionary specifies a length of 2 for the variable). You can map this value to a number in the dictionary using the special value "Default," but this should generally be avoided.

See also: Special Function, Refused Value, Boolean Values

Refused Value

The value `refused` is a special value that has been defined in the value set and is treated in several special ways by CSPro and CSEntry.

Displaying Refused Values in Response Lists

During data collection, CSEntry will show refused values in capture types by default, just as it displays other values defined in a value set. However, to make it harder for an operator to select a refused value, it is possible to turn off the default displaying of refusals. If turned off, the operator must select CSEntry's Show Refusal Options menu option to see

any defined refused values.

Using Refused in Logic

As with other special values, the number defined as refused cannot be used in logic comparisons. For example, if TRIBE has the value -99 defined as refused, then in logic:

```
if TRIBE = -99 then // does not work

if TRIBE = refused then // works
```

Overriding the Selection of Refusals

Advanced users may want to have more control over when a refused value can be entered. If a function is defined in logic with the name `OnRefused`, CSEntry will call this function every time a refused value is selected. If the function returns `0` (`false`), then the function will prevent the entry of that refusal. For example:

```
function OnRefused()

    string probe_text = maketext("Are you sure that you want to select Refused for %s? "
                                 "Remember the approaches discussed in training about "
                                 "trying to get answers for all questions.", getlabel());

    when warning("%s", probe_text) select("Yes", continue, "No", continue) default(1);
         1 -> exit true;
           -> exit false;
    endwhen;

end;
```

See also: Special Values

Operators

Operators

CSPro logic has a variety of arithmetic, relational, and logical operators. When more than one operator exists in an expression, the order in which the operators are evaluated is determined by their precedence.

Arithmetic Operators

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo (remainder)	%
Exponentiation	^

The arithmetic operators work on numeric expressions, though the addition operator can also be used to perform string concatenation.

Relational Operators

Operation	Symbol
Equal to	=
Not equal to	<>
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
In range	`in`
Has range (for repeating items)	`has`

The relational operators work on numeric and string expressions.

Logical Operators

Operation	Symbol	Keyword
Negation	!	`not`
Conjunction	&	`and`
Disjunction	|	`or`
If and only if	<=>	

With logical operators, either the symbol or the keyword can be used.

See also: Operator Precedence, And/Or Truth Table

In Operator

Description

The `in` operator is used in logical expressions to test whether an item or variable is within a set of values or ranges. The item or variable can be a number or a string. A range of values is separated by a colon, for example `1:5`. Elements of a list of values or ranges are separated by commas, for example `1,3:5,7`. You can also use `special` to mean all special values (as in the `special` function). The `in` operator can also be used to test whether a value is in several CSPro objects:

- A list can be used to determine if the value is located in the list's values (as in the `list.seek` function).
- A value set can also be used to test whether a value is in the value set's codes (as in the `invalueset` function).

Example 1

```
if RELATIONSHIP in 1:5 then
// is the same as...
if RELATIONSHIP >= 1 and RELATIONSHIP <= 5 then
```

Example 2

```
if WORK in 1, 3, 5 then
// is the same as...
if WORK = 1 or WORK = 3 or WORK = 5 then
```

Example 3

```
if X in 1:4, missing, refused then
// is the same as...
if ( X >= 1 and X <= 4 ) or X = missing or X = refused then
```

Example 4

```
if NAME in "A":"MZZ" then
// is the same as...
if NAME >= "A" and NAME <= "MZZ" then
```

Example 5

```
if AGE in AGE_TEENAGE_VS, special then
// is the same as...
if invalueset(AGE, AGE_TEENAGE_VS) or special(AGE) then
```

See also: Has Operator

Has Operator

Description

The `has` operator is used in logical expressions to test whether a repeating item is within a set of values or ranges. The item can be numeric or alphanumeric. A range of values is separated by a colon, for example `1:5`. Elements of a list of values or ranges are separated by commas, for example `1, 3:5, 7`.

This function is similar to the `in` operator except that it works on repeating items. It thus tests whether a group of items contains certain values.

Example

```
// assuming that there are five people listed on a record...

if SEX has 2 then    // means: are any of the five people women?

if AGE has 0:17 then // means: are any of the five people under the age of 18?
```

See also: In Operator, Count Function, Seek Function

If and Only If Operator

This operator has the following truth table:

if and only if (X <=> Y)

	Y	
X	**true**	**false**
true	true	false
false	false	true

The following two sets of code give the same result:

```
if ( SEX = 2 and AGE >= 10 ) <=> ( CHILDREN_BORN <> notappl ) then
    errmsg("Children ever born incorrect");
endif;
```

and:

```
if SEX = 2 and AGE >= 10 then

    if CHILDREN_BORN = notappl then
        errmsg("Children ever born incorrect");
    endif;

else
    if CHILDREN_BORN <> notappl then
        errmsg("Children ever born incorrect");
    endif;

endif;
```

See also: Operators, Operator Precedence

Operator Precedence

The table below shows the order of precedence for operators. When operators of the same precedence are in an expression, they are evaluated from left to right. The order of precedence can be changed using parentheses. Operators in parentheses are evaluated first.

Order	Operators
1	^
2	* / %
3	+ -
4	= < > <= >= <> has in
5	not !
6	and &
7	or |

And/Or Truth Table

The truth table summarizes all possible evaluations when two expressions (X and Y) joined by an operator (`and` or `or`) are true, false.

and (X and Y)

	Y	
X	**true**	**false**
true	true	false
false	false	false

or (X or Y)

	Y	
X	**true**	**false**
true	true	true
false	true	false

Files

External Files

An external file is a text file other than the primary data file that you can use in a data entry or batch application. You can read and/or write to external files, using CSPro logic. You must create a data dictionary that describes the format of any external file you want to use. An external file dictionary can contain only one level.

You can share external files across a network. If an external file is accessed only by read functions (loadcase, locate, find, key, retrieve), no special programming actions need to be taken to share the file. Multiple users can read the file at any time.

However, if an external file is accessed by any write functions (writecase or delcase) only one user at a time may use the file. For write functions, the external file is like a file in a filing cabinet. When one person has taken out the file for use, no one else can use the file until the person has returned it.

You can control when the file is in use by coding open and close functions. The file is in use between the execution of the **open** and the **close** function. This gives you complete control over when the file is in use. You should try to minimize the time the file is in use in order to allow other users to access the file.

If **open** and **close** functions are not coded for an external file used for writing, the following "open" and "close" rules apply:

- In batch processing, the file is opened at the beginning of the run and closed at the end.

- In data entry processing, the file is opened just before any external file function is executed and is closed immediately following the function, unless one of the following functions is used on the file:

 - **loadcase** without a *var-list*
 - **retrieve**
 - **key**

When any of the above functions is used, the file is opened just before the first file function is executed, but is left open after the function is completed. These functions depend on remembering the current position of the file. If the file is closed, the current position is lost.

See also: Insert or Drop a File from an Application, Lookup Files

Lookup Files

A lookup file (external file) is a text file that can be used in a data entry or batch application from which you retrieve data to display on a form or to use in a calculation. It requires a CSPro data dictionary. Possibilities include:

- Geographic codes and names. Your application could show the name corresponding to the code the user keyed.

- Industry and occupation codes. Your application could ensure the user keys a valid code.

- Last year's data. Your application could look up a corresponding field from last year's data and calculate a percentage change.

- Generalized menu choices. Your application could read a lookup file and show the contents on the screen as a menu, then convert the user's choice to a code.

To use a lookup file (external file) in your application, do the following:

- Create the lookup file and its data dictionary
- Close the lookup file's data dictionary
- Create a data entry or batch edit application with a standard forms file and data dictionary.
- Insert the lookup file's data dictionary into the application.
- Add logic to the application to manipulate the lookup file. The Loadcase and Selcase functions are particularly useful

Note: The CSPro examples include an application that demonstrates the use of a lookup file. This is normally installed in a subfolder of the Documents folder, "CSPro\Examples 7.7."

Working Storage File

"Working storage" contains alphanumeric variables and data items used in an application and which are not part of any data file. Definitions of working storage variables and data items are contained in a data dictionary which is not connected to any data file. This dictionary can have any number of records but can have only one level.

See also: Add Files to an Application

Miscellaneous

Display Dimensions

When working with "maximum display" width and height figures in CSPro, values are returned that may not correspond exactly to the pixels of the display:

Windows: The value returned depends on the current scaling, but the maximum display dimensions returned account for 90% of the display size after accounting for the size of the Windows taskbar.

Android: The unit returned is density-independent pixels: "an abstract unit that is based on the physical density of the screen."

When using display dimensions, you will generally want to use a scaled value for the width and height rather than using fixed values. For example, to view a webpage at 75% of the maximum display size, you could code:

```
view("https://www.csprousers.org/forum",
     width := tonumber(getproperty("MaxDisplayWidth")) * 0.75,
     height := tonumber(getproperty("MaxDisplayHeight")) * 0.75);
```

See also: GetProperty Function, JavaScript Interface

Data Entry Module

Introduction to Data Entry

The Data Entry module allows you to create, using a single dictionary, one or more forms [screens] for data entry. You may also specify the data entry behavior and incorporate logic in a program to check for consistency between variables and to set up skip patterns. After you have developed the forms and the program to your satisfaction, use CSEntry to input the data.

You may enter the data in the office after the information is collected or you might want to use the Computer Assisted Personal Interviewing (CAPI) feature, in which the interviewer uses a laptop computer to enter responses in the field as they occur.
This section contains the following information:

General Data Entry Concepts
CSPro Data Entry Concepts
Create a Data Entry Application
Change Data Entry Characteristics

Data Entry Application

General Data Entry Concepts

Data Entry Philosophies

There are two different approaches to data keying:

Heads-Down Keying

This approach is most commonly used for keying of census forms because of the large volumes of data involved. While entering data, the operator generally does not look at the computer screen, but rather, looks down at the questionnaire on the table or work surface. The objective of heads-down keying is to transcribe to the computer, as quickly and accurately as possible, the data as they appear on the questionnaire. On-line checking is generally kept to a minimum and consistency errors are resolved in a later phase, generally through computer edit programs. Operators do not need to be familiar with the subject matter of the questionnaire. They make very few decisions to resolve data errors. The most important skill is speed and accuracy. CSPro provides operator statistics to help measure operator speed and accuracy.

Heads-Up Keying

This approach is most commonly used for entering data from surveys, due to the smaller number and greater complexity of the questionnaires (as compared with a census). While entering data, the operator often refers to the computer screen as well as to the questionnaire. The objective of heads-up keying is to catch and correct as many errors as possible as the data are being entered. As a result, there is generally more on-line checking programmed into the application. Operators need to be very familiar with the subject matter of the questionnaire. They will make decisions to resolve data errors, and must be properly trained to do so.

Skip Issues

To Skip or Not to Skip?

Skipping—causing the cursor to jump over one or more fields during the data entry operation—is an issue that provokes discussion both pro and con. The decision of the application designer to use (or not use) skips will depend entirely on the type of application and the data entry staff.

When the data capture operation is expected to be heads-down, as it would be for a census or similar high-volume application, it will cause less confusion to the keyer if all skips are controlled by the operator rather than the application. There will then be no "surprises" for the keyer when the application logic forces the cursor to one field when the keyer, looking not at the screen but at the form, expects the cursor to be in a different field altogether. Thus, instead of speeding up the entry operation by anticipating probable cursor movement, this tactic eventually slows down the operation when there are inconsis tencies in the data or, simply, keying errors.

When the data capture operation is of smaller volume, or with a more complex questionnaire or forms, it may make sense to use application-controlled skipping. The operator will be more likely to be aware of the cursor movement and less likely to be surprised by unexpected interruptions in the normal sequence. Of course, a combination of operator-controlled and application-controlled skipping may be used in any given application; the designer will have to weigh the keying environment and the forms to be entered to make the appropriate decision.

Manual Skips

During data entry, there may be times when you want the operator to be able to skip over certain fields that do not apply to the current case. For example, in a Population record, the fertility fields do not apply to males or to underage females, nor do questions on economic activity apply to children under a certain age. In CSEntry, the '+' key on the numeric keypad is always active as a "skip" key. Every field has a skip field value associated with it. The default value is "next," meaning that when the cursor is on that field and the skip key is pressed the cursor will move to the next field in sequence. CSEntry allows you to change this value to any later field in the sequence, or to the end of the screen or form. This approach corresponds to the operator-controlled option in CSPro.

Automatic Skips

Automatic, or application-controlled, skips depend on information already keyed to direct the cursor movement. For example, in a household survey, if a female respondent states that she has at least one child living with her, the cursor can skip automatically to the form for capturing information about that child (and any others). Conversely, if the female indicates that no children are present, the cursor can be directed to skip over the information about children. It is clear that such skips depend entirely on the accuracy of the data keyed prior to the skip; if a "Yes" response is mistakenly entered as a "No," the cursor will be misdirected and the operator will find that the screen(s) presented for keying do not correspond to the information in the paper forms. This will cause loss of time as the operator seeks to uncover the error. Automatic skips, when used, must be well-documented so that the keyer is aware of the possibilities of non-sequential cursor movement. This approach corresponds to the system-controlled option in CSPro.

Errors at Data Entry

Errors are introduced into the data through miskeying. Verification (rekeying or double keying) can reduce these errors. A system called **intelligent data entry** may be used to prevent invalid entries from ever getting into the system. An intelligent data entry system ensures that the value for each field or data item is within the permissible range of values for that item. Such a system increases the chance that the data entry operator will key in reasonable data and relieves some of the burden on later stages of the data preparation process.

At data entry time, CSEntry shows a message every time the keyer enters a value that is out of range according to the data dictionary. You may set the attribute to override the message and force the out-of-range value into the data file. If this attribute is not selected, the keyer cannot proceed until a valid value is entered.

See also: Data Entry Philosophies

Adding Logic

In most surveys, consistency errors are corrected manually as opposed to automatically. The correction process, as done traditionally, is often very lengthy, time-consuming and painful. In surveys of small volume and high complexity, such as Household Surveys or Income and Expenditures Surveys, it is often desirable to apply the edit specifications rules at data entry time and resolve any errors immediately while the questionnaire is still at hand. This approach is not recommended for a census.

You can use the CSPro language to write consistency checks for virtually any part of your data entry application's level, form, roster, or field. The logic is executed as the data is being keyed. Any error messages are reported back on the screen, and the operator then has access to both the error messages and the questionnaire itself on the screen. The same logic can be run against the data after they are entered in either batch or interactive mode.

CSPro Data Entry Concepts

Operator vs. System Controlled

CSPro offers two distinct types of data entry applications. Your choice will determine certain behaviors at data entry time. Some special data entry keys will behave differently. For more detail about special data entry key behavior, please refer to the Data Entry User's Guide.

Operator Controlled

This is the default type of data entry application. This type generally allows more flexibility for the keyer during data entry. It is recommended for simple ad-hoc applications and for census applications. Operator-controlled applications have the following features:

- Some special data entry keys are active during data entry.
- CSEntry will **not** keep track of the path.
- "Not applicable" values will be allowed.
- More appropriate to the heads-down methodology.
- Operator can bypass logic in the application using special keys.

System Controlled

These applications generally place more restrictions on the data entry operator. This type is sometimes used for complex survey applications. System controlled applications have the following features:

- Some special data entry keys are not active during data entry.
- CSEntry will keep track of the path.
- "Not applicable" values will not be allowed unless defined in a value set.
- More appropriate to the heads-up and CAPI methodologies.
- Logic in the application is strictly enforced; operator cannot bypass or override.

You set the application type in the Change Data Entry Options dialog box (**Options** -> **Data Entry** from the main menu toolbar).

Data Entry Path

CSPro supports a powerful feature called data entry path. The path can either be "turned on" or "turned off," depending on the data entry application type selected on the Data Entry Options dialog box. Operator controlled applications always have path turned off, while system controlled applications always have path turned on.

Path On

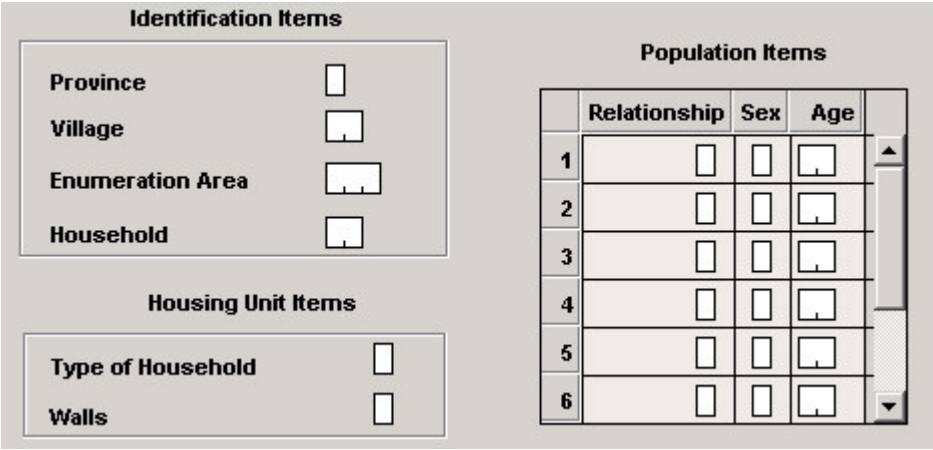
CSEntry will keep track of the order in which the data entry operator entered all fields. If the operator goes backward, the cursor will go to the fields in the reverse order in which they were entered. For example, if the logic causes the cursor to skip over a set of fields, the cursor will also skip over these fields when the operator goes backwards. Fields that were skipped can never be entered, unless the operator goes backwards and chooses different values to avoid the skip. This helps ensure the integrity of the data file.

Path Off

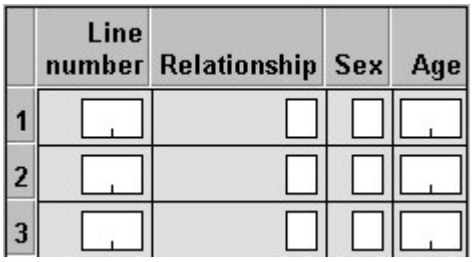
CSEntry will not keep track of the order in which the data entry operator entered the fields. If the operator goes backward, the cursor will go to the preceding field even if it had originally been skipped.

Data Entry Elements

- **Forms**
  A form is a collection of fields, text and/or rosters which appears on the screen at the same time during data entry. A form may be larger than the actual screen. When this is the case, the form will scroll as necessary during data entry so that the current point of entry is always visible to the keyer. A form may repeat if it contains fields from a dictionary record which has more than one occurrence.



- **Rosters**
  A roster is a grid that shows multiple occurrences of a group at the same time. Many questionnaires have rosters printed on them. A typical example would show each person as a row and each column as a variable, as shown below. Rosters can also have a vertical orientation, in which case the rows and columns would be reversed.



In CSPro, you can show repeating groups as a roster on a single form or as individual fields on a form that repeats.

The darker gray area at the top of each column is called a column heading. In the example above, the column headings contain the text "Line number", "Relationship", "Sex", and "Age". The text in the darker gray area to the left of each row is called the "occurrence label." In the example above, the occurrence labels are "1", "2", "3". These are the default values.

In rosters with vertical orientation, column headings and occurrence labels are reversed.

- **Fields**
  Fields are areas of a data entry form that may be keyed or may show values. Fields may be placed directly on the form or may be part of a roster on the form. Fields are always associated with dictionary items. Some properties of fields, such as length and type (numeric or alphanumeric), are defined in the data dictionary. Other properties are defined in the forms designer. In this example we have two fields:

See also: Add a Form, Add a Roster to a Form, Add Fields to a Form

Blocks

Overview

Blocks are a way to group several fields into a related unit. In summary:

- Blocks contain some number of fields that come from the same group.
- On mobile devices, a block's fields can be displayed on the same screen and the operator can enter values into these fields in any order.
- Blocks can have question text. When running a data entry application, this question text will appear above the question text for each field in the block.
- Logic can be defined for the block.

Blocks are primarily useful for two reasons. Firstly, they provide the mechanism for CSEntry to display multiple fields on a screen. Secondly, they allow you to write logic checks in one place that apply to a number of fields.

The use of blocks is entirely optional. Examples of potential uses of blocks include date fields (day, month, year) or consumption fields (quantity, unit). Alternatively, you can lump together related questions in one block for easy viewing on one screen (e.g., fertility counts of children).

Creating Blocks

To create a new block, select multiple fields in the Forms Tree by holding down the **Ctrl** key while selecting fields. Once the fields are selected, right-click and select **Add Block**. A dialog box allows you to specify the block's label, name, and whether or not the fields of the block should appear together on the same screen on mobile devices.

Blocks are shown in the Forms Tree with a red icon.

Only fields can be added to blocks. Blocks can be created from fields on a form or from fields on a roster. To add a field to an existing block, drag the field onto the block. To remove a field from a block, drag it outside of the block. A block can contain no fields, but once it contains one field, it can only contain other items from the same group. For example, if a block contains an item from the population record, then only other items from the population record can be part of the group.

Using Blocks

You can define question text for a block. When running a data entry application on a mobile device, the block's question text appears at the top of the screen, and then each field's individual question text appears underneath. This allows you to have some general text ("What is your date of birth?") and then specific text ("What is your year of birth?"). On Windows desktop, the block and field question text appears in the same window.

You can write logic that will run when the block is entered or exited. For example, if you have a block with two fields, FIELD1 and FIELD2, then logic will be executed in the following order if moving forward in your program:

BLOCK (`preproc`, `onfocus`)
FIELD1 (`preproc`, `onfocus`, `killfocus`, `postproc`)
FIELD2 (`preproc`, `onfocus`, `killfocus`, `postproc`)
BLOCK (`killfocus`, `postproc`)

Instead of putting logic in the field procedures, you will generally place all logic related to the fields of the block in the block's procedures.

You can refer to blocks in logic in the movement statements (`advance`, `ask`, `move`, `reenter`, `skip`) and you can use the blocks in occurrence-related functions (`count`, `curocc`, `maxocc`, `noccurs`, and `totocc`).

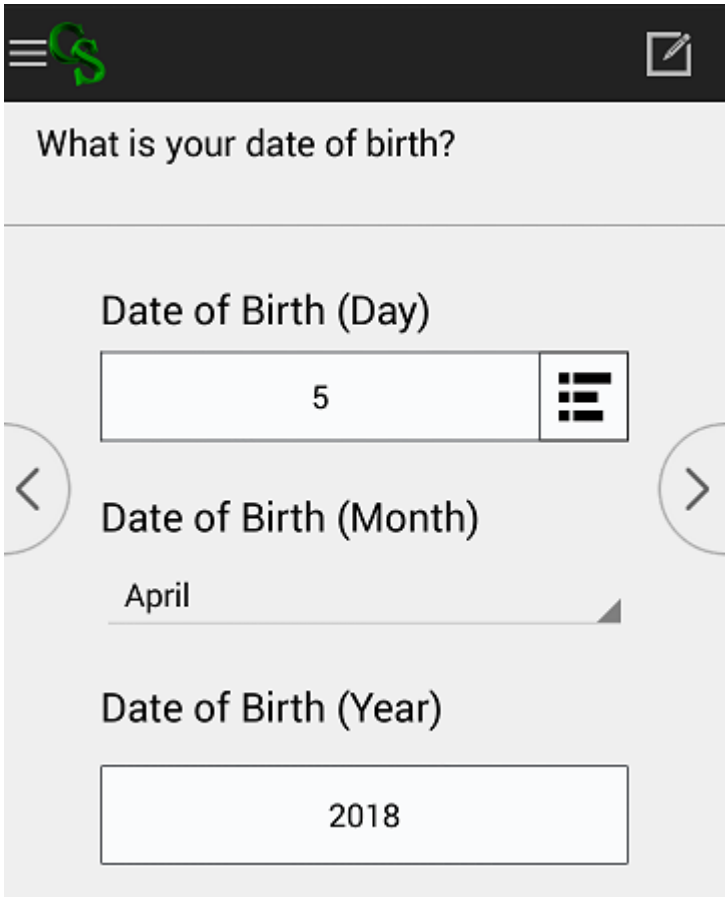
Just as you will generally want to put logic in the block procedures, you will likely want to use the block name in movement statements rather than a field name. One advantage of this is that you can modify or reorder the fields in the block without having to modify movement logic. When using `ask` on a block, if the condition fails, then all of the fields in the block will be skipped.

"No Field" Blocks

A block can exist without any fields added to it. The logic procedures associated with such a block are still executed based on where the block is in the Forms Tree. These blocks can be useful as "control fields," serving as an anchor upon which you can orient your skips.

Example

Thinking about the example in the above image, date of birth (day, month, year), one disadvantage of the one-question-per-screen approach on mobile devices is that it splits related fields onto multiple screens, and, depending on where logic checks are executed, results in a reenter potentially not going to the most relevant field. With blocks, all related fields appear on the screen at the same time, allowing the enumerator to enter the values in any order.



Only after the enumerator has filled all applicable values will they move past the block. Logic checks for date of birth would thus appear in the block's procedure:

```
PROC DOB_BLOCK

preproc

    // only ask date of birth in conventional households
    ask if HH_TYPE = 1;

postproc

    // if the month or day is missing, then assume 1 for these checks
    numeric dob_mm = DOB_MONTH;

    if dob_mm = missing then
        dob_mm = 1;
    endif;

    numeric dob_dd = DOB_DAY;

    if dob_dd = missing then
        dob_dd = 1;
    endif;

    numeric dob_yyyymmdd = DOB_YEAR * 10000 + dob_mm * 100 + dob_dd;

    if not datevalid(dob_yyyymmdd) then
        errmsg("The date of birth is not valid");
        reenter;

    elseif dob_yyyymmdd > sysdate("YYYYMMDD") then
        errmsg("The date of birth cannot be in the future");
        reenter;

    endif;
```

See also: Change Block Properties

Issues to Consider When Designing a Form

If you plan to key data from paper questionnaires you generally try to make the forms [screens] match the pages in the questionnaire. CSPro provides flexibility in the way you define forms, the number of forms to use, and the choice of fields to go on each form. There are, of course, limitations imposed by the structure of the data dictionary, some of which have to do with whether records and items are "multiple":

- A record is considered multiple if it is defined as "Max > 1" in the data dictionary.
- An item is considered multiple if it is defined as "Occ > 1" in the data dictionary.
- A sub-item is considered multiple if it has been defined as "Occ > 1" in the data dictionary or if the item it belongs to is defined as "Occ > 1".

Keep in mind the following rules when you design your data entry forms:

- You **can** mix items from different single records on the same form.
- You **can** mix ID items with items from single records on the same form.
- You **can** split items from the same record onto different forms.
- You **can** make more than one roster from a multiple record. The rosters can be on the same form or on different forms.
- You **can** mix items from a single and a multiple record on the same form, but the latter must be in a roster.
- You **cannot** mix items from different multiple records on the same form.

• You **cannot** mix items from different levels on the same form (applies to complex data dictionaries only)

If you have any multiple records, items, or sub-items in the data dictionary, you must decide whether you want to make them into a roster or use a form that repeats. You must take this into account when deciding which items to place on each form.

Cases and Levels
A **case** is the primary unit of data in the data file. A case usually corresponds to a questionnaire, although some complex applications may have a hierarchical set of questionnaires that comprise a single case. For example, the main questionnaire may consist of a household roster and other household information, and there may also be a separate questionnaire for each woman in the household.

The data entry application may then contain two levels —one for the household and one for each woman in the household. The set of forms corresponding to the household make up level one. The set of forms corresponding to each woman make up level two. Each case would consist of a level one and a variable number of level **occurrences** for level two. Most applications consist of a single level.

Tips on Reviewing Data Entry Applications

Here are some tips on reviewing a data entry application, including advice on best practices and warnings about common mistakes that people make when creating data entry applications.

Data Dictionary

1. All dictionaries that make up your application should use the same zero fill and decimal character settings. You can check this using Dictionary Analysis.
2. Every numeric item should have a value set defining the range of values that can be entered into the field. A common mistake is to not declare a value set for a two-digit age field, but without a value set, ages like -5 can be entered. Although protected fields do not need value sets (because the operator will not enter a value into the field), it is still a good idea to add a value set so that this information about possible values is available to other users of the dictionary. You can use Dictionary Analysis to identify items without a value set.
3. For items with multiple value sets, make sure that the primary value set (the first one showing in the dictionary editor) is the value set with the complete list of codes. This value set is used to show labels in the case tree. If the item's value set is defined by cascading options, consider adding a generic value set as the primary value set rather than having one that potentially leads to incorrect labels based on the cascading conditions.

Form Design

1. Ensure that every field's capture type is set to match the way that you want an operator to enter data for the field. As much as possible, prevent an operator from having to enter codes. For numeric fields, text boxes should generally only be used for values like age or currencies; you can use combo boxes for instances when an operator typically will type a value but can occasionally select a discrete code.
2. Check for the consistency of field properties. For example, if the upper case setting is used for some fields, it should generally be used for all fields. Field properties can be standardized using the Field Properties (for Multiple Fields) dialog.
3. All protected fields must be prefilled with a valid value; if not, CSEntry will terminate data collection. Make sure that all protected fields are filled in via logic or by using PFF parameters.
4. If an operator is never going to collect data directly for a given field, consider leaving it off the form. Items in a dictionary can be filled in using logic without needing to exist on a form. Alternatively, you can hide such fields from an operator using the Hide in Case Tree field property.

Question Text

1. Make sure that every question has defined question text. Even for protected fields, it is a good idea, for completeness, to define question text.
2. For multiply occurring questions, ensure that the minimum and maximum occurrences defined for a field's question text cover the whole range of occurrences. You can use the Harmonize Occurrences functionality to

ensure that the maximum occurrence values match the current dictionary settings.

Logic

1. Ensure that there is logic to check for acceptable values for fields without value sets. For example, you may want to ensure that operators enter proper alphanumeric responses for fields such as names and other/specifies.
2. For alphanumeric text box and checkbox fields, you may want to ensure that a certain number of characters is entered or choices are selected.
3. Review all hard checks and soft checks and think about what data respondents will be able to provide. You do not want operators to enter invalid data to bypass a hard check, so use soft checks for questions where some flexibility is allowable.
4. When working with dynamic value sets or question text fills that are set via logic, make sure that any logic that is executed for such tasks on a multiply occurring field is in an `onfocus`, not `preproc`, event. If the logic is in the `preproc`, an error can occur as an operator moves backwards to a previous occurrence because the value set and question text will not be updated (because `preproc` events only occur as an operator moves forwards).
5. Verify that fields that should have values are not skipped over. When using system-controlled mode, the values in skipped fields are deleted when a case is finalized. A common mistake is to prefill a field and then skip over it, which results in the data being lost when the case is finalized. In such instances, the field should be prefilled and then protected, which will prevent the operator from modifying the value but ensuring that the data is maintained.
6. A best practice is to minimize the use of unnamed constants throughout the application. For example, instead of coding `2020` as a year, declare `numeric CensusYear = 2020;` and then use `CensusYear` throughout the application. This makes the code clearer and also makes it easier to change your application when circumstances change.

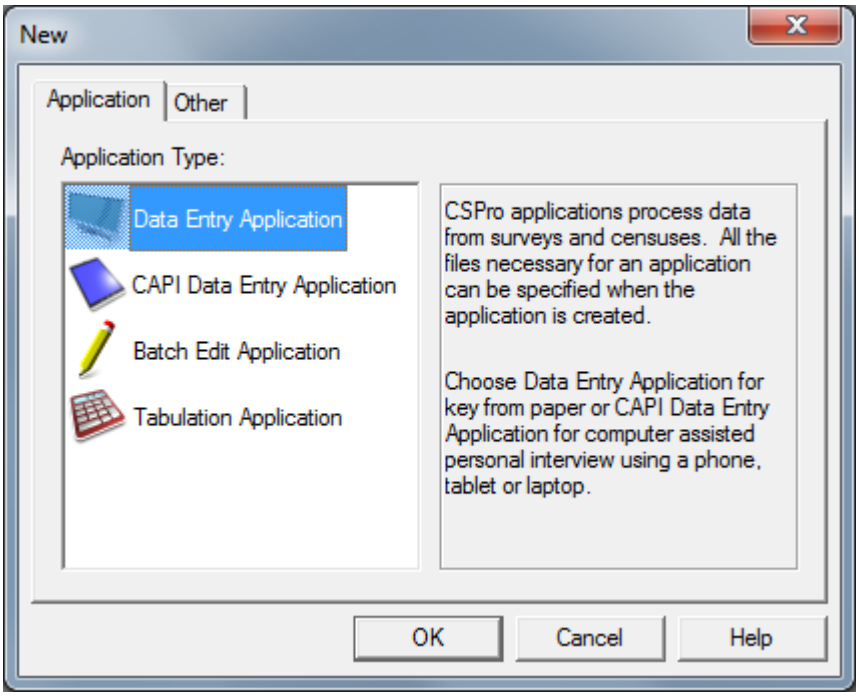
See also: Programming Standards

Create a Data Entry Application

Create a New Data Entry Application

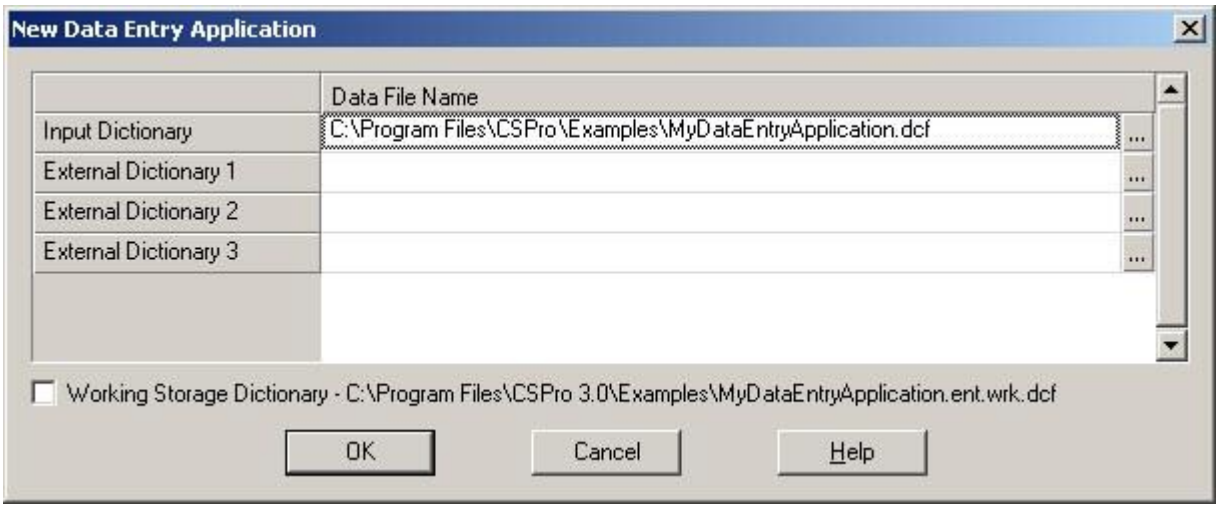
To create a new data entry application:

• Click on the toolbar, or from the **File** menu, select **New**. The following dialog box will appear.



• Select **Data Entry Application** or **CAPI Data Entry Application** and press **OK**. Both of these options create data entry applications, but selecting CAPI sets some options that will be more convenient for creating CAPI applications. (These options include: turning on CAPI mode, running in system controlled mode, and not asking for the operator ID.)

• A file dialog box will appear. Enter the name of the application file. Make sure you are located in the folder where you want to place the application files. Then press **Create**. The following dialog box will appear.

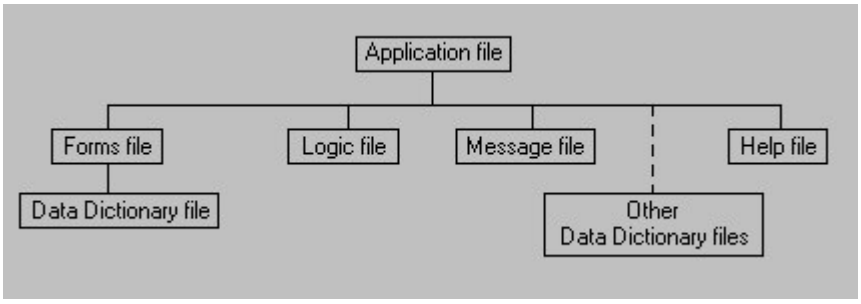


• A default name of the data dictionary describing the data enter file is given. You can use this name or change it. If you give the name of a dictionary file that already exists, that data dictionary will be used by the application. If you give the name of a dictionary that does not exist, a new data dictionary will be created.

- If you are using an existing CSPro data dictionary, then the system displays the question: "Would you like CSPro to create a set of forms for you, based on the input dictionary? If you answer "Yes" CSPro will automatically generate default data entry forms and will display the the drag option menu. If you answer "No" you may begin creating data entry forms.

- If you are creating a new CSPro data dictionary, you will need to enter information into the dictionary about records, items, and values before you can create forms.

Data entry applications consist of the following files:



- Data Entry Application File (.ent)
  This specifies all other files contained in the application and includes other application information.

- Form File (.fmf)
  There is usually one forms file per application, but there may be multiple forms files. Each forms file contains one Data Dictionary file (.dcf) which represents the primary data file that is being created or modified.

- Logic File (.apc)
  Contains CSPro language statements.

- Message File (.mgf)
  Optional file, it contains text for messages displayed during data entry.

- Question File (.qsf)
  Optional, contains text for CAPI text and help screens displayed during data entry.

- Other Data Dictionary Files (.dcf)
  Optional, it represents secondary data files (such as lookup files) that are read and/or written to during data entry.

Generate Default Data Entry Forms

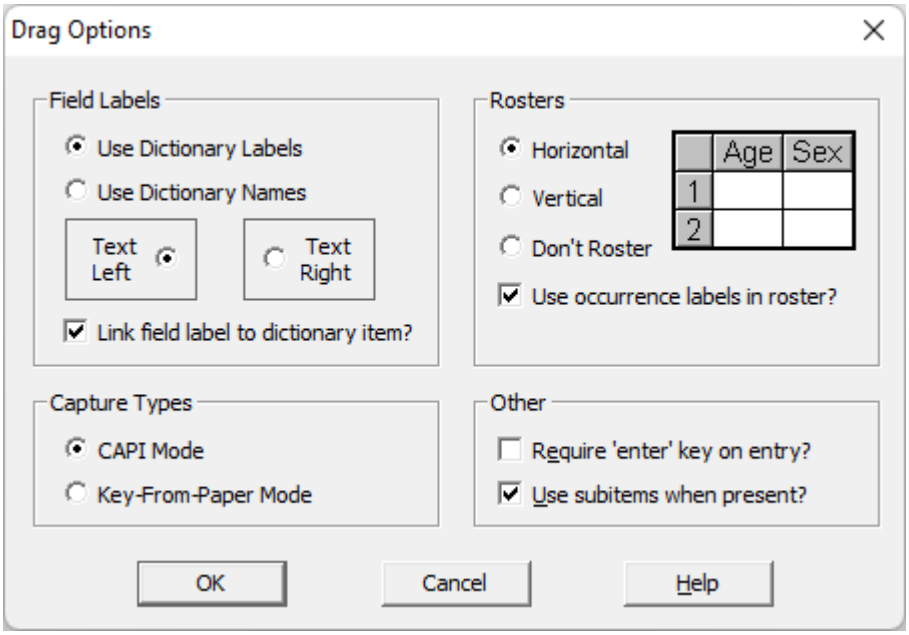
CSPro can automatically generate data entry forms that places all dictionary items onto forms. This can save time as it quickly builds up your form(s), allowing you to easily customize them to your specific needs. One form will be created for each dictionary record (identification items get their own form as well). Thus, for a one-level dictionary that contains at least one level ID and three other records, you will end up with four forms.

To generate new data entry forms, either press **Ctrl+G**; or, from the **Edit** menu select **Generate Forms**; or from the dictionary tab on the left side of your screen, drag the dictionary onto a form. As this action will destroy all existing forms, a warning message will appear, asking you to confirm that you wish to proceed.

If you choose to proceed, the Drag Option dialog will appear. At this point you have the opportunity to decide text placement with respect to the data entry boxes; whether you want to roster items (when possible); whether you want subitems dropped instead of the item, etc.

Drag Options Menu

Whenever you automatically generate data entry forms, drag an entire dictionary , or drag a dictionary record onto a form, this dialog box will appear. When you drag an individual dictionary item to a form, this dialog will not appear, but the settings in effect will be used. To access this dialog box without dragging, select **Drag** from the **Options** menu.



The following choices are available to customize your drag-and-drop operation:

Field Label Options

When fields are dragged onto a form from the dictionary, the dictionary text associated with the item is usually also included. You can select whether the item's label or the item's name is dragged onto the form.

You can also select whether the text is placed to the left or to the right of the data entry box. (This setting has no effect if the item is rostered.)

If you select to use the item labels, you can link the field label to the item. That means that if you change the item's label in the dictionary, then the label on the form will automatically adjust.

Roster Options

This affects dictionary records and items with more than one occurrence. To enter this type of data, you either need a form that repeats (to allow for the multiple occurrences of the data), or you need a roster.

If you choose **Horizontal**, CSPro will make rosters in which the occurrences are the rows and the fields are the columns. In CSEntry the cursor will move from left to right by default.

If you choose **Vertical**, CSPro will make rosters in which the occurrences are the columns and the fields are the rows. In CSEntry the cursor will move from top to bottom.

If you choose **Don't Roster**, CSPro will make forms that repeat.

If you select **Use occurrence labels in roster**, the roster's rows, instead of being labeled 1, 2, 3, etc., will use the occurrence labels defined for the item or record in the dictionary.

Capture Type Options

When **CAPI Mode** is selected, if the item has a valid value set that can be can be represented by a capture type other than a text box, the item will be added to the form with a preselected capture type. The capture type selected depends on the type and length of the item but can be changed once the item is on the form.

If **Key-From-Paper Mode** is selected, the capture type will default to a text box.

Require Enter Key on Entry

This option determines whether the **Enter** key must be pressed to advance an operator to the next data entry field.

If left unchecked, the cursor will automatically advance to the next field as soon as the maximum number of characters are entered for the field (that is, if the field length is two, then after entering two characters the cursor will advance to the next field). An operator can always hit the **Enter** key to complete a field without having entered the full complement of digits.

If this option is checked, the operator must always press the **Enter** key to advance to the next field.

Use Subitems When Present

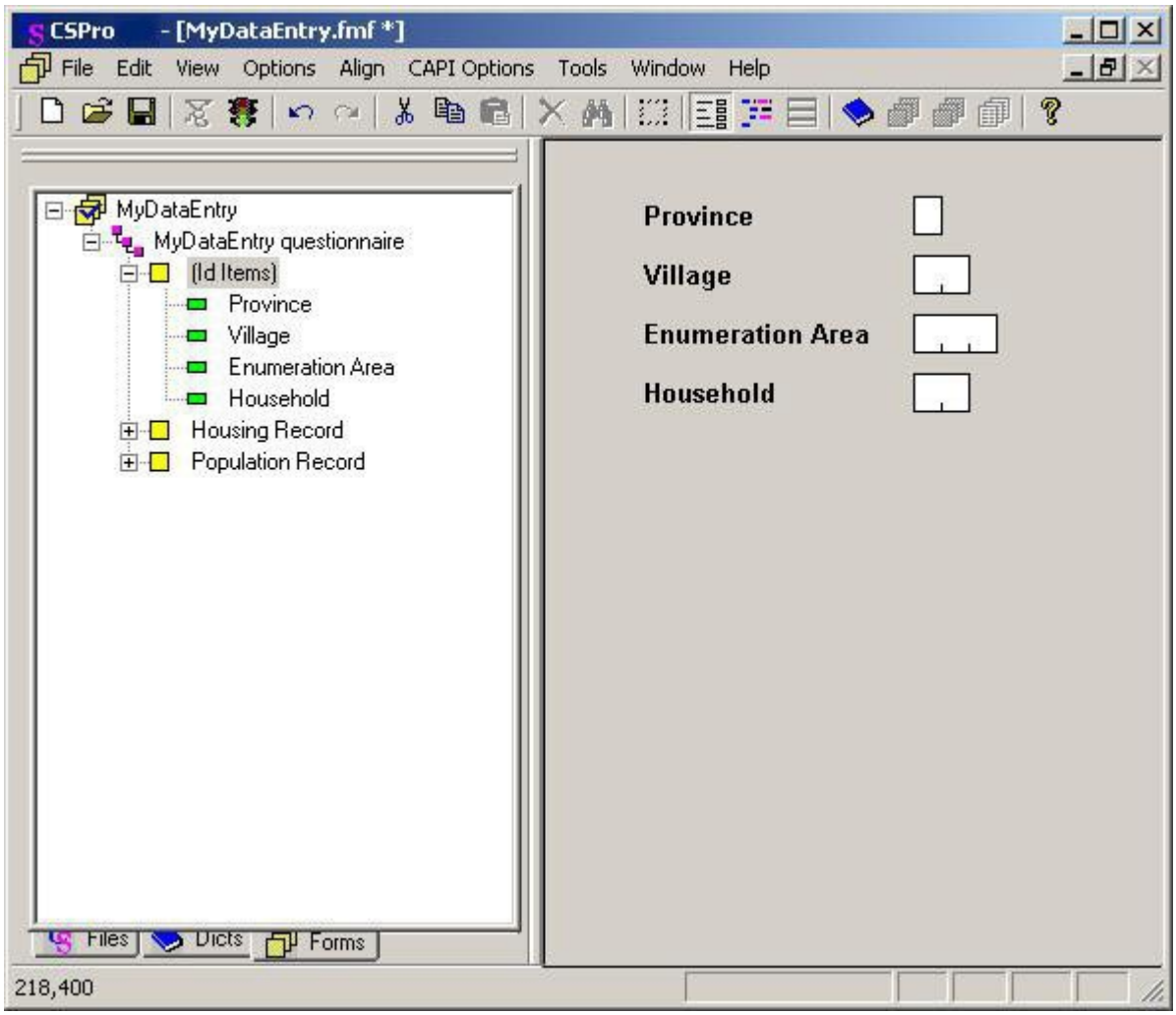
If you have items with subitems, you may check this box to place the subitems, instead of the item, on the form. For example, if you have a `DATE` item that contained the three subitems `DAY`, `MONTH`, and `YEAR`, the subitems, rather than the item `DATE`, would be placed on the form. However, if any of the subitems overlap, the item will be used instead. (This setting has no effect if no subitems are present.)

If this box is left unchecked, items will always be used.

Data Entry Forms Screen Layout

The CSPro window is split in half. The left side contains the data entry tree and three tabs at the bottom. The two tabs of interest when designing forms are the **Dict** [Dictionary] and **Form** tabs at the bottom. By pressing either one you will be able to view the dictionary tree or the data entry tree on the left side.

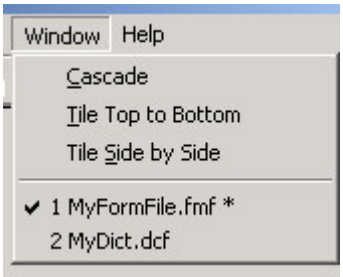
If you generated the screens by default, then the system will display the data entry tree and the Form tab on the left, and the first form created on the right window.



If you opted for creating the forms yourself then the system will display the data dictionary tree and the Dict tab on the left, and a blank form on the right. You are now ready to start dragging items and/or records from the dictionary to the form(s).

If you did not have a data dictionary, then the system will display the data dictionary tree and the Dict tab on the left, and the dictionary window on the right.

Use the **Window** menu to display a list of currently open files at the bottom of the **Window** menu. A check mark appears in front of the name of the file in the active window. Activate a window by choosing the name of its file from this list.

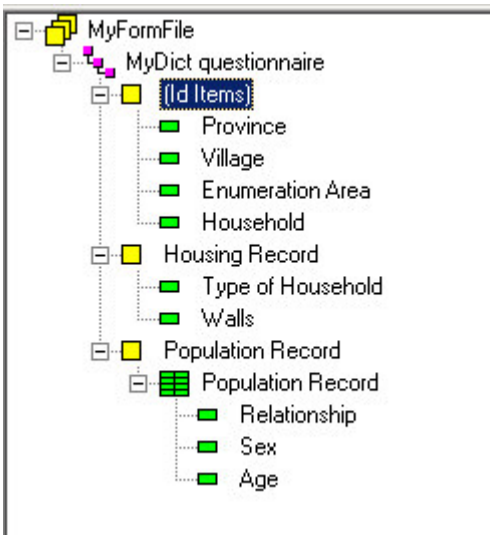


You can also switch between the forms and dictionary window by pressing either or in the Toolbar.

Forms Tree

If you choose to generate the default data entry screens, CSPro will create one screen for the identification items, and a separate screen for each record defined in the dictionary. Press the "Forms" tab on the bottom left of the screen to see the data entry tree. The data entry tree will be identical to the dictionary tree; that is, the items will be listed as named and ordered in the dictionary.

However, if you design the forms yourself, you might decide to include more than one record in one screen or combine fields from different records in one screen. In that case the data entry tree will not be identical to the dictionary tree. In any case, the data entry tree has the following items:



- **Forms File:**
  This is the highest level node, i.e., the root node. It is the owner of all code, which is to say [1] level-, record-, and item-related code, [2] user-defined functions, and the [3] global routine.

- **Level:**
  This is the second-tier tree node, just below the root. It has a 1-to-1 correspondence with the same-named dictionary level.

• **Form:**
This is the third-tier tree node, just below its level. It represents the form and all the items in that form.

• **Roster:**
It represents the roster and all the items included in the roster.

• **Field:**
This is the terminal or "leaf"-node; i.e., the lowest accessible level. It has a 1-to-1 correspondence with a dictionary item.

You are free to rename any of the above the unique names via the properties dialog box, but it is recommended that you retain the original name, so that it is easier for you to see which dictionary entity is being referenced. The data entry tree represents the order in which the data entry is keyed.

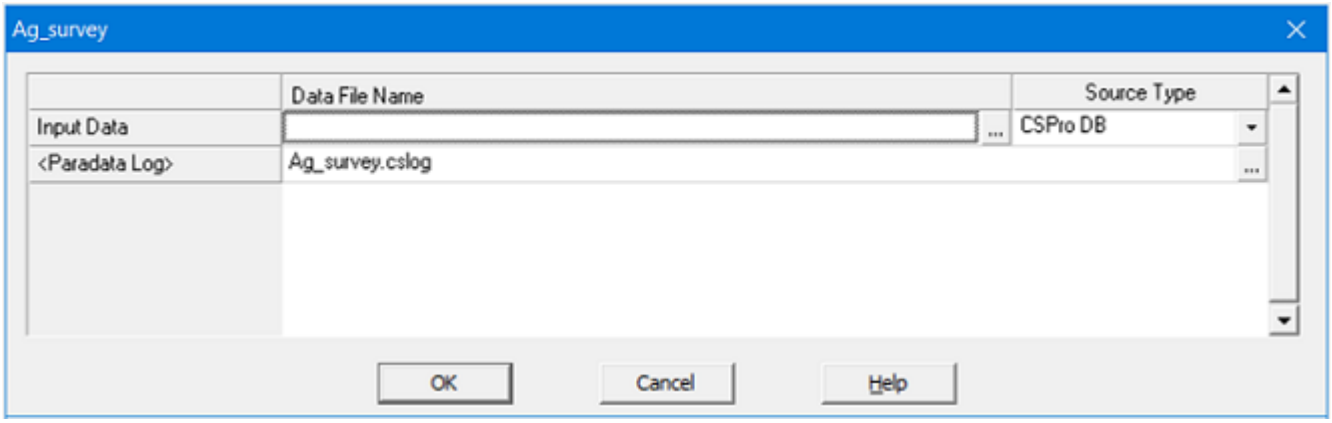
You can change the order of entry by reorganizing the forms or items within the forms by dragging them within the Data Entry tree view. When the operator enters the data, the cursor will follow the order in the tree not the order in the form. When selecting a new edit item, the contents of the logic view will change to display the logic for the selected entity.

Pressing **Ctrl+T** in the data entry tree will allow you to switch between the labels and the names of the items.

Run a Data Entry Application

After you have created your forms, you can run the application to start entering data in CSEntry.

Press ; or press **Ctrl+R**; or select **Run** from the **File** menu to launch CSEntry. You will see a screen that looks like this:



The first row, **Input Data File**, lists the file name where entered data will be saved. If you want to select an existing file, you can click on the ... button to select it. Otherwise, you must give the name of the file and the desired data source (generally either a CSPro DB file or a text file).

In addition to this row, there may be rows where you can specify the names of lookup files, external files, and the paradata log.

After specifying the file names, CSEntry may ask for the operator's ID. CSEntry keeps track of the operator's keying record and can display the operator statistics in CSEntry. You can turn off this option in the data entry options.

You will, of course, want to test the behavior of your data entry application before using it in a production environment, so this is just a quick way to launch CSEntry. When your application is ready for production, you can launch CSEntry independently of CSPro. For assistance with this, see Run Production Data Entry.

Run Production Data Entry

You can customize CSEntry's behavior by creating a PFF file. You can then use the PFF file as a command line parameter for CSEntry.exe. For example, if you name your PFF file MySurvey.pff, then you run the data entry application by invoking:

```
"C:\Program Files (x86)\CSPro 7.7\CSEntry.exe" MySurvey.pff
```

This assumes that CSPro was installed in the default directory. Your PFF file must have a .pff extension.

You can create a PFF file in one of two ways: either [1] create one with a text editor (such as Notepad or Wordpad), or [2] have it generated automatically for you by launching your data entry application from within the CSPro Designer. The file will have the same name as your application, but with a .pff extension instead of .ent. For example, if your data entry application was named MySurvey.ent, the generated PFF would be named MySurvey.pff.

The following section shows the options available to you in a CSEntry PFF file. A PFF file is not case sensitive, so you can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Entry
Description=My Survey
ShowInApplicationListing=Always

[DataEntryInit]
OperatorID=John
Key=1250
StartMode=Add
AutoAdd=Yes
FullScreen=No
NoFileOpen=No
Lock=Verify,Stats
Interactive=Ask
CaseListingFilter=12

[Files]
Application=.\MySurvey.ent
InputData=.\MyData.csdb
Paradata=.\MyParadataLog.cslog
CommonStore=.\MySettings.db
HtmlDialogs=.\MyDialogsDirectory
BaseMap=.\MyMap.mbtiles

[ExternalFiles]
LOOKUP_DICT=.\LookupFile.dat

[Parameters]
Parameter=your choice
Language=LINGALA
OnExit=.\Menu.pff

[DataEntryIDs]
PROVINCE=12
DISTRICT=05
```

[Run Information]

The **[Run Information]** block is required. While Version and AppType must appear exactly as shown in the example above, there are two optional properties:

- Description=if specified, it will be used instead of the name of the file in the list of applications (mobile only) and in the window title during data entry (all platforms).
- ShowInApplicationListing=determines whether the application associated with this PFF is displayed on the mobile Entry Applications screen. The possible options are: **Always** (default), **Never**, and **Hidden**. Operators must manually elect to show hidden applications, whereas PFFs with the **Never** option will never be shown.

[DataEntryInit]

The **[DataEntryInit]** block is optional. It gives you the opportunity to customize the following runtime characteristics:

- OperatorID=specifies the operator ID for the purposes of logging operator statistics. If this line is not present but your data entry application has been set to ask for this, CSEntry will prompt the operator for one at runtime.
- StartMode=determines the mode in which CSEntry starts. The possible options are: **Add**, **Modify**, and **Verify**. If this line is not present, one of two things will occur: if the data file does not exist, then the program will start in **Add** mode; if the data file does exist, then CSEntry will wait for the operator to choose their desired mode. The operator's choices may be constrained due to options indicated in the Lock feature.

  You can also specify the case upon which to act by adding a semicolon and the case IDs of an existing case (for example: StartMode=**Modify**;010502). This will open CSEntry in **Modify** mode and open the case indicated by the ID. If the StartMode conflicts with the Lock parameter settings, then the StartMode parameter will take precedence.
- Key= specifies the case ID to automatically open. If a key is provided that exists in the data file, CSEntry will open that case and then close once the case has been entered. If the case does not exist, a new case is created and the IDs automatically filled from the values specified. If only a partial key is provided, then only the values specified are prefilled. (For example, if the IDs are cluster and household number and the key only specifies the cluster, then the cluster will be automatically filled but the interviewer will have to specify the household number.) The StartMode attribute takes precedence over the Key attribute. You generally will use only one, if any, in your design. This attribute is typically used in menu programs, often with the output of the `key` logic function.
- AutoAdd=indicates whether, while in **Add** mode, CSEntry should continuously add cases, meaning that after a case is added, CSEntry will begin adding another case. The possible options are: **Yes** (default) and **No**.
- FullScreen=determines whether CSEntry will open the application in full screen mode, with no case tree on the left. The possible options are: **Yes**, **No** (default), and **NoMenus**. The parameter **NoMenus** will open CSEntry in a tablet-friendly mode whereby the case tree and the CSEntry menus and title bar are not shown.
- NoFileOpen=determines whether CSEntry will permit the operator to open another data file while running a data entry application. The possible options are: **Yes** (default) and **No**.
- Lock=indicates what CSEntry modes the operator cannot access. The possible options are:
  - **Add**: The operator cannot add new cases.
  - **Modify**: The operator cannot modify existing cases.
  - **Verify**: The operator cannot verify existing cases.
  - **Stats**: The operator cannot view operator statistics.
  - **Delete**: The operator cannot delete cases.
  - **CaseListing**: On mobile CSEntry, the Case Listing screen will be bypassed.

  Multiple modes can be specified by listing each mode, separated by a comma.
- Interactive=specifies the default way that CSEntry runs in Interactive Edit mode. The possible options are:
  - **Ask**: The default option, CSEntry will ask the operator how to run Interactive Edit.
  - **Both**: CSEntry will display both out-of-range messages and errors generated from the `errmsg` function.
  - **ErrMsg**: CSEntry will only display errors generated from the `errmsg` function.
  - **Range**: CSEntry will only display out-of-range message.
  - **Off**: Interative Edit mode will be disabled.

  If the mode is followed by a comma and **Lock**, then the mode cannot be changed by the operator. Otherwise, unless the option is **Off**, the operator can change the mode from within CSEntry.
- CaseListingFilter=filters the cases in the data file shown to the operator. The filter string can use regular expressions. If you have multiple operators entering data to the same data file, this can be a way to prevent operators from interacting with data entered by other operators. The filter is matched against the IDs of each case.

[Files]

The **[Files]** block is required. A description of the files, not all of which have to be specified, is as follows:

- Application=the name of the data entry application that you created.

- InputData=the data file that an operator will be creating, modifying, or verifying with this data entry application.
- Paradata=the name of a file where events logged during the application's run are saved.
- CommonStore=if you use the `loadsetting` or `savesetting` functions in your program, this file overrides the default location where these settings are saved.
- HtmlDialogs=the name of a directory that contains custom HTML dialogs that override the default CSPro ones.
- BaseMap=the name of the online map or the filename of an offline map used for CSPro mapping.

[ExternalFiles]

If the **[ExternalFiles]** block is present, it means that a second (or more) dictionary was linked to the data entry application. In the example above, LOOKUP_DICT is the dictionary name, and LookupFile.dat is the name of the data file that contains the lookup codes.

[Parameters]

The **[Parameters]** block is optional. This section defines parameters for the data entry run.

- Parameter=allows you to pass in a string to your program. The parameter can be any length, although the string that retrieves the value in your program (via the `sysparm` function) must be large enough to accommodate it. Once the parameter is retrieved, it can be parsed by your program for further usage.
- Language=specifies the initial language of the program. The parameter must match the name of a language specified in the question text, dictionary, or message file.
- OnExit=specifies a PFF file to run after the data entry program closes. This can be useful, for example, if you want to relaunch a menu program after collecting data.

When starting a new case, if the name of a parameter matches the name of a dictionary item, the value of that item will be set to the value specified in the PFF. This is similar to how the persistent fields are processed (see below), but this works for non-persistent fields.

[DataEntryIDs]

The **[DataEntryIDs]** block is for use with any persistent IDs that you have defined. CSEntry will assign the specified values to the indicated persistent fields when a new data file is created. This feature allows automatic definition of persistent fields, such as geographic IDs. If you provide values and run this on an existing data file, and the PFF file values do not match the values in the data file, then the PFF values will be ignored. The syntax is as follows:

```
id_item_name=initial_value
```

Generate Binary Data Entry Application

Binary data entry applications provide security during a production data entry operation. A binary data entry application consists of one single file with extension .pen, which includes the same information as the set of text files that normally make up a data entry application.

This generated file cannot be changed in any way. It cannot be opened by the CSPro Designer and it cannot be read in a text editor. If you make changes to the data entry application, you must generate the binary application again.

The binary application will run identically in CSEntry to a normal (text) application. An operator will not see any difference in behavior.

To generate a binary application, open the application in the normal way, then from the **File** menu select **Publish Entry Application (.pen)**.

Before CSPro 6.0, binary data entry applications had the extension .enc. The 'P' in the new extension stands for "portable," which indicates that the file can be run not only on desktop CSEntry, but also on mobile devices. CSPro can

no longer read .enc files and if you would like to run an application with such a format, you will have to use an older version of CSPro.

Change Data Entry Characteristics

Change the Order of Entry

During data entry, the forms will be shown to the keyer in the order in which they appear in the forms tree. Within a form, the cursor will move among the fields in the order in which they appear in the forms tree.

To change the form order, simply drag and drop on the form tree. For example, suppose you currently have Form A, Form C, Form D and Form B in your level. The forms tree will now show the following:

1. Form A
2. Form C
3. Form D
4. Form B

If you drag the form icon for Form B and drop it on top of Form C, the forms tree will now show:

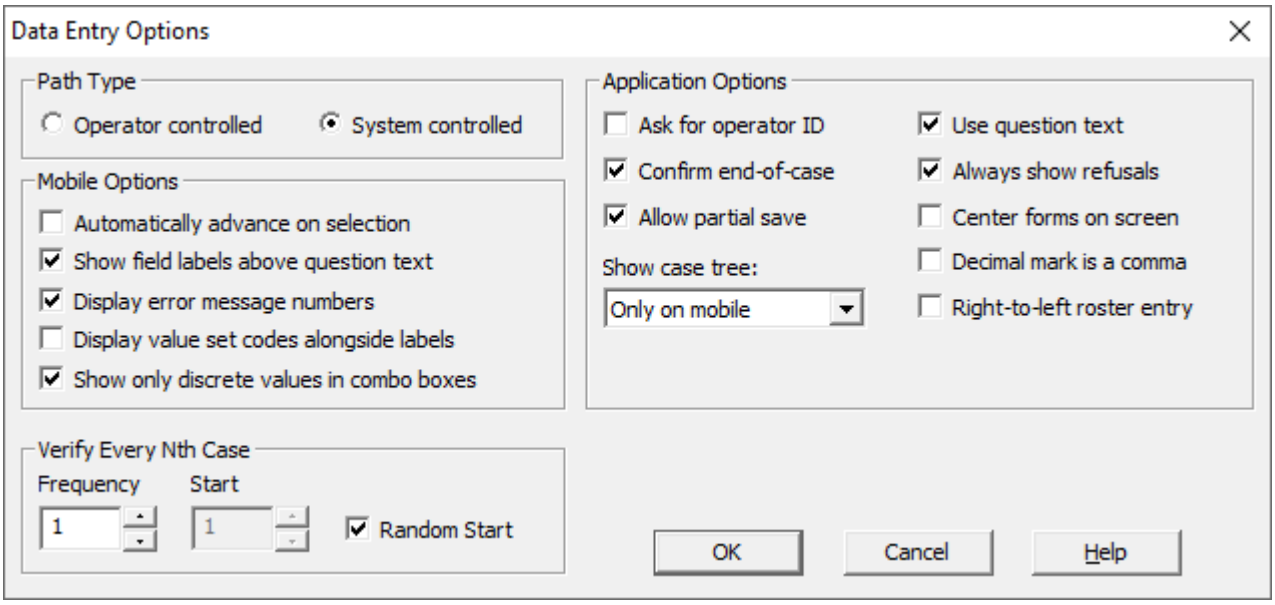
1. Form A
2. Form B
3. Form C
4. Form D

Similarly, to change the order of fields within a form, use drag and drop on the forms tree.

To change the order of fields in a roster you must drag and drop within the roster rather than on the tree. You can change the default order in which the forms and fields will be keyed by using logic in the application.

Change Data Entry Options

Select **Data Entry** from the **Options** menu to change any of the following:



Path Type

This choice is very important and will have a large effect at data entry time. See Operator vs. System Controlled for more information.

Application Options

- **Ask for operator ID**: If this box is checked, CSEntry will prompt the operator to enter an operator ID.
- **Confirm end-of-case**: If this box is checked, CSEntry will prompt the operator to accept the case at the end of each case entered.
- **Allow partial save**: If this box is checked, CSEntry will allow the operator, when in add, modify, or verify mode, to save a case which has not been completed. You can also have CSEntry automatically partially save cases.
- **Show case tree**: There are four options for the case tree selection. There are options to never show a case tree, to show it only on mobile devices (Android), to show it only on desktop devices, and to show it on all devices. If shown, CSEntry will display a tree on the left showing each item in the case currently being added, modified, or verified and its value. The tree can also be used to navigate between parts of the questionnaire.
- **Use question text**: If this box is checked, CSEntry will display the question text window. The top part is for question text (to be read during the interview). The bottom is for the normal form content.
- **Always show refusals**: If this box is checked, refused values defined in a value set will always be shown to the operator. If unchecked, the operator can view refused values by selecting the Show Refusal Options menu option.
- **Center forms on screen**: If this box is checked, CSEntry will center forms horizontally in the display window.
- **Decimal mark is a comma**: If this box is checked, CSEntry will use a comma instead of a period when showing numeric fields with a decimal component. The keyer must also type a comma instead of a period to enter the values after the decimal mark.
- **Right-to-left entry**: If this box is checked, CSEntry will orient rosters in a right-to-left manner, which is useful for languages like Arabic. The first column of a roster will be the furthest right in the roster and the roster will scroll from right to left..

Mobile Options

If you are running CSEntry on a mobile device, there are some options to consider:

- **Automatically advance on selection**: When clicking on a response (when displayed as a radio button), CSEntry will automatically advance to the next field, rather than requiring the interviewer to swipe or press the next button.
- **Show field labels above question text**: Each field's label will be displayed on the screen above the field's associated question text. If unchecked, field labels will not be shown at all.

- **Display error message numbers**: When displaying an error message (from a `errmsg` call), the message number (either a number from the message file or the line number) will be displayed along with the message.
- **Display value set codes alongside labels**: When displaying a field using a control other than a text box, in addition to showing the labels, the codes will also appear.
- **Show only discrete values in combo boxes**: When displaying the values in a combo box, only the discrete values, not the range values, from the value set are shown.

Verify Every Nth Case

During verification, you may choose to verify only a subset of the cases in the data file, instead of verifying all the cases.

- **Frequency**: This is the interval between cases that CSEntry will use for verification. For example, if this value is 10, every 10th case will be verified.
- **Start**: This is the number of the first case in the data file to verify. For example, if this value is 5, and the frequency is 10, cases number 5, 15, 25, etc. will be verified. The case number is determined by the physical order of the cases in the data file. The start value must be less than or equal to the frequency value.
- **Random Start**: You may check this box instead of specifying a start value. CSEntry will then choose a random number for the start value. The random number will be between 1 and the frequency value.

Data Source Options

With a data entry application, you can select **Data Sources** from the **Options** menu to change any of the following:

General Options

- **Automatic partial saves**: You can specify the number of minutes after which a case will automatically be partially saved.

Additional Metadata Files

- **Listing file**: If checked, a listing file will be created for the main input file.
- **Operator statistics log file**: If checked, an operator statistics file, with the extension .log, will be created for the main input file. This file is only created by desktop CSEntry and not on mobile devices.

It may be useful to uncheck both options if creating a menu or CAPI program, as these files are mostly useful in key-from-paper operations.

Multiple Operator Notes

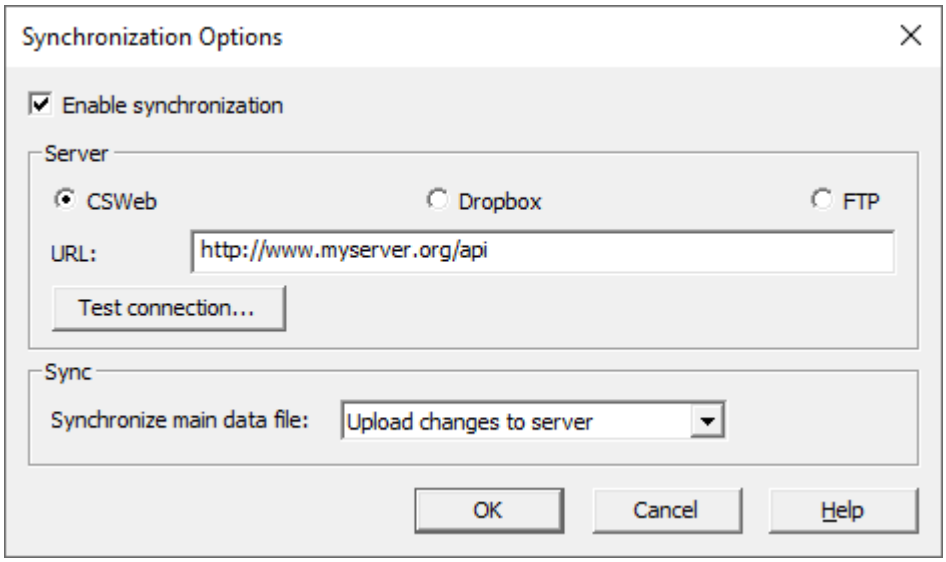
- **Delete other operators' notes**: If checked, any operator can delete the field notes left by other operators. If unchecked, an operator can only delete notes that they personally have left.
- **Edit other operators' notes**: If checked, any operator can edit the field notes left by other operators. If unchecked, an operator can only edit notes that they have personally left.

See also: GetNote Function, PutNote Function, GetOperatorId Function

Simple Synchronizations

Setting the Synchronization Options

For basic synchronization scenarios, you can configure your data entry application for synchronization by selecting **Synchronization** from the **Options** menu:



To add support for data synchronization to your application, check the **Enable synchronization** box.

The following **Server** options can be specified:

- **CSWeb**: Synchronize with a CSWeb server. This requires a properly configured CSWeb server. The data dictionary for the application must be uploaded to the CSWeb server. For more information about CSWeb servers, see the CSWeb help documentation.
- **Dropbox**: Synchronize using the online file sharing service Dropbox. This requires an account with Dropbox.
- **FTP**: Synchronize using an FTP server. This option requires that you have an account configured on an FTP server.
- **URL**: Specify the URL of the CSWeb server or the URL of the FTP server. For CSWeb servers the URL should start with http:// or https://. For FTP servers the URL should start with ftp://, ftps:// or ftpes://. The URL is not used for Dropbox.
- **Test connection**: Click this button to attempt to connect to the server to verify that the URL and settings are correct.

The following **Sync** options can be specified:

- **Synchronize main data file**: This setting determines how the main data file is synchronized. There are three options:
  - **Upload changes to server**: Only data that is modified on the device will be sent to the server. No data will be downloaded from the server. This is the most common option for interviewers when all interviewers will work on unique assignments.
  - **Download changes from server**: Only receive modified data from the server. Do not upload local changes. This might be used for a supervisor who wants to see what changes interviewers have made but does not want to make changes themselves.
  - **Sync local and remote changes**: Send local changes to the server and download changes from the server. This option can be used when multiple interviewers need to work on the same assignments. Note that if both interviewers modify a case at the same time, one will overwrite the changes made by the other.

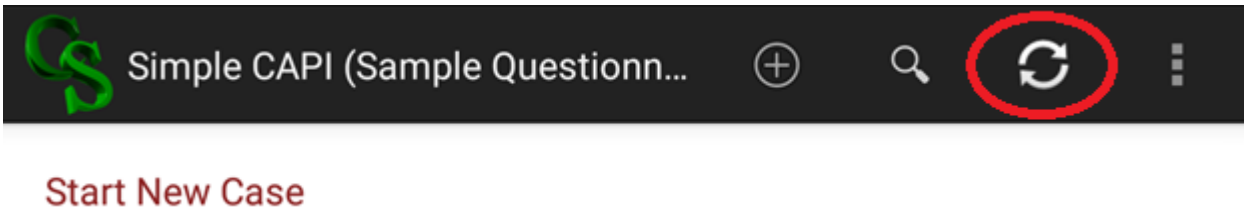
Simple synchronizations do not support synchronization of external dictionaries or files other than the .pen and .pff files. To implement these advanced synchronization scenarios, you can create your own synchronization routines using logic functions.

The synchronization options are used to synchronize data with a server but are not meant for deploying and updating the application itself. To download an application to a mobile device or update the application files of an existing application on a mobile device use the Deploy Application tool to package and upload an application to a server. Interviewers can then download the application to their mobile device using Add Application from the Entry Applications screen of CSEntry. Interviewers can also download to an application using Update Installed Applications from the Entry Applications screen of CSEntry.

Running Synchronizations

If the synchronization options have been configured, then the operator can launch the synchronization when running the data entry application. Synchronization may not be run while entering case data.

On mobile devices, the synchronization may be launched from the case listing view by choosing *Synchronize* from the menu or tapping the following icon:



On desktop devices, select the **Synchronize** option off of the **File** menu.

Downloading Data

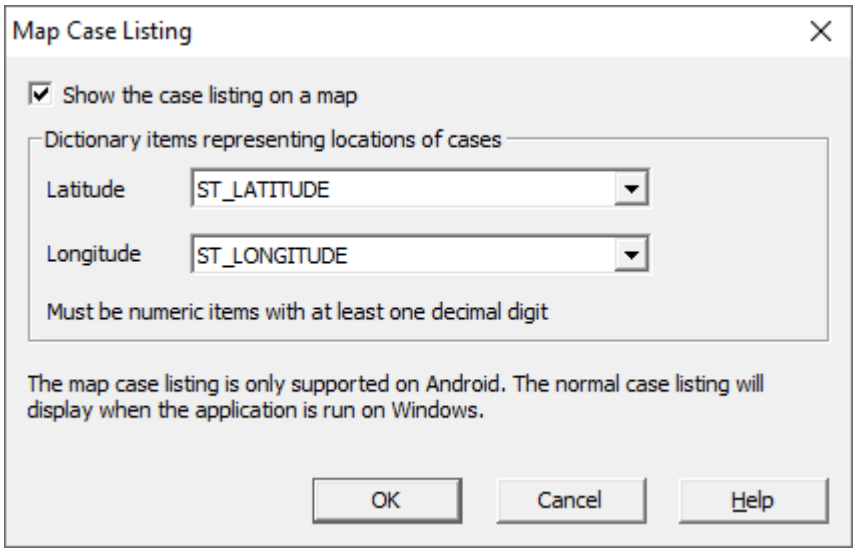
In order to download the data that has been uploaded to the server, use the Data Viewer's download function. This will download all the data on the server into a single CSPro data file that can be used by other CSPro applications and tools.

See also: Synchronization Overview

Display the Case Listing on a Map

In addition to showing the already-captured cases in a scrolling list on the case listing screen, it is possible to show the cases on a map, based on latitude and longitude variables for each case. For example, if the data entry application captures GPS coordinates for each household, the application could automatically display all of the interviewed households on a map, based on their captured locations.

To enable the map-based case listing screen, select **Map Case Listing** from the **Options** menu:



In the **Map Case Listing** dialog, check the **Show the case listing on a map** option and choose the variables in your data dictionary where the coordinates (latitude and longitude) of each case is stored. The variables must be numeric variables from the main data dictionary with at least one decimal place.

For each case in the data file, CSEntry displays a marker on the map at the location specified by the values of the selected latitude and longitude variables. Completed cases are represented by blue markers and partially-completed cases are represented by red markers. Tapping on a marker displays a popup window on the map containing the case label or case ID for the case, along with any case notes. Tapping on the popup window launches data entry for the

selected case. You can use the `setcaselabel` and `putnote` functions to customize the case label and case note displayed in the marker popup window. Both the case label and case note support a limited subset of HTML tags for formatting including <b> (bold), <i> (italic) and <font> (font, size and color).

Any cases that have blank values for the latitude or longitude variable will not be displayed on the map. This means, for example, that a partially-saved case where the GPS coordinates have not yet been captured will not be displayed on the map.

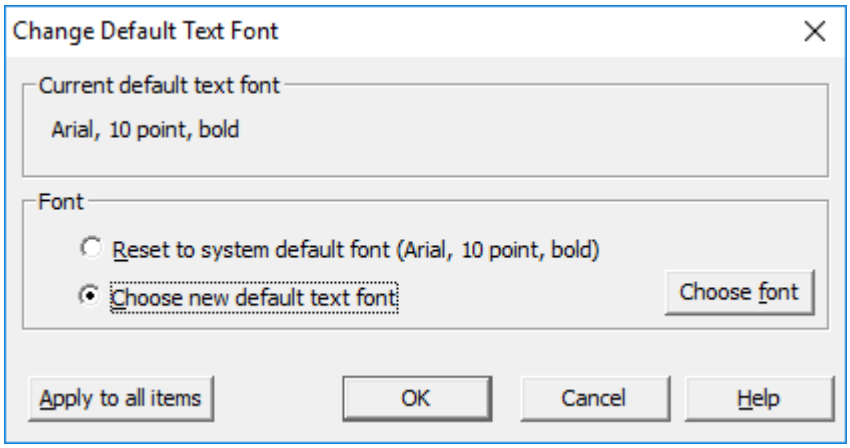
Using the map case listing is currently only supported on Android. No maps are displayed in the case listing when running your application on Windows.

By default, the cases will be displayed with a Google Maps base map, which requires that the Android device be connected to the Internet. You can modify the default base map used, or in cases where no Internet connection is available, you can specify an offline map file to use in the PFF file.

See also: Mapping, Offline Maps

Change Default Text Font

The default text font is what CSPro will use whenever you add new text to any form. From the **Options** menu, select **Default Text Font**.



You can change this by using the Font radio buttons. If you select **Choose new default text font**, you can then click on the **Choose font** button to customize your own font.

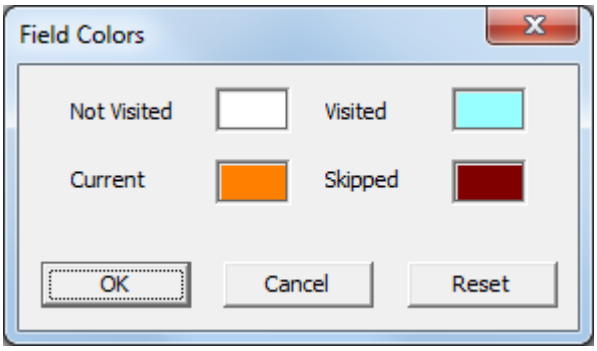
If you want to change the font for all text that is already on forms, you must press the **Apply to all items** button.

See also: Change Field Font, Change Text Properties

Change Field Colors

From the **Options** menu, select **Field Colors**. By clicking on any of the four colored boxes, you can change the color that CSEntry uses to display fields of the type listed. These changes are made on a per-application basis. Changing the field colors does not change any of the behavior of the program, including values returned from the highlighted function, it is simply for aesthetic purposes. You can change the colors of:

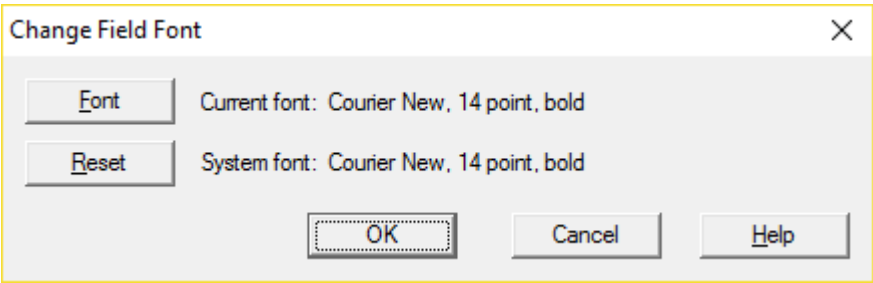
- Fields that have not yet been visited (white by default).
- The current field (white by default).
- Fields that have been visited (green by default).
- Fields that have been skipped (yellow by default in operator controlled mode; gray by default in system controlled mode).



See also: Change Field Font, Highlighted Function, Color of Fields in Data Entry User's Guide

Change Field Font

From the **Options** menu, select **Field Font**. If you change the field font you may want to also change the default text font and apply it to all items.



Press the **Font** button to change the font in all the field boxes on all the forms. Changing the size of the font will change the size of the field boxes. You can change the appearance of the characters by choosing a different script in the font dialog box.

Press the **Reset** button to reset the font in all the field boxes on all the forms to the system default.

See also: Change Text Properties, Change Field Colors

Change Error Sound

When an error occurs during data entry, an error box is shown of the screen and sound (beep, tone or other sound) is generated. In order for the sound to be heard:

- The computer must have a sound card, with a speaker connected and turned on.
- The volume on the sound system must be turned on and sufficiently loud to be heard.
- There must be a sound file associated with the Default Sound(Beep) under Control Panel/Sound.

To change the sound, go to Control Panel/Sound and change Default Sound(Beep) to a different sound file.

Forms Designer

Introduction to Forms Design

The previous topics show you how to create a data entry application using the forms generated by default. However, CSPro allows you to create, using a single dictionary, one or more forms (screens) for data entry.

This is the design stage of the data entry process. This tool allows you to create new forms, add or modify text, enter lines and/or boxes, add color to the forms or text. If you have a printed questionnaire you will probably want to use it as a guide when deciding text and field placement, as well as the order of entry for the items.

After you have developed forms to your satisfaction, use CSEntry to input the data.

This section contains the following information:

- Add Things to a Form
- Modify Things in a Form
- Change Form Properties

Add Things to a Form

Add a Form

There are three basic ways to add a new (blank) form. Each method will present you with the Form Properties dialog box.

Method 1, from the Form Designer Tree—right-click over any of the form tree entries (i.e., the Form File, a Level, Form, or Item), and a popup dialog box will appear. Select the Add Form option.

Method 2, from the Form itself—right-click anywhere over a form. A popup dialog box will appear. Select the "Add Form" option.

Method 3, from the Form Designer's Menubar—select Edit, and choose the Add Form option.

After you have entered the required (label and name) and desired (color, capture) information on the dialog, pressing OK will create the (blank) form. You will notice the form has been placed last in the form tree for the current level. You can change the form order by dragging the form within the form tree.

Add Fields to a Form

- **Drag a Dictionary Item to your Form**
  Expand the dictionary tree so that the desired item is visible. Holding down the left mouse button, select the item ( ) and drag it to the form, releasing the mouse button when the cursor is at the desired location on the form. Depending on the drag option settings, either your item or existing sub-items ( ) will be dropped onto the form. For example, if you have dragged an item from a record with multiple occurrences and you have chosen (in the **"Drag Options"** dialog) to roster items when possible, the item will appear as a one-column roster. Dragging additional items from this record and dropping them onto the roster will append the items to the roster.

- **Drag a Dictionary Record to your Form**
  Expand the dictionary tree so that the desired record is visible. Holding down the left mouse button, select the record ( ) and drag it to the form, releasing the mouse button when the cursor is at the desired location. Depending on the record's properties and the Drag option settings, the item(s) within your record will either be dropped as individual fields or as a roster.

See also: Change Field Properties

Add a Roster to a Form

CSPro automatically creates a roster, under appropriate conditions, when you drag a dictionary item onto a form. In most cases where a roster is possible, CSPro obeys the **Roster Options** on the drag options dialog box. Make sure this option is Horizontal or Vertical before you begin. In some drag and drop operations a roster is not possible and will not be created. In other drag and drop operations a roster is the only alternative.

Common ways to create a roster include:

- Drag a multiple record ( ) from the data dictionary to a blank form. This will generate a roster containing all the items in the record.

- Drag one item ( ) from a multiple record in the data dictionary to a blank form. This will generate a roster containing only that item. You can then add more items to the roster one at a time.

- Drag an item from a multiple record, or the record itself, to a form that contains only items from another single record or ID items.

- Drag a multiple item or sub-item ( ) to a form. If you have a multiple item that has sub-items, and you want to create a roster of the sub-items, make sure you have the **"Use sub-items if present"** box checked in the Drag Options dialog box.

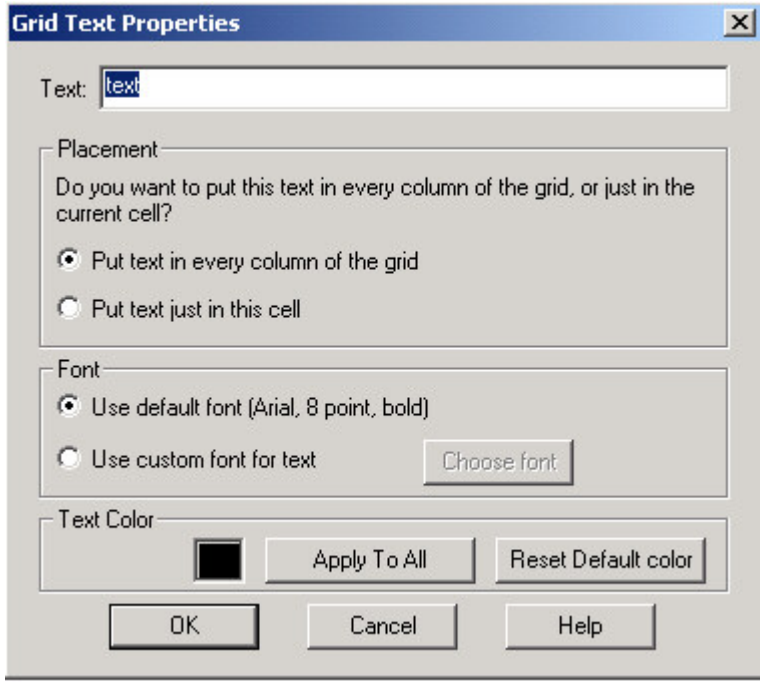
See also: Add Things to a Roster, Change Roster Properties

Add Things to a Roster

- **Add Items**
  Rosters can only include items from the same multiple record, or sub-items from the same multiple item. If you created the roster by dragging the entire multiple record or item onto the form (or by generating a set of forms), there are no more fields that can be added to the roster.

  Otherwise, you can drag an appropriate field from the data dictionary and drop it on the roster. CSPro will add a column to the end of the roster. If you don't want the field's column to be at the end, you can reposition the column after you add it by dragging and dropping the field in the desired position on the form, or in the list of fields on the Form tree. Be sure to drop the data dictionary item on top of the roster; otherwise, you will create a new roster.

- **Add Text**



  Right click on the light gray space in the desired column of the roster and select "Add Text". You will see the Grid Text Properties window. Note that you can choose the text placement and select whether the text will go only in the cell in which you clicked, or if it will go at the same position in every cell in the column. You can change this attribute later if you want.

- **Add Boxes**
  Right click on the gray space in the roster and select "Add Boxes". Note that you can choose whether the boxes will go only in the cell in which you clicked, or if they will go at the same position in every cell in the column. Drawing boxes in a roster is essentially the same as drawing boxes on a form.

Add Text to a Form

When you add a field to a form by dragging it from the dictionary tree, the dictionary item's label is automatically placed on the form. You may also add other text to the form (a heading across the top, for example) by doing the following:

1. Right-click on the form at the point where you want the text to start.
2. Select **Add Text** from the popup menu.
3. Type in the text, and press **Enter**.

See also: Change Text Properties

Add Lines or Boxes to a Form

CSPro also allows the user to draw boxes, as a means to both help visually organize your data and make the layout of your form look more professional. For example, if you wish to place fertility data on one portion of your form and then indicate to the viewer that these data are related, you could draw a box around the related items.

When you select multiple items with the mouse, you'll notice during the selection process a box that drags with you to show what you're including. To draw a box on a form, it seemed logical to have that same mechanism at work, so we've introduced the **Select Items/Boxes** button. Click on to toggle between the two states. When you first click on this button it will appear depressed, and a floating toolbar will appear with the following buttons:

Click	To
	Toggle states between selecting items and drawing boxes without having to close down the toolbar.
	Draw a box with an etched edge.
	Draw a box with a raised edge.
	Draw a box with a thin edge.
	Draw a box with a thick edge.

When you have finished drawing boxes and no longer need the Box-Draw toolbar, close it down by either toggling the button, or close the Box-Draw toolbar.

Modify Things in a Form

Selecting Items

When the forms designer first opens, the mouse is in selection mode. That is, if you click on a field, roster, or text item, the item becomes selected. Similarly, if you press the left mouse button and hold it down while dragging over a group of fields, rosters, and/or text items, all of those items will be selected. You can then choose to do operations on the selected item(s), such as moving or deleting them. If one item is selected, you can also review its individual (field/roster/text) properties. You can also hold down the **Ctrl** key while individually clicking on each item to be selected with your left mouse button.

To quickly select several fields in their entirety, just grab their data entry boxes. This will cause automatic selection of any accompanying text, as well. To quickly select just the text portion of several fields, be sure that the selection field visible on the screen does not touch any of the data entry boxes.

There are several methods of selection:

- To select several items, hold down the left mouse button while dragging a selection box around the desired items. Or you can also hold down the **Ctrl** key while individually clicking on each item (box or text) to be selected with your left mouse button.
- To select both a data entry box and its associated description text, hold down the **Shift** key while clicking on either the entry box or its associated text.
- To select several data entry boxes and their associated descriptions, hold down both the **Shift** and **Ctrl** keys while individually clicking on either the edit box or its text for each different field.

Move Things

When you drag a dictionary item onto a form, it will be placed on the form at the point where you released the mouse button. The dictionary label will be used as identifying text for the field, and it will be placed on the form according to the Drag Options in effect, which may mean the item becomes rostered. Once the field is on a form, you can fine-tune its placement.

- **Move a Field**
  To move a field, select the box and drag it to the desired location. Each field has a text item associated with it. You can see the text by holding down the Shift key and clicking on the field. This will select both the field and its text. You can now move both of them together by dragging and dropping. You can move a field's associated text separately by simply dragging and dropping only the selected text.

- **Move a Roster**
  To move a roster, select it by clicking on the gray space in any cell or in the small box in the top left corner of the roster. Drag it to the desired location. You can also resize a roster.

- **Move Text**
  To move any text, simply select and drag it to the desired location.

- **Move a Block of Items**
  First select a block of items. Then, move the mouse over one of the tracker regions selected. When you see the mouse cursor change from to , you are ready to move the block. Press down with the left mouse button and drag it to its new location.

  A tracker (or tracker region) refers to the item(s) that has(have) been selected with the mouse. Visually, you will see a heavy dashed line drawn around the item(s).

Align Things

If you have developed your form by dragging individual items from the dictionary to the form, it is probable that the fields are not precisely aligned to the left and/or right margins of the form. To correct this problem, you need only select the items you wish to align, and choose one of the alignment schemes below.

Left

This will take the left-most item (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as the basis for aligning all other selected elements. This alignment method works best for fields that have been placed on the form in a top-to-bottom manner, with text either to the left or right of the respective data entry box.

Center

This will take the [horizontal] mid-point between the left-most and right-most items (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as the basis for centering all the selected elements. This alignment method works best to center text items that have been placed in a top-to-bottom manner, or to center the text of a field over the data entry box.

Right

This will take the right-most item (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as the basis for aligning all other selected elements. This alignment method works best for fields that have been placed on the form in a top-to-bottom manner, with text either to the left or right of the respective data entry box.

Top

This will take the top-most item (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as the basis for aligning all other selected elements. This alignment method works best for fields that are spread out across the form in a left-to-right manner, with text either above or below the respective data entry box.

Mid

This will take the [vertical] mid-point between the top-most and bottom-most items (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as the basis for aligning all the selected elements on the mid-point. This alignment method works best to center text items that have been placed in a left-to-right manner, or to center the text of a field next to the data entry box.

Bottom

This will take the bottom-most item (whether the text of a field, the data entry box of a field, a roster, etc.) and use it as a basis for aligning all other selected elements. This alignment method works best for fields that are spread out across the form in a left-to-right manner, with text either above or below the respective data entry box.

Evenly Horizontal

This will evenly space three or more items (whether the text of a field, the data entry box of a field, a roster, etc.) horizontally. The left-most and right-most items will not move. This alignment works best to evenly space data entry boxes across the screen.

Evenly Vertical

This will evenly space three or more items (whether the text of a field, the data entry box of a field, a roster, etc.) vertically. The top-most and bottom-most items will not move. This alignment works best to evenly space data entry boxes one above the other.

Please note that aligning items could have unintended results. For example, if your fields are spread across the form

from left to right and you choose to left- or right-align them, they will end up superimposed, one field on top of another. Similarly, if your fields are displayed in a list-type fashion down the page and you choose to top or bottom align them, they will again end up superimposed, one field on another. If this happens, you should press **Ctrl+Z** to undo the change and restore your previous layout.

Cut, Copy, or Paste Things

To move things around in the Dictionary use **cut, copy**, and **paste**. **Cut** will delete the material from the dictionary and place it on the clipboard. **Copy** will just place a copy of the material on the clipboard. **Paste** will place a copy of the material on the clipboard into the dictionary.

- **To cut things**

  Select the material you want to cut, then click on the toolbar; or from the **Edit** menu, select **Cut**; or press **Ctrl+X**.

- **To copy things**

  Select the material you want to copy, then click on the toolbar; or from the **Edit** menu, select **Copy**; or press **Ctrl+C**.

- **To paste things**

  Select the place where you want the records, items, or values to be pasted, then click on the toolbar; or from the **Edit** menu, select **Paste**; or press **Ctrl+V**.
  You can paste cut or copied material to more than one location. Use undo if you paste to the wrong place.

See also: Undo

Resize and Reposition Things in a Roster

Changing the Roster Size

Select the roster by clicking on the gray space in any cell. You will see eight small black squares (the resize handles) around the edges at the corners and sides of the roster. Move the mouse pointer on top of any resize handle until the mouse pointer changes to a double-headed arrow. Click and drag to the desired size. CSPro will automatically create or remove scrollbars as needed.

Changing the Column Width

Move the mouse pointer over the right edge of the column you wish to resize until the mouse cursor changes to a double-headed arrow. Click and drag to the desired width.

Changing the Row Height

Move the mouse pointer over the bottom edge of the row you wish to resize until the mouse cursor changes to a double-headed arrow. Click and drag to the desired height.

Changing the Order of Columns

At data entry time, fields are keyed in the same order in which they appear in the roster columns, left to right (or top to bottom if the roster orientation is vertical). To change the order of columns, click on the column heading and drag to the desired position. A gray separator line will tell you where you are about to drop the selected column.

Moving Fields, Text, or Boxes

Select the object by clicking on it. Move the mouse pointer over the object until the mouse pointer changes to a four-headed arrow. Click and drag to the desired position.

Join and Split Roster Columns

By default, CSPro puts one field in each column. You can put two or more fields in a column by using the **Join** facility.

Joining Two or More Columns

Select two or more adjacent columns. You can do this by holding down the **Ctrl** key and clicking on each column. Right-click and choose **Join** and type in the text for the joined column.

Splitting a Column

Click on the column heading to select it, then right-click and choose **Split**. If a column already has more than one field in it (from a previous join), you can **Split** the column so that there is one column for each field.

Delete Form Elements

Deleting a Field From a Form

Click on the field on the form, or on the forms tree, then press **Delete**, or right-click and choose **Delete**.

To Delete a Form

Click on the form on the forms tree and press **Delete**, then choose **Delete Form** from the main menu at the top.

Matching the Application to the Data Dictionary

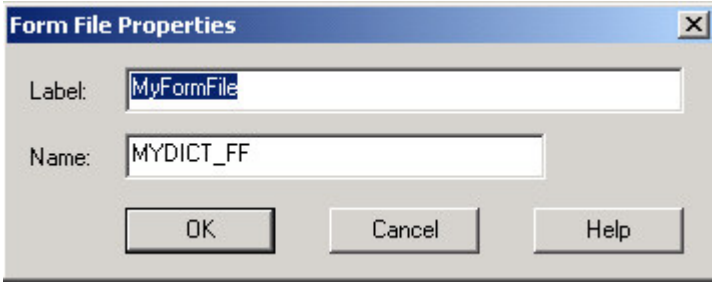
Whenever an existing application is specified, the system automatically checks to make sure the application matches the data dictionary. This is to alert you in case a change was made to the data dictionary after the application was last saved, or in case you typed in the wrong data dictionary name.

If the application does not match, the system will indicate that discrepancies exist and will ask you to permit the updating of the application. If you prefer not to accept the update or if you wish to first investigate the cause of the discrepancy, you may answer **No** and the system will not open or update the application. You can then verify that you have the correct dictionary and review any changes that might have been made since the last time the application was opened.

Change Form Properties

Change Forms File Properties

Right click on the form file ( ) on the tree (top most entry) and choose Properties.



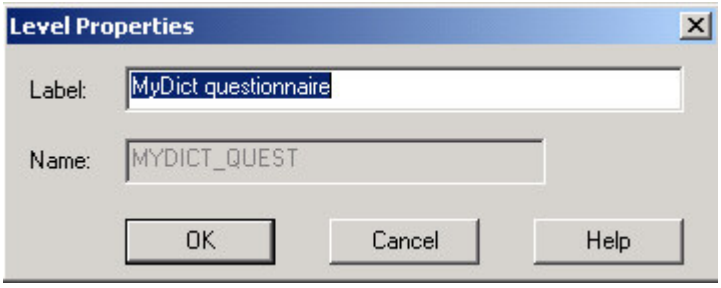
- **Label**
  This is descriptive text that helps you identify the current forms file. It may contain any characters (including blanks) and be up to 120 characters long.

- **Name**
  This is the name of the forms file which you would use when writing programming logic. It must consist of letters, digits and the underscore ('_') character. It must not begin or end with underscore.

  You can see either labels or names on the forms tree. Press **Ctrl+T** to switch back and forth between them.

Change Level Properties

Right click on the level ( ) on the tree and choose "Properties."



- **Label**
  This is descriptive text that helps you identify the current level. It may contain any characters (including blanks) and be up to 120 characters long.

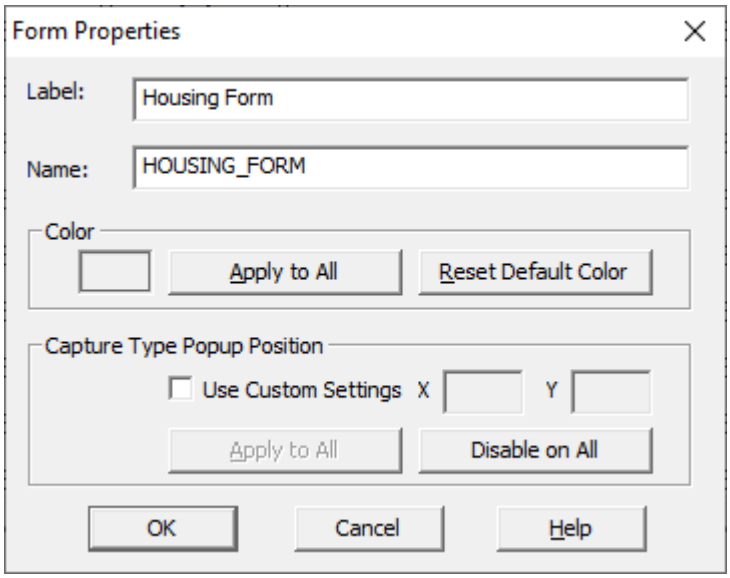
- **Name**
  This is the name of the level which you would use when writing programming logic. It must consist of letters, digits and the underscore ('_') character. It must not begin or end with underscore.

  You can see either labels or names on the forms tree. Press **Ctrl+T** to switch back and forth between them.

Change Form Properties

Right-click on the form button on the tree and choose **Properties**, or right click on empty space on the form itself and

choose **Form Properties**.



Label

This is descriptive text that helps you identify the current form. It may contain any characters (including blanks) and be up to 120 characters long.

Name

This is the name of the form which you would use when writing programming logic. It must consist of letters, digits, and the underscore ('_') character. It must not begin or end with underscore.

You can see either labels or names on the forms tree. Press **Ctrl+T** to switch back and forth between them.

Color

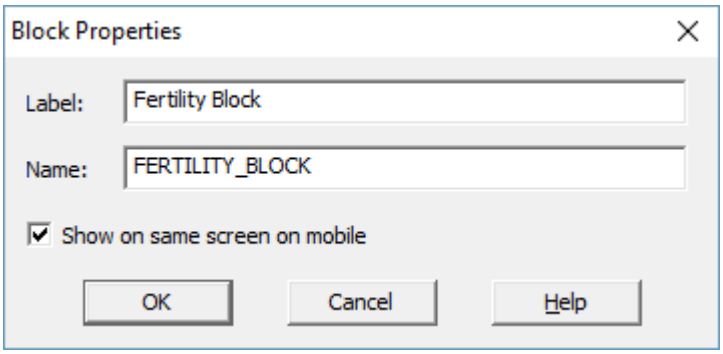
The button shows the color of the form. To change the form color, click on this button, select a new color and click **OK**. You can change the form color back to what it was originally (usually gray) by clicking on the **Reset Default Color** button. You can make all forms the same color by clicking on the **Apply to All** button.

Capture Type Popup Position

If your program uses capture types, you can specify where on the form you want the popup boxes to appear. Enable this option by clicking on the **Use Custom Settings** checkbox and then specify the X (horizontal) and Y (vertical) coordinates for the boxes. The (0,0) origin point of the form is located at the form's top left corner. When editing a form, the status bar will show the X and Y coordinates of your cursor, which is one way to determine your desired coordinates. Clicking **Apply to All** will set the capture position for all forms. Clicking **Disable on All** will turn off this functionality for all forms and the popup windows will appear in their default positions, as close to the field as possible. The capture position can also be set in logic using the `setcapturepos` function.

Change Block Properties

To bring up the **Block Properties** dialog, right-click on a block in the tree and select **Properties**.



Label

This is descriptive text that helps you identify the block. It may contain any characters (including blanks) and be up to 120 characters long.

Name

This is the name of the block, which you use when writing programming logic. It must consist of letters, digits, and the underscore ('_') character. The name must begin with a letter and cannot begin or end with an underscore.

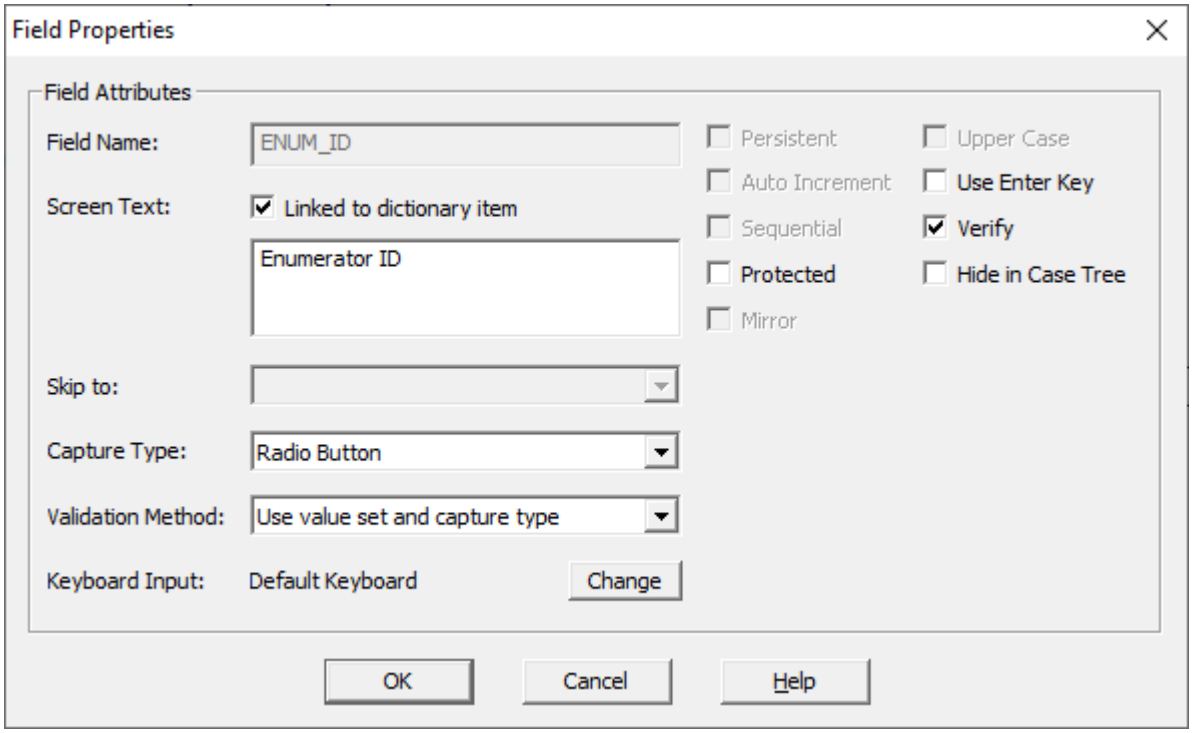
Show on same screen on mobile

When this option is selected, all the fields in the block are shown on the same screen during mobile data entry. The operator can fill in these fields' values in any order before continuing with data entry. If the option is not selected, each field will be displayed separately, as is the case for fields that are not part of blocks.

See also: Blocks

Change Field Properties

Right-click on a field in the tree or on the form and select **Properties**. You can also press **Alt+Enter** to bring up the properties window.



Field Name

This is the name of the dictionary item associated with this field. It is the name you use to refer to this field when writing logic. Mirror fields will show the dictionary name with three digits appended to it. You cannot change this property.

Screen Text

This is the text that is associated with the data entry box on the form. It is also the text that is displayed on the mobile case tree for the field. If **Linked to dictionary item** is checked, then the screen text is synced to the label of the associated dictionary item. That is, if the dictionary item's label changes, then the screen text will change automatically.

Skip to

This is the name of the field that will be skipped to if the operator presses the plus (+) key on the numeric keypad. If the "skip to" field it is blank and the plus key is pressed, CSPro skips to the next field in sequence. "Skip to" is available only in operator-controlled data entry mode.

Capture Type

This attribute allows you to specify a capture type (a popup window that shows entries in the value set) for the field, or to change the appearance of an alpha field.

Validation Method

This attribute allows you to specify how CSEntry validates the value entered in the field. The default option **Use value set and capture type** means that CSEntry will ensure that a value is part of the field's value set (if applicable), and also, for date and check box capture types, that the value is valid (beyond simply the value set checks). The option **Allow out of range with confirmation** means that, if the value is not in the value set or invalid based on the capture type, the operator will be presented with a confirmation message and the operator can accept the invalid value. The option **Allow out of range without confirmation** allows the entry of any value into the field. More information about validation can be found on the capture types page.

Keyboard Input

This attribute allows you to choose a keyboard ID that specifies what keyboard input to use for the field, which may be useful in multiple language environments.

Persistent

Check this box to make a field persistent. Persistent fields are ID fields that take the value from the previous case in the data file as their default. Persistent fields are typically used for geographic IDs that change very rarely from one case to another. These fields are shown as light gray boxes on the form. In CSEntry, the operator must press a special key (**F7**) to change the value of a persistent field. You can make any ID field (except for mirror fields) persistent.

Auto Increment

Check this box to make a field auto increment. Auto increment fields are ID fields that receive an automatic value based on a calculation using the other cases in a data file. For the first case in the data file, the ID field is set to 1. For subsequent cases, the field takes the value of the highest value in the data file plus 1. Auto increment fields are similar to sequential fields except that they work on ID fields across multiple cases, whereas sequential fields work across a repeating group in a single case.

Sequential

Check this box to make a field sequential. Sequential fields automatically increment at data entry time. They are commonly used as occurrence-number fields in multiple groups. A sequential field takes the value 1 on the first occurrence. For subsequent occurrences, CSEntry will use the value of the previous occurrence and add 1. If the field is not also marked as protected, the operator may change the sequence at any time by simply keying a new value, and from that point, CSEntry will use this new value to continue the sequential incrementation. You can make any field (except for mirror fields) sequential. You can define your own kinds of sequential behavior for fields by writing pre-processing logic. In this case, do not use the sequential field attribute.

Protected

Check this box to make a field protected. Protected fields are not keyed during data entry. Protected fields are commonly used to display a value that is calculated elsewhere (for example, the sum of other keyed fields). You must write logic to set the value of a protected field. You can make any field protected.

Mirror

Mirror fields show the value of a previously entered field on the screen. The cursor never goes to a mirror field during data entry. Mirror fields are useful to display values from one screen on another screen. Any singly occurring item can be a mirror field. A common use of mirror fields is to show the geographic IDs on all screens. The first form might contain the geographic ID fields which the operator keys in, and subsequent forms might contain the geographic ID mirror fields, which will show the operator the ID values even when the ID form is not on screen. The first time you drag a dictionary item onto a form you create the normal entry field. On each subsequent occasion that you drag the same dictionary item onto a form, you create a mirror field.

Upper Case

Check this box to make a field upper case. Alphanumeric fields can be upper case. This means that every alphabetic character that is keyed will be forced to upper case.

Use Enter Key

Check this box if you want to force the data entry operator to press the Enter key to advance to the next field. If left unchecked, the cursor automatically advances to the next field (after the maximum number of characters have been entered).

Verify

Check this box if you want to verify the field when the operator is in verification mode. If left unchecked, verification is skipped. If checked, verification will occur as follows: after each field is keyed, the value entered is compared with the value currently in the data file. If there is a difference, an error message is displayed, and the field must be reentered.

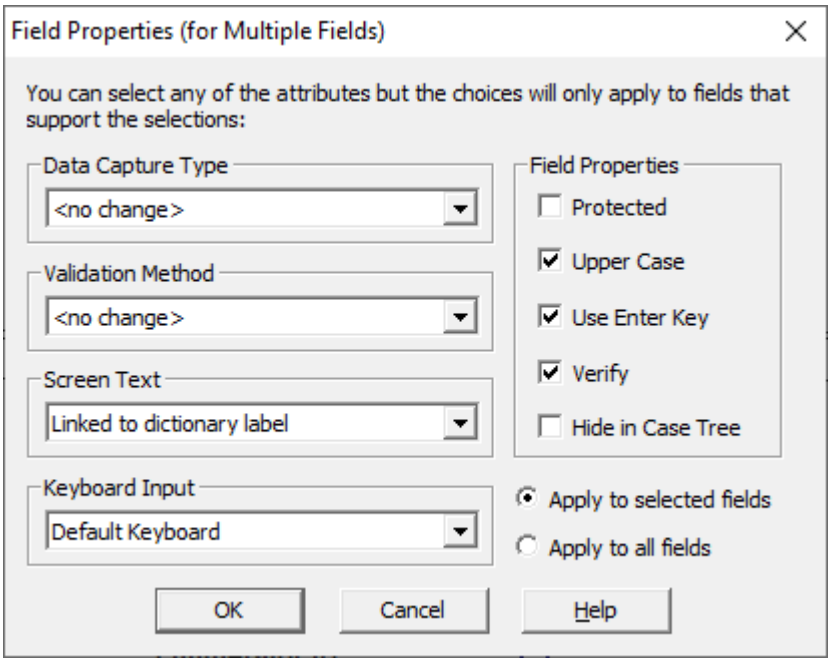
Hide in Case Tree

Check this box if you do not want the field to appear in the mobile case tree. You might, for example, have some control fields that the interviewer does not need to see when conducting the interview.

See also: Change Field Properties for Multiple Fields

Change Field Properties for Multiple Fields

It is possible to change the properties for many fields at one time. To change the properties for all fields in your application, select **Field Properties** from the **Options** menu. Alternatively, select multiple fields on a form and then right-click on a field and select **Properties**, or press **Alt+Enter**.



The selected attributes will only apply to fields that support the options. For example, if **Upper Case** is selected, that option will be ignored for numeric fields and will be applied only to alpha fields. Selecting **<no change>** means that that setting will be ignored when applying the new properties.

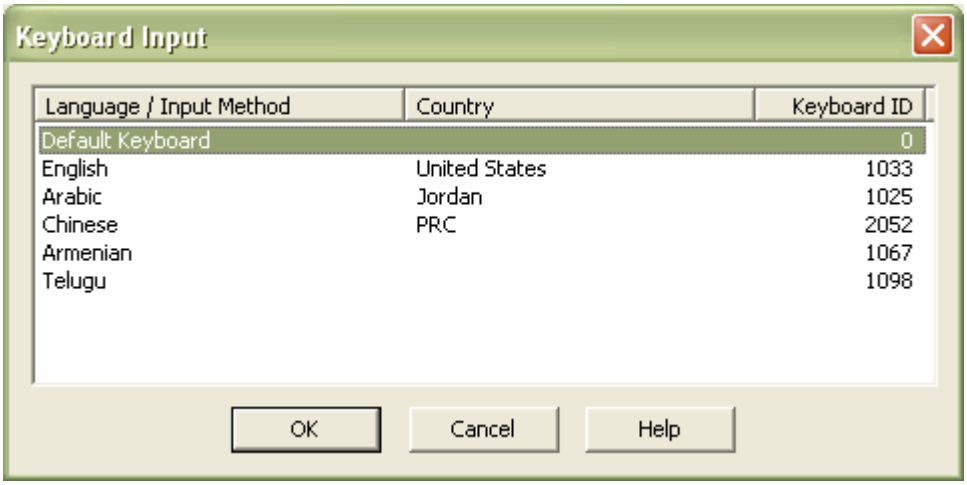
The **Data Capture Type** options allow you to change the type of capture type used for the field. It will list the union of valid possibilities for the selected fields, but if a change is requested it will apply the change only to fields that can support the specified type. For example, selecting **Check Box** will only apply that option to alphanumeric fields that can support checkboxes. The capture type of fields that do not support the selected type will remain unchanged. If you would like to set the capture type to the default based on the field's item and value set properties, select **<default for field>**.

If you selected multiple fields on a form to bring up this dialog but would like to apply the changes to all the fields in your application, select the **Apply to all fields** option.

See also: Change Field Properties

Change Keyboard Input

To manually change the keyboard input for a field, view the field's properties and select **Keyboard Input** -> **Change**.



You can specify a specific keyboard input method for a field if you would like to avoid making keyers change input methods manually. The list is populated by currently activate input methods. If an input method is not listed, activate it first, typically by modifying the Regional and Language Options in the Control Panel.

The keyboard input method will only apply to the selected field. After the keyer moves away from the field, the input

method will return to the default keyboard setting. For example, suppose that there are four fields:

Field Name	Keyboard ID
Field1	0
Field2	0
Field3	1067
Field4	0

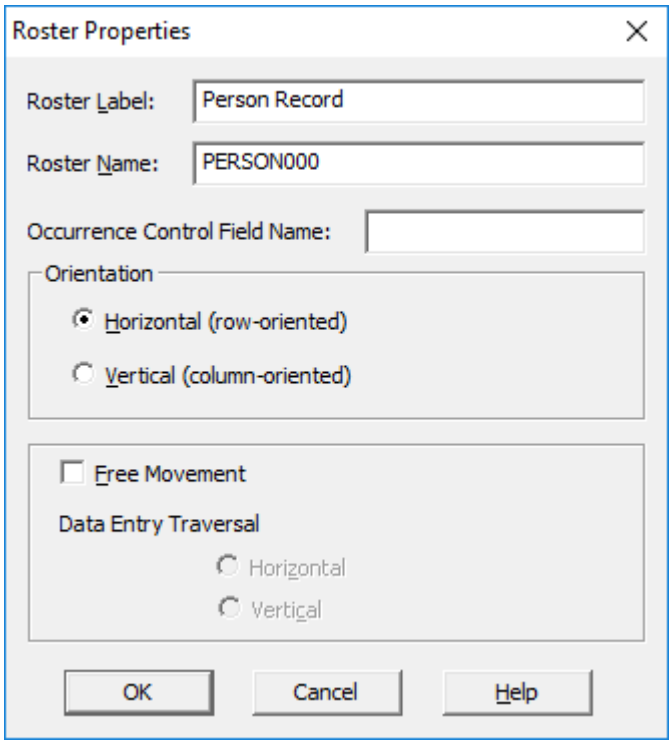
In the above scenario, the keyboard input method will only be modified for Field3, which will be entered in Armenian. However, suppose that the keyer manually changes the keyboard from the default keyboard to Arabic while on the Field2. In that case, Field3 will be entered in Armenian, and then the input method will be changed back to Arabic for Field4. In other words, the Default Keyboard input method resets the keyboard back to whatever input method was last active on another field with a keyboard ID of 0.

When CSEntry loads the data entry program, it also loads any keyboard input methods that are not currently active on the machine. The input methods will be unloaded when CSEntry terminates. However, some input methods require additional files that do not come in a standard Windows installation. CSEntry can only use input methods that contain all the required files. If the files do not exist, the Default Keyboard will be used for those fields. Before deploying your application you will want to ensure that the required input methods are installed on each keyer's machine.

See also: ChangeKeyboard Function, Change Field Properties for Multiple Fields

Change Roster Properties

Right click on the gray space in any roster cell and choose **Properties**.



Roster Label

This is descriptive text that helps you identify the current roster. It may contain any characters (including blanks) and be up to 120 characters long.

Roster Name

This is the name of the roster which you would use when writing programming logic. It must consist of letters, digits and the underscore (_) character. It must not begin or end with underscore.

Occurrence Control Field Name

If set to a singly-occuring field on the form, entry in the roster will stop after the keyer has entered the number of rows determined by the occurrence control field name's value.

Orientation

This defines whether the cursor will move from left to right or from top to bottom during data entry.

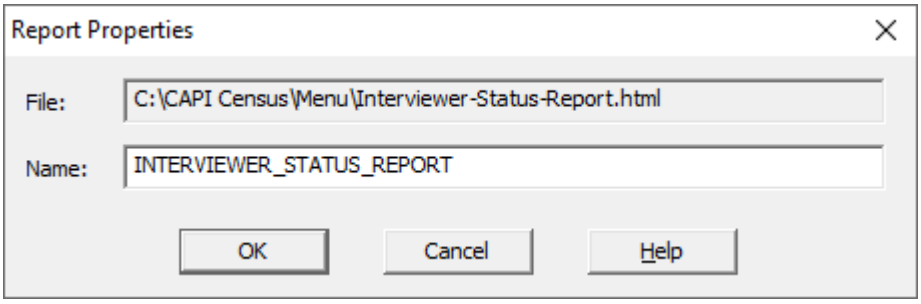
Free Movement

This defines the order in which to enter a record with multiple occurrences: all the items in a record n times or each item n times before continuing with the next item

You can see either labels or names on the forms tree. Press **Ctrl+T** to switch back and forth between them.

Change Report Properties

To bring up the **Report Properties** dialog, expand **Reports** in the tree and then right-click on a templated report and select **Properties**.



File

This is the full filename of the templated report.

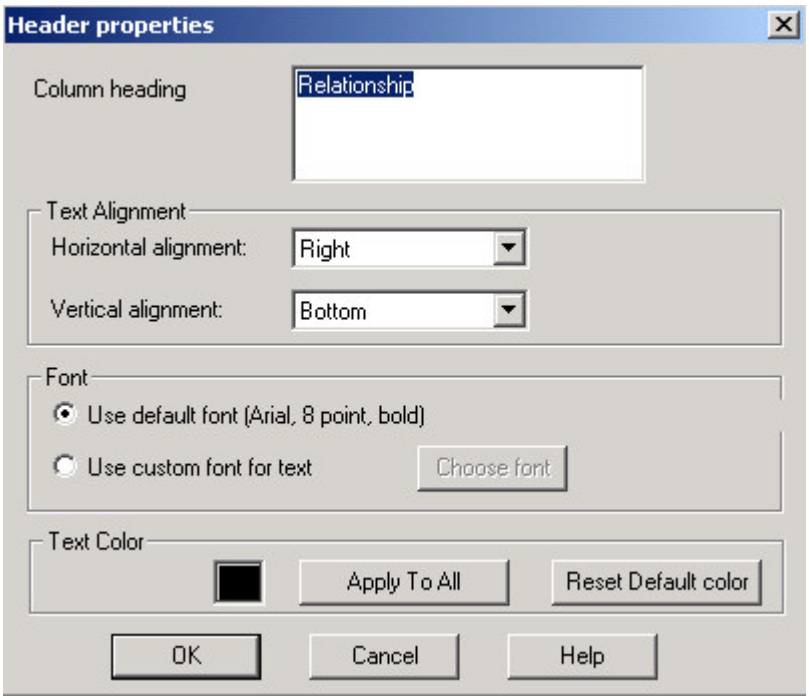
Name

This is the name of the templated report. The name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter.

See also: Templated Reports

Change Column Heading Properties

Right click on a column heading and choose **Properties**.



- **Column Heading**
  This is the text that shows in the heading. CSEntry may automatically wrap text to make two or more lines, if the column width is small. You can force your own multi-line text by using **Ctrl+Enter** at the end of each line. For example if you type "Age of", then **Ctrl+Enter**, then "Mother", you will have two lines of text no matter how wide the column is.

- **Horizontal alignment**
  This allows you to force the text to be left-aligned, right-aligned, or centered within the column heading area.

- **Vertical alignment**
  This allows you to force the text to be aligned at the top, middle, or bottom of the column heading area.

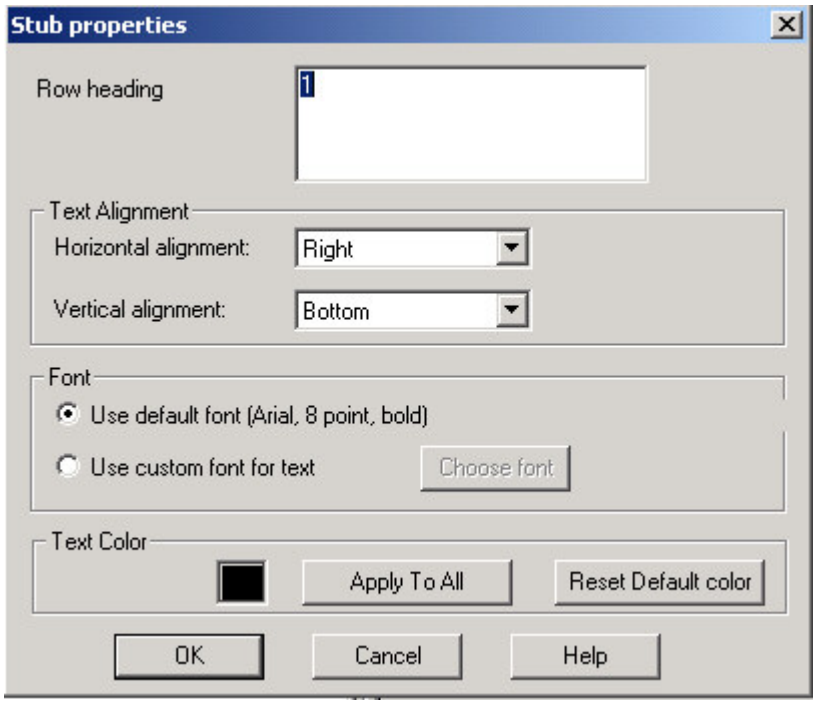
- **Font**
  To change the font of the column heading text, choose the "Use custom font for text" radio button then click on the "Choose font" button.

- **Text Color**
  The button shows the color of the selected text. To change the text color, click on this button, select a new color and click "OK." You can change the text color back to what it was originally (usually black) by clicking on the "Reset Defaut Color" button. You can make all text on all forms the same color by clicking on the "Apply to All" button.

See also: Change Row Heading Properties

Change Row Heading Properties

Right click on a row heading (occurrence label) and choose "Properties."

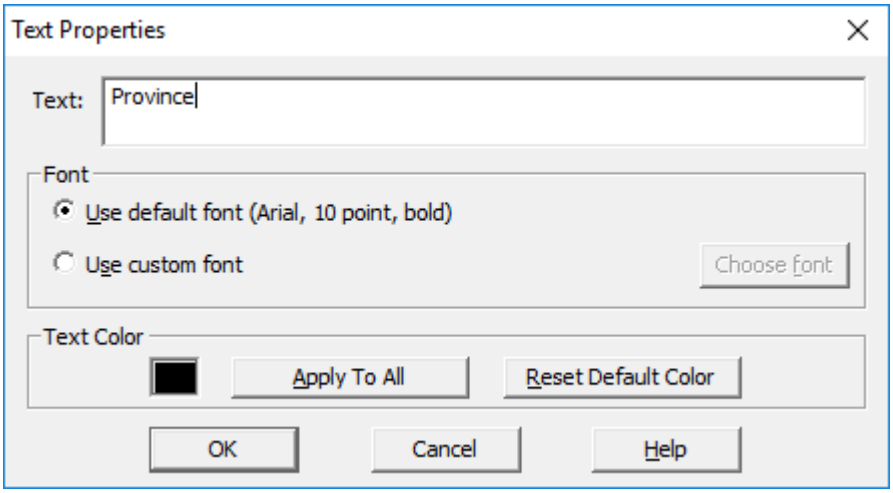


- **Row Heading**
  This is the text that shows next to the row. CSEntry may automatically wrap text to make two or more lines, if the width of the occurrence label area is small. You can force your own multi-line text by using **Ctrl+Enter** at the end of each line. For example if you type "College or", then **Ctrl+Enter**, then "University", you will have two lines of text no matter how wide the occurrence label area is.

See "Change Column Heading Properties" for a description of the rest of the properties.

Change Text Properties

Select any text item to change the text itself. Select any text item or group of text items to change their font. Once selected, right-click and choose **Properties**. You can change the font for all text fields on all forms by choosing **Option -> Default Text Font** from the main menu.



Text

Enter the new text here. It may contain any characters (including blanks) and be up to 120 characters long.

Font

This shows what font is in effect for the selected text. To change the font, select the **Use custom font** radio button then click on the **Choose font** button.

Text Color

The button shows the color of the selected text. To change the text color, click on this button, select a new color and click **OK**. You can change the text color back to what it was originally (usually black) by clicking on the **Reset Default Color** button. You can make all text on all forms the same color by clicking on the **Apply to All** button.

Data Entry Editing

Introduction to Data Entry Editing

As we saw in previous chapters, a data entry application contains a set of forms (screens) which a data entry operator uses to key data to a disk file. Data entry applications can be used to add new data to a file and to modify existing data. You can also include logic to perform consistency checks, generate complex skip patterns, look up information in one or more external files, and display messages after any field is entered. A number of statements and functions in the CSPro language set may be used only within data entry applications. They are generally related to the creation and/or modification of a case, and may require operator interaction.

You use the Forms Designer module of CSPro to develop the data entry application. You use CSEntry to run the data entry application, and use the CSPro language to develop validation procedures to be carried out at data entry time.

This section contains the following information:

Editing Concepts
Writing Logic

Editing Concepts

Type of Edits in Data Entry

Validate Individual Data Items

These checks are designed to determine whether a response has a value that is inside or outside the valid limits for that response as defined in the dictionary. The data entry application can be designed to allow forcing out-of-range values, to force the operator to reenter a valid value, or not to allow any input on the item.

Perform Structure or Consistency Edits

You can write code to check the structure of the case or test the consistency within related items. These instructions can be written for any object such as a case, level, form, record, roster, or field. The instructions can be executed before the cursor moves into the object (`onfocus`); at the beginning of an object (`preproc`); after cursor moves off the object (`killfocus`); or at the end of the object (`postproc`). You can also perform Interactive Editing after you finish entering a case.

Display Error Messages

The system displays automatic error messages if the item is out-of-range, but you can also use the `errmsg` function to write customized messages to be displayed in the screen during data entry.

Control the Data Entry Flow

You can use `skip` or `advance` statements to control the data entry flow in a case or end data entry at any time if a particular condition occurs. Also see the `endgroup` and `endlevel` statements.

Control Stopping Data Entry

The `OnStop` function can be used to keep the operator from stopping data entry or to allow stopping only under certain conditions. You can also `stop` the application for the current case or terminate the operation. You can stop the application at any time and the system will save the partial case. Also see the `ispartial` function.

Write Notes

You can add notes to a field, which can be viewed by the operator and/or edited. These notes can be used to display instructions to the keyer or to elaborate on a particular value. See the `putnote`, `getnote`, and `editnote` functions.

Generate Reports

You can `write` customized reports to a file. For example, you could create a report to summarize the demographic characteristics in the household.

Use Secondary Forms

You can use a secondary form to `enter` data under certain conditions.

Display Option Menus

The system displays a window with the valid values and the operator may select the correct one. See the `accept` or `showarray` functions.

Use External Files

The most common functions are `loadcase` and `retrieve`. The `selcase` Function can only be used in data entry applications.

Examine Content of Item

You can examine the content of a numeric item with the `visualvalue` function or alphanumeric item with the `highlighted` function before the item has been keyed.

Structure Edits

Structure edits can be used to check coverage and to establish that each case has the correct number and type of records. The tests that make up a structure edit ensure that the questionnaires are complete before beginning the consistency edit. In a typical Housing and Population census, the structure edits will check that:

- For all collective and conventional households within an enumeration area, all required records are present and are in the proper order.
- Each enumeration area (EA) batch has the correct geographic codes.
- Where required, each household has one and only one housing record.
- Each occupied housing unit has at least one person record, and vacant housing units (where permitted) have no person records.
- Within a household (conventional or collective), there must be one and only one person record for each member of the household (i.e., no duplicate or missing records) such that the total number of person records is equal to the recorded count of persons in the household.
- Within each conventional household, there is one and only one qualified head of household; within each collective household, there is no head of household.

Consistency Edits

Consistency edits look at the individual data items within a questionnaire or case, and examine the validity of each item and the consistency of each item with respect to other related items. For example, the degree of educational attainment of an individual should have some predictable relationship with the person's age. This means that it is not expected that a 9-year-old child will have progressed much beyond the fourth or fifth year of elementary school, depending on local standards. If, in the unedited census data, a 9-year-old indicates educational attainment above that level, it is probable that either the age or the educational level have been incorrectly recorded. Whether the age or the educational level is modified to produce data consistency is a decision left to the subject-matter specialists, who should know whether age data or education data are more correctly reported. It is also important to take into account the method of data capture; some, like scanning, are prone to systemic error (that is, where a "0" is consistently read as a "9," for example). If such error is not recognized and taken into consideration when assessing the reliability of reported information, it can lead to even greater error in the final data.

At a minimum, consistency checks should seek to resolve all errors which might eventually lead to doubts about the quality of the data. That is, the subject-matter specialists must consider the types of tabulations to be produced and the uses to which the data may be put in the future. For example, if the plan includes tabulations of educational achievement by age and sex, the edit specifications must include a means of detecting and correcting the data for individuals whose declared educational achievement is not in line with their declared age (as in the example of the preceding paragraph). If a published table shows even one 9-year-old having completed secondary school, the quality of the data will be called into question, simply because (in that particular culture) it is not the norm that a child so young could be so advanced educationally. It makes no difference if, in fact, the child is a prodigy and did indeed complete secondary school; when it is a case of one or two "outliers" and the weight of probability is against them, the values must be changed and brought in line with reasonable expectations. It is not recommended that such changes be made only in the logic which generates the tables; if the data (even a subset) are to be given out to researchers or other users outside the statistical office, they must be clean and consistent throughout before distribution.

Checking Errors

You can check for errors at the time you enter the data or perform interactive editing after the data have already been entered.

You can use the same programmed logic that was in effect during data entry to find problems that were left unresolved by the original keyer, or you can use different logic to check for other conditions.

Improperly identifying errors can waste precious personnel resources, so a precise set of rules should be developed with input from subject-matter specialists.

During data entry the system automatically displays a message if value for item is out-of-range. However, you might want to write your own message for missing or out-of-range values. For example, if enumerators had been instructed to record all persons older than 110 years of age as '110,' the first-pass check for errors might include the following code:

```
PROC AGE
if AGE in 0:110 then
  exit; { the age range is OK, nothing else to do }
else
  errmsg ("Person %d, has incorrect age: %d", $);
endif;
```

So what does this code do? If a person's age is in the range from 0 to 110 ('0' is for infants born less than 12 months before the Census reference date), then the value is accepted as valid and program flow exits the procedure. If not, then the value is either outside this range or missing, in which case the subsequent errmsg statement will be executed, showing the reported age for Person N.

More involved edits may be needed for other variables. For example, fertility information is only asked of a female of a certain age. So if fertility information is present, you may wish to confirm the values of other variables. A possible test could be as follows:

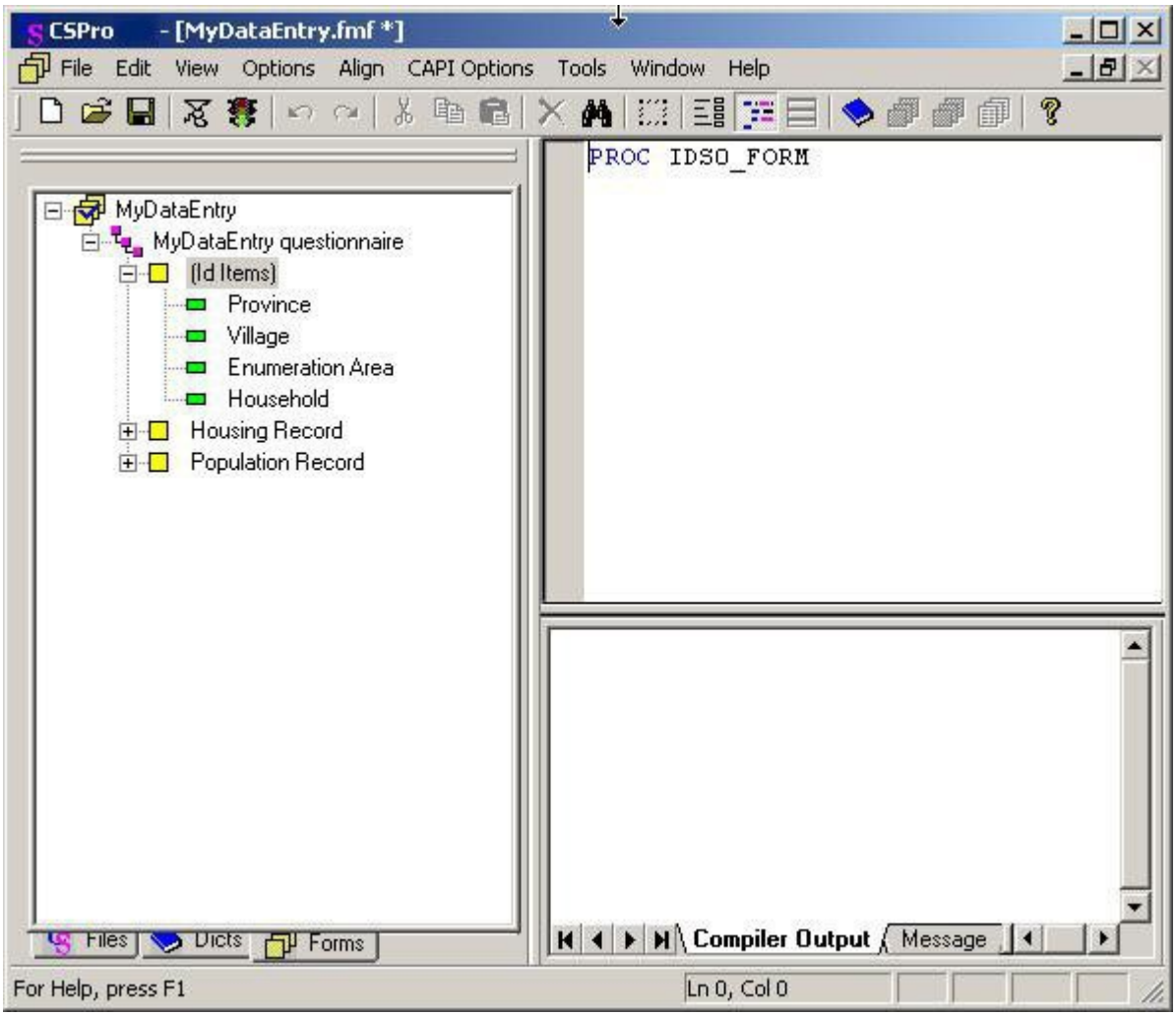
```
PROC FERTILITY
if FERTILITY in 0:20 then { there are 0-20 children }
  if sex = 1 then {if sex = male}
    errmsg ("male has fertility info present"); {message displayed}
    reenter; {operator must enter valid value}
  else {sex is not male}
    if sex = 2 then {if sex = female, check woman's age }
      if age < 15 then { 15 = minimum age for fertility }
        errmsg ("woman is too young (%d) to have children", age);
      endif;
    endif;
  endif;
else { Fertility is blank }
  if sex = 2 then { a problem if the woman is "of age" }
    if age >= 15 then
      errmsg ("woman aged %d should have fertility info", age);
    endif;
  endif;
endif;
```

As you see, the complexity of the logic used to find (and soon, correct!) errors will depend on the specifications provided. In the case where specifications are minimal, it is important that the programmer consider all situations/paths in developing the logic.

Writing Logic

Data Entry Logic Screen Layout

Logic for data entry screens is written when developing the application. Forms Designer permits viewing either the forms or the logic associated with the application on the right-hand side of the split screen. To initiate the process of creating logic for an application, click on ; or from the View menu select "Logic"; or press **Ctrl+L.** To go back to the forms, click on ; or from the View menu select "Form"; or press **Ctrl+M**.



The screen is divided into three main work areas: the Tree View, the Logic View and the Message View.

- **Tree View**
  The window on the left half of the screen displays the data entry tree with the first form ( ) selected.

- **Logic View**
  This is the window block in the upper portion of the right half of the screen. It is the programmer's "clean slate, to which may be added logic for any part of the data file: any item, any section, any record, even the file as a whole. It is up to the programmer to determine the correct placement and sequence of execution for each logical element. The initial screen `(PROC IDSO_FORM)` represents the Identification form or initial form. If you move to the top of the tree ( ), then the logic view will display:

```
{Application 'MYDATAENTRY' logic file generated by CSPro }
PROC MYDICT_FF
```

```
PROC GLOBAL
```

This is the beginning of your program. This two lines of code will always be in your application file. You write the declaration statements under `PROC GLOBAL`, then the procedures for the event.

If code has been written for a given edit level, form, roster, or field, a check mark will appear superimposed on the icon for that entity. This is how, at a quick glance, you can see where you have placed programming logic. Once one line of code has been written anywhere in the program, a check mark will appear on the Forms File icon.

- **Message View**
  This is the window block in the lower portion of the right half of the screen. It is devoted to messages (user-created and system-generated). As with the Tree View, tabs are available to the programmer; clicking on one of them will make the contents of that view active. The **Compiler Output** tab displays errors found during compilation of your program; if the code compiled successfully, it will state "Compile Successful." The **Message** tab is used to type in error messages that will be used in the execution of the program.

If you wish to modify the size of any of these three work areas, just place the mouse over one of the separating bars, grab it, and drag to resize.

Moving Around a Logic Application

To move around your edit application, select individual items from the **Forms** (in a data entry application) or **Edits** (in a batch application) tree tab. The logic view will update to display the programming logic (if any) for that item. If you select the root of the tree, all logic written for the entire edit application is displayed.

For example, suppose you select AGE from the tree and there is no associated programming logic, you will see in the logic view:

```
PROC AGE
```

Since there is no logic, `PROC AGE` is generated "on-the-fly" and will not be saved in the logic file. If there was associated logic, you might see something like this:

```
PROC AGE

    if not ( AGE in 0:99 ) then
        errmsg("Invalid age found");
    endif;
```

Order of Executing Data Entry Events

An "event" is a unit of logic associated with an element of the edit tree, such as a form, a field, etc. The order in which events are executed during data entry is dependent on two factors:

1. The inherent order imposed by the CSPro software design, and
2. The order in which data items are entered, which is determined by the designer of the application

CSPro is based on the concept of a "case" containing one or more types of "collections of information." The "case" will usually correspond to the questionnaire used in the survey or census, and the "collections of information" [or groups of data items] will usually correspond to one or more record types that make up the case/questionnaire. These elements constitute a hierarchy, and in applying logic, CSPro follows this hierarchy; that is, it begins with any logic that pertains to the file itself, and works "down the tree" through the various levels. CSPro executes application events one case at a time. During data entry, `preproc`, `onfocus`, `onoccchange`, `killfocus` and `postproc` statements are executed in the order in which they are encountered.

The following diagram illustrates the default order of events for a two-level data entry application that has no `skip` or `advance` statements that might otherwise alter the program's natural flow. Level 1 has two forms (1 and 2) and level 2 has one form (3). This description applies to both system- and operator-controlled applications.

```
Form File preproc
    Level 1 preproc
        Form 1 preproc
        Form 1 onfocus
            Field 11 preproc
            Field 11 onfocus
            (entry of Field 11)
            Field 11 killfocus
            Field 11 postproc
            ...
            Field 19 preproc
            Field 19 onfocus
            (entry of Field 19)
            Field 19 killfocus
            Field 19 postproc
        Form 1 killfocus
        Form 1 postproc
        Form 2 preproc
        Form 2 onfocus
            Field 21 preproc
            Field 21 onfocus
            (entry of Field 21)
            Field 21 killfocus
            Field 21 postproc
            ...
            Field 29 preproc
            Field 29 onfocus
            (entry of Field 29)
            Field 29 killfocus
            Field 29 postproc
        Form 2 killfocus
        Form 2 postproc
    Level 2 preproc
        Form 3 preproc
        Form 3 onfocus
            Field 31 preproc
            Field 31 onfocus
            (entry of Field 31)
            Field 31 killfocus
            Field 31 postproc
            ...
            Field 39 preproc
            Field 39 onfocus
            (entry of Field 39)
            Field 39 killfocus
            Field 39 postproc
        Form 3 killfocus
        Form 3 postproc
    Level 2 postproc
    (repeat level 2 procedures for other level 2 nodes)
    Level 1 postproc
Form File postproc
```

Note that the natural flow through the fields can also be altered by the use of the cursor or mouse. For example:

Scenario 1

Given: Three fields (A, B, and C). From Field A, click on Field C. Assume all fields have `preproc`, `onfocus`, `killfocus`, and `postproc` events.

Result: Field A's `killfocus` would execute, but its `postproc` would not. Nothing would execute for Field B. Finally, Field C's `preproc` and `onfocus` would execute.

Scenario 2

Given: Form 1, which contains one field, Field 1. Form 2, which also contains one field, Field 2. After keying Field 1, you are automatically advanced to Form 2, Field 2. You then decide to use the up/left arrow to move back to Form 1. Assume both forms and both fields have `preproc`, `onfocus`, `killfocus`, and `postproc` events.

Result: Field 2's `killfocus` would execute, but its `postproc` would not. Next, Form 1's `onfocus` would execute. Finally, Field 1's `onfocus` would execute (but its `preproc` would not). Note that it does not matter how many fields are on Form 1; the `onfocus` for Form 1 would always execute.

Essentially, if the programmer uses logic, or if the data entry operator moves backwards or forwards with the mouse or arrow keys, the natural flow of the program will be altered. If exiting a form, field, or roster prematurely, the `killfocus` event will execute but the `postproc` event will not. Similarly, if entering a form, field, or roster by backing up into it, the `onfocus` event will execute but the `preproc` event for it will not.

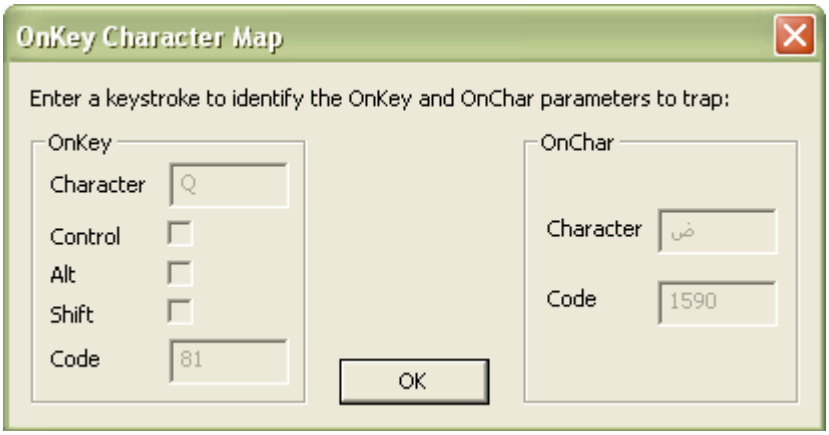
See also: CSPro Program Structure, Order of Executing Batch Edit Events

Sequence Dictated by Designer

In the course of creating the application, the designer may choose to modify cursor movement under certain conditions. In the example of a population census, if data are being captured for a male, or for a female under the age of 15, the application designer may wish to skip over the items relating to fertility because they are not applicable to these population subgroups. If the skip occurs (whether programmed or through operator control), the edit sequence will be modified accordingly. In addition, if the operator moves backward and forward through the forms, the edit sequence may also be affected, depending on the type of control (operator or system).

OnKey Character Map

When editing a data entry application, select **View** -> **OnKey Character Map**.



The OnKey Character Map is used to determine the codes for keystrokes or characters. Type a character and the OnKey and OnChar codes for that character will be displayed in the Code field. In the above image, an Arabic keyboard was used to generate the letter Dad. In this example, the OnKey function will be called with the keystroke value 81 because the 'Q' key was pressed to generate the letter 'Dad.' The OnChar function will be called with the character value 1590, representing the Arabic letter.

See also: OnChar Global Function, OnKey Global Function

User and Configuration Settings

With a data entry or batch edit application active, select **View** -> **User and Configuration Settings**. A dialog box will open that lets you add, modify, or delete user and configuration settings. These settings are attribute-value pairs that are alternate ways to store information (rather than using external dictionaries, working storage dictionaries, or text files). A setting, the "value," is stored in a way that can be looked up using an "attribute" key. Both the value and attribute are

strings.

User Settings

User settings are accessed from logic with the `loadsetting` function and are modified using the `savesetting` function. Unless otherwise specified, all CSPro applications on a device share the same settings, which make user settings a convenient way to share information across different applications. For example, a menu program might save some information:

```
PROC INTERVIEWER_NAME

    savesetting("InterviewerName", INTERVIEWER_NAME);
```

And then another program might automatically fill in the interviewer name based on the saved setting:

```
PROC INTERVIEWER

preproc

    INTERVIEWER = loadsetting("InterviewerName");
    noinput;
```

Configuration Settings

Configuration settings are stored locally on your device and are loaded at compile time. The value of a configuration setting is accessible by declaring a string variable using the `config` type. Unlike user settings, configuration settings cannot be modified in logic. They can only be modified using the dialog box.

Because binary data entry applications (.pen files) are only compiled with the .pen file is created, the configuration settings stored on the computer used to create the file are saved in the .pen file.

Configuration settings are a convenient way to store some information outside of your logic file. If you are using a revision control system (e.g., git), there may be some values, like the URL you are using to synchronize data, that you do not want to store in your publicly visible logic. You can define these settings on your machine without having to share them with strangers. For example, this is logic that you can safely commit to GitHub:

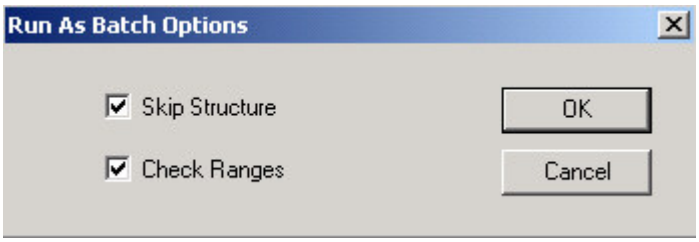
```
PROC GLOBAL

config ftp_server_url;

PROC EXAMPLE

    syncconnect(FTP, ftp_server_url);
```

For this to work, you need to use the dialog box to add an attribute **ftp_server_url** with its value being the FTP server URL.

See also: Config Statement, LoadSetting Function, SaveSetting Function, CSEntry Settings Modification

Run as Batch

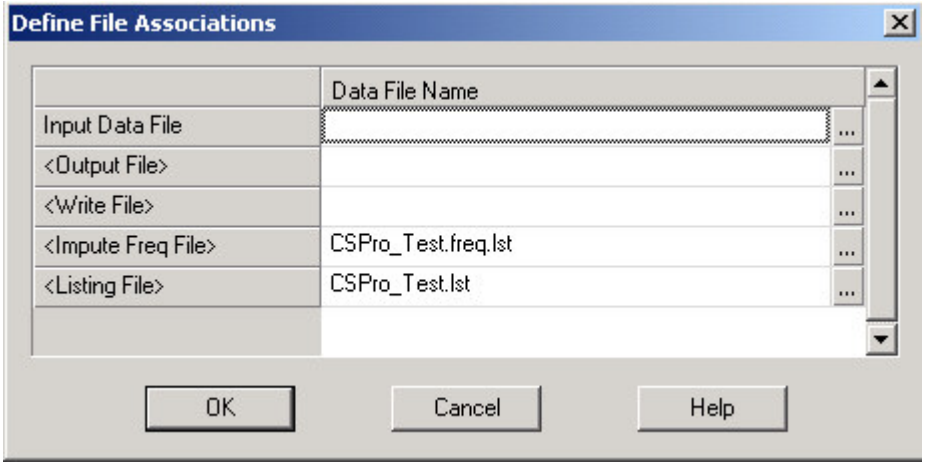
You might also run the data entry application after you finish entering data for a file. If you select "**Run as Batch**" under the File menu, the system will display:



- **Skip Structure**
  If this option is checked, the application will check for entries in items that should be blank when they are skipped under certain conditions, and will list those items.

- **Check Ranges**
  If this option is checked, the application will list the items with out-of-range values.

After you press "**OK**" the system will display the following dialog box:



- **Input Data File**
  This line is required, and asks for the name of the data file against which you wish to run your batch application. This data file will not be modified in any way; it will only be opened, read, and closed.

- **Output File**
  The output file is where the results of correcting your data will be written. If you are **not** making any corrections in your program, then the generated file will be an exact copy of the original data file. If you **are** making corrections to your data file, then this will be the corrected data file. The default file extension is .out, but you can use whatever you'd like. This field is optional; therefore, if you are making corrections to your data, but forget to specify an output file, no corrected file will be generated.

- **Write File**
  If you have any write functions in your program, they will write information to this file. The default file extension is .wrt, but you can use whatever you'd like. This field is optional; therefore, if your program contains write statements, but you forget to specify a write file, no file will be generated. Similarly, if you indicate a write file but your program does not contain any write statements, no file will be generated.

- **Impute Freq File**
  If your program contains any impute statements, the results of this command will be written to this file. The default file extension is .frq, but you can use whatever you'd like. This field is optional; therefore, if your program contains **impute** commands, but you forget to specify a frequency file, no file will be generated. Similarly, if you indicate a frequency file but your program does not contain any impute commands, no file will be generated.

- **Listing File**
  This line is required, and asks for the name of the file to which you want to write the results of the run. Results from errmsg functions will be written here. This file will always be generated, regardless of whether or not your program includes errmsg commands.

CAPI Data Entry

Introduction to CAPI

CSPro supports Computer-Assisted Personal Interviewing (CAPI) data entry. In CAPI the interviewer uses a laptop, tablet, or mobile device to enter responses as they occur. The interview is conducted face-to-face, and the entry application (sometimes called the "instrument") determines the question order, and also performs editing of responses.

CAPI offers a flexible approach to collecting data, and can result in improved data quality and more efficient interviewing and processing. Creating good instruments, however, requires training and experience on the same level as creating good batch edit programs.

CATI (computer-assisted telephone interviewing) is similar to CAPI, but interviews are conducted over the phone instead of in person.

This section contains the following information:

- CAPI Features
- Capture Types
- CAPI Strategies
- Multimedia Features
- How to ...

CAPI Features

A CAPI application is similar to a regular data entry application, but involves making use of some additional features. CSPro offers the following features suited to implementing CAPI surveys:

Feature	Description
Question text	Customized text for each question can be displayed at the top of the entry form.
Help text	A help shortcut (**F2** on desktop or the **?** button on mobile) can bring up customized help information for each field.
Multiple languages	The CAPI entry application can be developed with question and help text in several languages; the interviewer can switch among languages as needed.
Response boxes	During entry, a response list of valid values for each question can be optionally displayed; these can make the interview go more smoothly.
Notes	Interviewers can enter field notes when needed.
Support for partial save	CAPI applications can be developed so that they support partial save. This lets the interviewer save an incomplete case, then return later to complete it.
Customized branching	The entry application can determine on the fly which forms or questions to present based on the user's previous responses. This is accomplished through program logic.
Immediate editing	Responses can be edited, and errors or possible mistakes signaled to the interviewer. This can help improve data quality.

See also: Multimedia Features

Capture Types

Capture Types

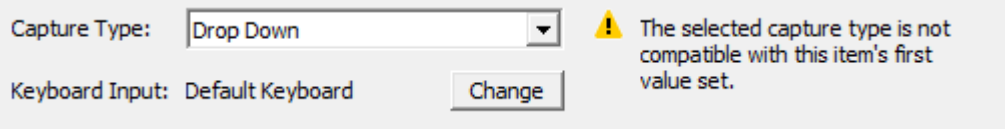
Occasionally it may be useful to display to an enumerator or keyer the possible values for a given field in a data entry application. For example, if a survey is being conducted on a tablet, it may be easier for an enumerator to use a finger or stylus to select a value, rather than to key the value using the screen's touch keyboard. CSPro's capture types allow for such flexibility by allowing the program to specify the capture type associated with a field.

Selecting Capture Types

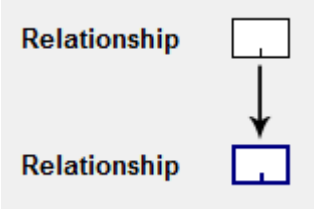
You can specify a capture type while editing a dictionary item, or while modifying a field's properties. If a field has a capture type specified, it will override the dictionary item's capture type setting.

When dragging an item to a form to create a field, you can indicate that you want to automatically use capture types by selecting the **Capture Types: CAPI Mode** option in the Drag Options dialog. If this option is selected, CSPro will automatically choose a capture type for the field based on the field's first value set.

You can manually change the capture type for a field by editing the field's properties. CSPro allows you to select a capture type that is suitable for the item's first value set. A warning will appear if the value set is not suitable. In this case, if the value set is compatible at the time of data entry (because the value set was changed dynamically using the `setvalueset` function), it will be used; if not compatible, the field will appear using the base type (generally a Text Box).



The CSPro Designer shows the fields that use capture types by coloring the border of the field in blue. The blue border does not exist when the data entry application is run in CSEntry. The field border will be a lighter blue color when the Number Pad capture type is selected.



Capture Types

- Barcode - Capture contents of a barcode.
- Check Box - Capture a multiple response variable.
- Date - Capture a date.
- Drop Down / Combo Box - Capture a discrete value as a drop down or a numeric range as a combo box.
- Number Pad - Capture a numeric value.
- Radio Button - Capture a discrete value.
- Slider - Capture a numeric range.
- Text Box - Capture numeric or alphanumeric text. The default capture type.
- Toggle Button - Capture a discrete value with two states (selected and not selected)

Hiding the Title of a Capture Type Window

By default CSEntry displays the label of the value set in the window of the capture type, but this title can be turned off using programmatic logic:

```
setproperty(dictionary_symbol, "ShowExtendedControlTitle", "No");
```

Validation Method

Capture Type	Numeric Field	Alphanumeric Field
Barcode	In value set	In value set
Check Box	n/a	Comprised of values in the value set
Combo Box	In value set	In value set (only for fields of length 1)
Date	In value set and a valid date	In value set and a valid date
Radio Button / Drop Down	In value set	In value set
Slider	In value set	n/a
Text Box / Number Pad	In value set	In value set (only for fields of length 1)
Toggle Button	In value set or `notappl`	In value set or blank

See also: Multimedia Features, GetCaptureType Function, SetCaptureType Function

Barcode Capture Type

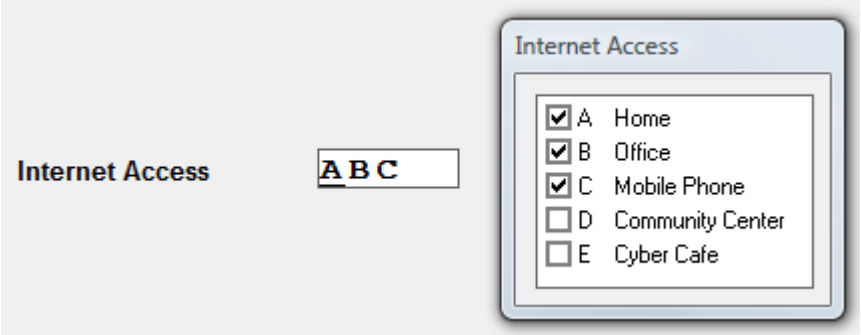
The Barcode capture type starts an Android device's camera and allows the operator to scan a barcode. The barcode can be in several formats, including QR codes. Note this capture type only appears on Android. On Windows a Text Box is displayed.

It is also possible to capture barcodes using the `barcode.read` logic function.

See also: Capture Types, Barcode.Read Function

Check Box

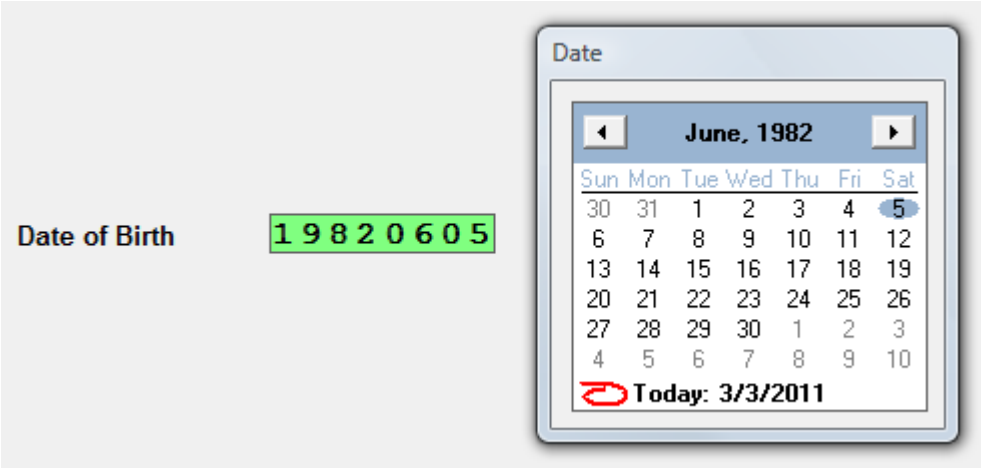
Check Boxes are used to capture multiple response variables. Items using Check Boxes must be alphanumeric. Each value in the value set corresponds to one of the responses and will be displayed as a separate check box. When the boxes are checked, the corresponding values will be put into the alphanumeric field, from left to right. The length of the field must be a multiple of the maximum length of a response. For example, if each response code is one character and you have five responses, then a field of length five would allow for the selection of all responses. A field of length two would allow for only two responses (e.g., "pick the top two...").



See also: Capture Types

Date Capture Type

The Date capture type is valid for numeric and alphanumeric items of length 4, 6, or 8. The item does not need to have a value set. The choice of date format affects how the date is stored in the field.

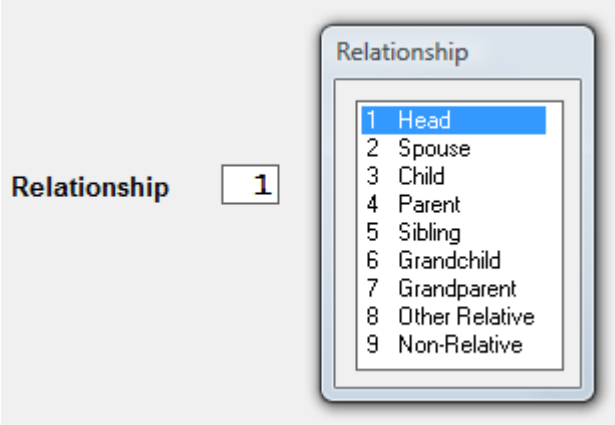


See also: Capture Types

Drop Down / Combo Box

These capture types display all values in the value set of numeric and alphanumeric items. The Drop Down capture type is identical to the Radio Button capture type, except that it displays differently on Windows. In contrast, the Combo Box capture type allows for the displaying of ranges of values (that is, value sets with "to values").

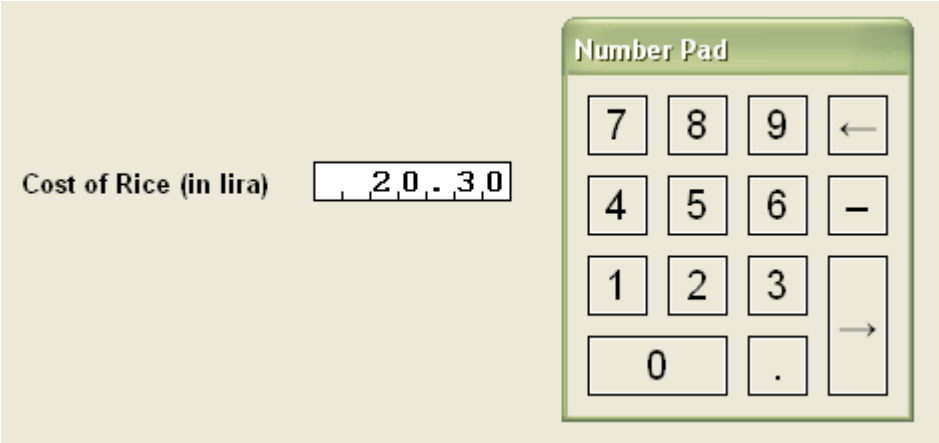
While the Drop Down and Combo Box capture types are visually identical on desktop CSEntry, in the mobile environment there is a difference between the two options. Drop Down fields, which contain only discrete values, will be displayed as radio buttons. Combo Box fields, which can include ranges, will appear as text boxes with a corresponding button that displays the entire value set, allowing users to see the valid ranges as well as to select from discrete values (such as a Missing code).



See also: Capture Types

Number Pad

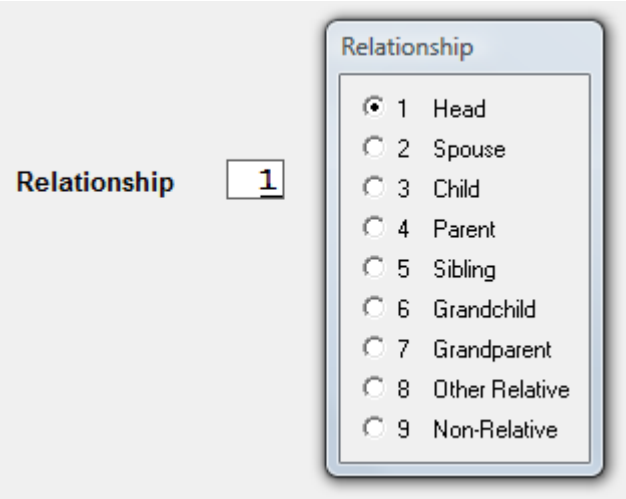
Some users find it cumbersome to use the on-screen keyboard while entering data on a tablet using Windows. This capture type displays a number pad for entering numeric values with a finger or mouse.



See also: Capture Types, Text Box

Radio Button

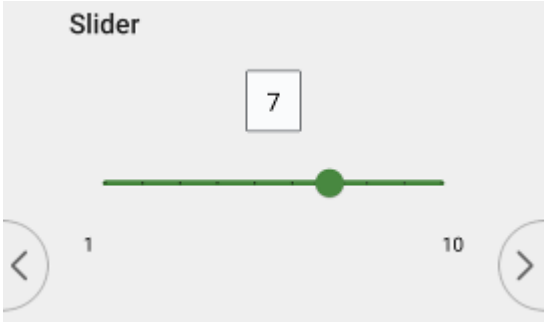
The Radio Button capture type can be selected if an item is either numeric or alphanumeric and if the value set has only discrete values, meaning that there are no "to values" defined.



See also: Capture Types, Toggle Button

Slider

The Slider capture type allows the selection of a numeric range. A range can be defined in the "to values" of the item's value set. Note that this capture type only appears on Android. On Windows a text box is displayed, but only a numeric value can be entered.



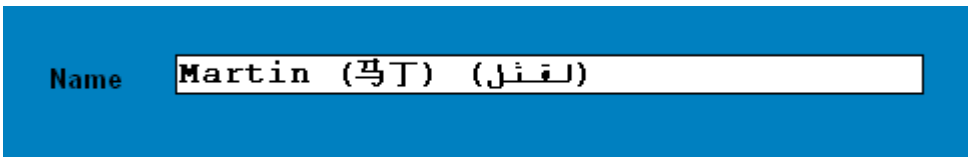
See also: Capture Types

Text Box

The Text Box attribute is the default option for a field and indicates that no capture type will be displayed. On Windows, tickmarks divide each character. Text boxes with tick marks are not resizable.

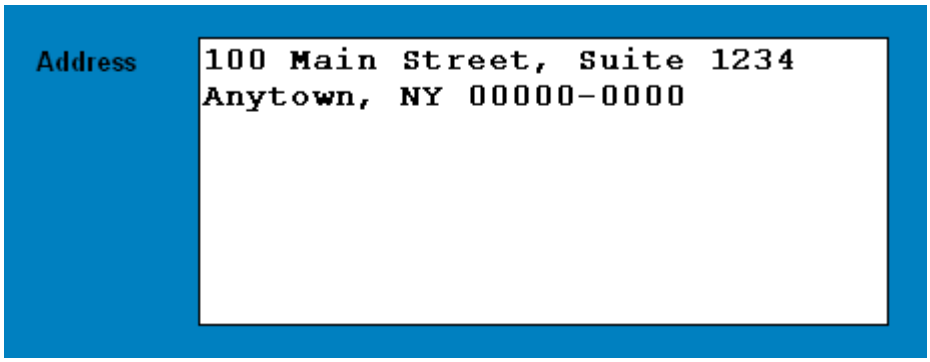
Text Box (No Tickmarks)

The Text Box (No Tickmarks) option is only available for alphanumeric fields. On Windows, CSEntry will not display any tickmarks and characters will be displayed with connecting features, if any are associated with the script being used. This is useful for some scripts, like Arabic, which have elements that connect, or for other languages that require multiple characters to create a composite character. In the CSPro Designer, you can click on the field and resize it horizontally to set an appropriate size. During entry, ahould an interviewer wish to paste values into this field, they can trigger the context menu by pressing Shift+F10 and choose the paste option.

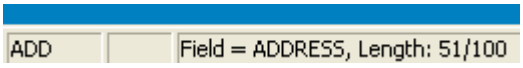


Text Box (Multiline)

The Text Box (Multiline) capture type provides a convenient way to collect multiple lines of information in one alphanumeric field. This may be useful for information like an address or a memo field. In the CSPro Designer you can click on the field and resize it horizontally and vertically to set an appropriate size. In CSEntry on Windows, the data entry operator can move up and down in the field using the arrow keys, but as with other fields, pressing Enter will move to the next field. To add a line break, the operator must hold down the Control key before pressing Enter.



Due to the lack of tickmarks, a data entry operator will not immediately know how many characters they have entered into the field and how many characters are remaining. CSEntry shows, on the status bar at the bottom of the screen, the number of characters entered, and the total size of the field.



For multiline controls, the characters \n will be placed in the field text for every line break. CSEntry will automatically interpret \n characters as linebreaks, but if you use your data with another software package, you may need to adjust for this behavior. In the above screenshot, the text would appear in the data file as:

```
100 Main Street, Suite 1234\nAnytown, NY 00000-0000
```

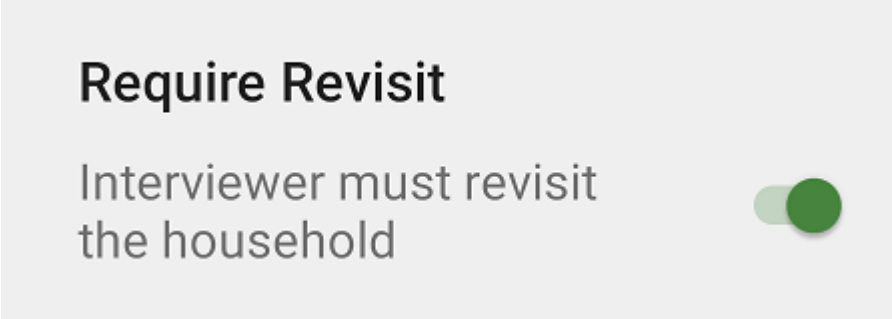
Both of these Text Box options can also be specified for alpha fields in rosters. To resize the field within a roster you may need to first adjust the roster's row heights or column widths.

See also: Capture Types, Number Pad

Toggle Button

The Toggle Button capture type allows an enumerator to select a single discrete value or to proceed with no selection. To use the capture type, define a single value set value for a numeric or alphanumeric field. If the user selects the Toggle Button, that value will be selected. If the user does not select the Toggle Button, then the field will be validated with a value of `notappl` for numeric fields and `""` for alphanumeric fields.

It is possible to specify labels for the two states (selected and not selected) if the value set contains two values with one value mapped to `notappl` or `""`. Although only the selected state label will appear during data entry, having the other value defined may be useful for other purposes (e.g., using the `invalueset` function or creating frequencies/tabulations).



See also: Capture Types, Radio Button

Multimedia

Multimedia Features

When collecting data, especially when conducting a CAPI census or survey, you can take advantage of several multimedia features available in CSPro. The following features are available when running CSEntry on Android devices:

Feature	Description
Audio	Record audio in the background or in an interactive mode controlled by an enumerator.
Barcode	Scan a barcode or QR code.
Camera	Take a photo and save it as an image.
Signature	Capture a signature and save it as an image.

See also: Capture Types, Media Store

Media Store

On Android, the Media Store is a mechanism for accessing audio, image, and video content that can be shared across applications. The media of the given type is not necessarily stored in a single directory. The media can all be accessed using the following identifiers:

- `Media.Audio`: Audio files.
- `Media.Images`: Image files.
- `Media.Video`: Video files.

You can use the `dirlist` function to retrieve a list of these files or you can use the `path.selectFile` function to show the operator a visual listing of these files.

Using any of the identifiers on Windows will result in an empty list of files.

See also: Multimedia Features

Camera

***Feature Upgrade**: Starting with CSPro 7.7, you should no longer use the `execsystem` method of taking photos, as it may soon be removed from CSPro. To replicate the behavior of the execsystem function call, use the `image.takePhoto` function.*

Passing the `camera` command to `execsystem` opens the camera widget on mobile devices. The camera UI includes two options:

- **Flip camera icon**: Tap to toggle between the front and rear-facing camera.
- **Camera icon**: Tap to take the photo.

When using `execsystem` to launch the camera, you can include an optional text string that will appear on the mobile

device's screen while the camera is active. The string will not be saved to the actual image file. Specify the text by following the image filename with a `"|"` and the overlay text.

Examples

```
execsystem("camera:" + pathconcat(Application, "photo.jpg"));

execsystem("camera:evaluation.png|Take a photo of your evaluation form.");
```

See also: Multimedia Features, ExecSystem Function (Mobile), Image.CaptureSignature Function

Signature

***Feature Upgrade**: Starting with CSPro 7.7, you should no longer use the `execsystem` method of capturing signatures, as it may soon be removed from CSPro. To replicate the behavior of the execsystem function call, use the `image.captureSignature` function.*

Passing the `signature` command to `execsystem` opens the signature widget on mobile devices (i.e., Android; it will have no effect on applications run under Windows). The signature UI includes two options:

- **Save**: Tap to save an image of the displayed signature.
- **Clear**: Tap to to clear the displayed signature.

When using `execsystem` to launch the signature widget, you can include an optional text string that will appear on the mobile device's screen while the signature screen is active. The string will not be saved to the actual image file. Specify the text by following the image filename with a `"|"` and the overlay text, as shown in the second example below.

Examples

```
execsystem("signature:" + pathconcat(Application, "saved-signature.jpg"));

execsystem("signature:consent.png|Please confirm that you have read the consent statement."));
```

See also: Multimedia Features, ExecSystem Function (Mobile), Image.CaptureSignature Function

CAPI Strategies

Forms

Form layout should be easy for the interviewer to see and understand. For this reason, it's usually best to place fewer items on each form that you might with a non-CAPI application. Also, remember that screen real estate is limited in a CAPI application, since question text will occupy a large portion (sometimes up to half) of the top of the entry window.

Consider breaking your application into a series of sections, each of which deals with a topic (earnings, fertility, etc.). One or more forms are then used for each section's questions.

Avoid scrolling in CAPI applications, since this prevents the interviewer from seeing the entire form (or roster). Also, the interview environment—laptop, maybe no mouse, possibly poor lighting—often makes scrolling cumbersome.

Also, it is good practice to gather the names of all household members at the start of the interview. Then the interview can cycle through each member. Doing this minimizes the risk of accidentally omitting household members.

Try to limit the size of your forms so that the interviewer won't have to scroll them. When developing your application, be sure to test your application using the screen resolution that will be on the laptops used in the field.

Fields

In CAPI data entry, consideration needs to be given to the properties of fields used in the application. The following table lists certain field properties and how they might be used:

Use enter key	This is usually selected, because otherwise the application will automatically advance to the next field, which might be confusing for the interviewer.
Mirror fields	These are very useful in CAPI applications, as they show the contents of fields in other forms. This can assist the interviewer.
Protected fields	CAPI applications use protected fields to show calculated or derived values. These provide useful feedback to the interviewer.
Force out-of-range	This is almost always turned off, since the interviewer should only be allowed to enter valid responses. However, you should consider including "not applicable", "don't know," and "refused" response categories.

CAPI applications frequently make greater use of alphanumeric responses than do traditional key-from-paper systems. This is because of the interactive nature of CAPI, and it also makes the interview more interactive (by capturing names, for example).

Questions

CSPro shows customized question text in the top part of the entry window. To help the interviewer distinguish between text for the interviewer and the respondent, consider using different color fonts. For example, things the interviewer should say to the respondent (for example, "How many children do you have?") might be shown in black. Instructions to the interviewer (for example, "Ask of each household member") might be shown in blue. CSPro includes customizeable styles which can facilitate this.

Question text can be customized using fills or text substitutions. These fills can contain CSPro logic expressions which reference the contents of variables, dictionary items, or user-defined functions. They are identified by the CSPro logic of the object surrounded by two consecutive tilde characters (~~). During the interview, the logic is evaluated and the value of the expression is substituted into the question text. For example, "How old was ~~FIRST_NAME~~ on January 1?" might be transformed into "How old was Edward on January 1?" Fills help customize the question text and frame it for

specific respondents.

Your CAPI application can also present different sets of question text based on conditions. For example, the text for certain questions might differ based on the number of persons in the household. Conditions can also let your application support multiple kinds of questions, depending on criteria you decide.

See also: Change Formatting

Organization of the Instrument

CAPI entry applications are almost always created in system controlled mode. because they take advantage of CSPro's ability to use logic to determine the question order.

To get started, it is a good idea to define sections containing sets of questions related to one topic. The number of sections will vary from survey to survey. You might need a section to begin the interview and a section to end the interview. You also might need some help sections with FAQ or roster of persons in household.

Next you need to develop an order of the sections, that is the order of the interview. Some sections like FAQ or person roster may be independent of the interview sequence, and may best be placed after the last subject matter section or after the end of the interview section.

Now with a section you need to define the order of questions. Place one item or several items which are related to one another on separate forms. The forms will be in the sequence the questions are asked.

Using Multiple Languages

CAPI respondents often speak different languages. In fact, it is not uncommon to find several languages spoken among the households included in a survey. It is advantageous for your entry application to accommodate the different languages that respondents might speak, since doing so will improve data quality and response rates.

CSPro supports multiple languages, and lets you add languages and define question text for each. By default, just one language (marked as English) is available. You can define additional languages and then enter question text for each as needed.

During the interview, the interviewer can switch among the application's languages on-the-fly. Each question's text will be displayed at the top of the screen, in the chosen language. So, if an interviewer arrives at a household and finds that the respondents prefer to conduct the interview in another language, this can be easily done.

When you develop a multi-language application, it is probably easiest to finalize and test all the question texts in one language. Then, once this is done, language specialists can translate them into the other languages your application will support. This is easy to do, since the question text editor can display question text for two languages at the same time. The translator can copy and paste text or bitmaps, if needed. Finalizing all the question text in one language first also helps avoid version control problems that might arise if things were translated then later modified.

Breaking Off the Interview

Sometimes an interview cannot be completed in one session. CSPro gives the developer control over how an interview can be terminated. You might want to include a form that asks when the respondent would prefer to do the follow-up visit.

CSPro fully supports partial save, which causes a case to be saved to disk even if it is not yet complete. The case can be continued at a later time. Partial save is a very useful feature in CAPI applications, and should be used if any of the interviews might not be completed in a single sitting. Very small applications, like listing operations or a short-form census, probably would not require partial save ability. To enable partial save, select the box in the data entry options dialog.

The user function OnStop can be used to trap the stop button (■) or attempts to exit the interview (for example, by pressing **Alt+F4**). The developer might want to include a form that manages the exit process and saves the partial case

to disk.

You could also use OnStop() to intervene and prevent an interview from stopping, or to jump to a callback form before exiting. Your application also could store the field being entered, so that it could jump to that section when the interview is later continued.

If it is necessary to return later to finish a CAPI interview, you might want to include a form that asks when the respondent would prefer to do the follow-up visit. In a production CAPI system, a separate case management system could take advantage of this information and assist the interviewer in tracking and scheduling appointments.

Coming Back Later

You need to think about what behavior will be needed if an interview is continued at a later time. That is, should they start again from the beginning of the instrument, the place where the last interview left off, or something else. Sometimes instruments will always go through a front section to confirm household member names and other vital information and then jump to the section where the earlier interview ended.

Prefilling Values

In a data entry application, it is possible to prefill fields using several approaches:

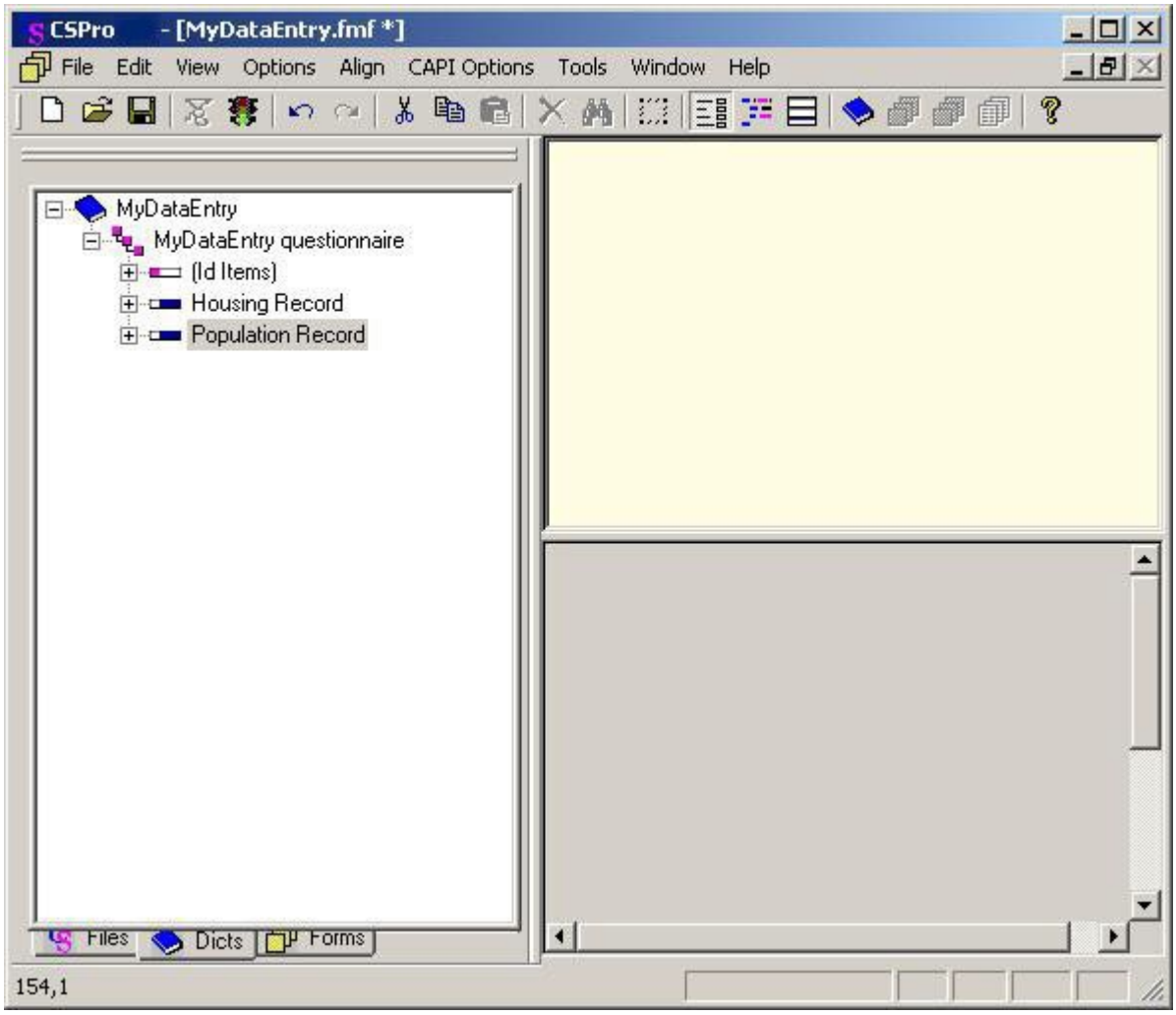
1. Make an ID field persistent, which uses the value entered for a previous case to prefill the value for a new case. The initial values of persistent fields can be specified in a PFF.
2. Make an ID field auto increment, which takes the value entered for a previous case and increments it by one to prefill the value for a new case. If no previous case has been entered, the value starts at 1.
3. Use the Key attribute of a PFF to specify the initial value of the case's ID items.
4. Use the Parameters section of a PFF to specify the initial values for non-repeating, non-persistent items.
5. Make a repeating field sequential, which automatically increments a field on a roster, incrementing it by one on each added occurrence.
6. Specify a value in the `preproc` of the field; for example:

```
PROC INTERVIEW_END_TIME

preproc

    INTERVIEW_END_TIME = timestamp();
```

How to ...

Create a New CAPI Application

1. Create a New Data Entry Application
2. Don't generate forms
3. From the options menu, select Data Entry Options.
4. Check the box "Use question text."
5. Check the box "Allow partial save", if you want to use partial save.



Notes:

- After enabling question text, the forms screen will be divided horizontally. The top part is for question text (to be read during the interview), the bottom is for the normal form contents.

Define Languages

A CAPI application can contain question text in multiple languages. The interviewer can switch among available languages as needed.

1. From the **CAPI** menu, select **Define CAPI Languages**. The CAPI Languages dialog box will be displayed,

showing the languages currently defined for the application.

2. You can add, remove, or modify the your application's CAPI languages.
3. You can also define multiple languages for your dictionary and message file.

Notes

- Language names follow the same rules as names; that is, they must be unique and cannot contain spaces. Try using abbreviations like EN (English), ES (Spanish), FR (French), or PT (Portuguese). We recommend using ISO 639-1 two digit codes. When those codes are used, on mobile the initial language will be set automatically based on the language in use on the mobile device.
- Language labels can contain any text to describe the language.
- During data entry, the interviewer can easily switch among languages.

See also: Create Helps for Fields

Organize Forms

Divide your CAPI questionnaire into sections, one for each topic area. Add any special sections, such a begin and end section or global help sections, like an FAQ. These sections organizing units for defining forms and movement within the application.

Break up the questions within a section into forms containing one or several responses. Remember, you only have about half of the screen at the bottom to use for a form, because the question text may take up the top half of the screen.

If the form contains a roster, you should make sure that only the roster scrolls, not the form. Scrolling forms can be confusing to interviewers.

Enter Question Text

CAPI applications include a question text window (at the top of the form). You can customize the text that appears in this window for each question in the application.

1. From the **View** menu, press **CAPI Questions** (or press **Ctrl+Q**). The question editor will be displayed.
2. Select an item from the tree on the left. Any text already defined for this item will be displayed in the editing window on the right.
3. Enter text into the editing window. You can format the text using the toolbar, or use the formatting choices shown in the **CAPI** menu.
4. When you have finished entering the question's text, you can select a different item or switch to the form view or logic view.

The question text editor supports copy and paste operations, including pasting formatted text from other documents.

See also: Change Formatting, Create Conditional Questions, Create Fills In Questions, HTML Question Text

Create Fills In Questions

Fills are used to customize question text based on respondents' specific characteristics. This is done by embedding CSPro logic expressions into the text and surrounding them with pairs of tilde characters (~~).

Fills may contain any valid CSPro logic expression that evaluates to a string or numeric value. Logic expressions may include dictionary items, variables, occurrence labels, user-defined functions, as well as any built-in CSPro functions and operators.

Fills with Dictionary Items or Variables

Enter some question text that needs to be customized. For words or phrases that need to be filled, enter a dictionary item or variable with ~~ characters before and after. For example:

> Can I speak with **~~FIRST_NAME~~** now?

When the entry system comes to this text, it will insert the person's FIRST_NAME value into the question text:

> Can I speak with **Marjorie** now?
> Can I speak with **Allyson** now?

When used with numeric values, the fill uses the value code. For example, ~~SEX~~ might be 1 or 2. If you want to use the value label, you can use the `getvaluelabel` function:

> How old was **~~FIRST_NAME~~** when they completed **~~getvaluelabel(HIGHEST_GRADE_COMPLETED)~~**?

Fills with Occurrence Labels

You can customize the question text by inserting an item or group's occurrence labels into the text. For example:

> How much did you spend on **~~getocclabel()~~** in the last month?

Fills with User-Defined Functions

You can insert the return value of a user-defined function into the question text. The function can return either numeric or string values. For example:

> Thinking now about **~~FIRST_NAME~~**, what is **~~SexPronoun()~~** age?

In logic, a user-defined function must be declared, as in:

```
function string SexPronoun()

    recode SEX -> SexPronoun;
           1   -> "his";
               -> "her";
    endrecode;

end;
```

HTML in Fills

You can use HTML tags in order to dynamically format fills using logic. If you use HTML tags, you must surround the fill with three tildes (~~~) instead of two. This tells CSPro not to escape the tags when substituting the fill value. If you use the standard two tildes ~~, the literal HTML will substituted instead of the formatted value.

For example the question text:

> You have entered the following household members:
>
> ~~~householdMembers~~~

with the following logic:

```
householdMembers = "<ul><li>Bouba</li><li>Frank</li><li>Chen</li></ul>";
```

will result in:

You have entered the following household members:

- Bouba
- Frank
- Chen

Or more realistically:

```
householdMembers = "<ul>";

do numeric ctr = 1 while ctr <= totocc(PERSON_REC)
    householdMembers = householdMembers + maketext("<li>%s</li>", encode(strip(NAME(ctr))));
enddo;

householdMembers = householdMembers + "</ul>"
```

See also: HTML Question Text

Edit Question Text Styles

Frequently, CAPI applications format question text using a few specific combinations fonts and colors. For example, 16 point Arial black for literal question text and bold 14 point blue for interviewer instructions. CSPro supports **styles** that can be defined and used for this purpose. Styles allow you to define standard formats that are shared between all questions in your data entry application.

By default, every CSPro application comes with the following default styles: normal, instruction, heading 1, heading 2 and heading 3. These styles appear in a dropdown list in the toolbar of the question text editor.

Applying Styles

To apply a style to text in the question editor:

1. Select all or a portion of the text in the question text editor.
2. Click on the styles dropdown in the toolbar and choose the style to apply.

Customizing Styles

You can modify the default styles or add your own styles:

1. Select **Edit Styles...** from the **CAPI** menu.
2. Select the style to modify from the list on left or click on the **+** button to add a new style to the list.
3. Change the font type, size, font formatting (underline, boldface, italic) and color as desired.
4. Click **OK** to save the changes.

All question text that uses any of the modified styles are automatically updated to reflect the changes. Any new styles created are added to the styles dropdown list in the question text editor.

Change Formatting

The question text editor allows you to apply fonts, colors, and formatting to your question text. You can either use the formatting controls on the toolbar or in the **CAPI** menu.

Question text formatting options include:

Edit Styles	Define custom styles to use in the question window.
Bold	Switch between **bold** and non-bold text.
Italic	Switch between *italicized* and non-italicized text.
Underline	Switch between underlined and non-underlined text.
Color	Change text color.
Left	Align text to left of question window.
Center	Align text to center of question window.
Right	Align text to right of question window.
Bullets	Convert text to a bulleted list.
Numbering	Convert text to a numbered list.
Insert Image	Insert an image into the question text.
Insert Table	insert a table into the question text in order to show information in a tabular format.
Insert Link	Insert a hyperlink into the question text.
Text Direction LTR	Mark the paragraph as for left-to-right text.
Text Direction RTL	Mark the paragraph as for right-to-left text.

For more advanced formatting it also possible to directly edit the HTML code for the question text. To view the HTML click on the **Code** button on the question text editor toolbar.

Add Images

Pictures and other graphics can be added to your question text. Most common image formats are supported, including JPEG, PNG, GIF, and Bitmap formats. You can insert an image by:

1. Clicking on the **Insert Image** on the toolbar in the question text editor.
2. Selecting **Insert Image** from the **CAPI** menu.
3. Right-clicking inside the question window and choosing **Insert Image**.
4. Using the code view and adding an HTML image tag.

Image File Storage

By default, when you insert an image, the image file is added as a resource file. A copy of the image will be placed in a directory named **Resources/Images** that is added to your application. When you generate a pen file the image will be added to the pen file. This is the simplest way to ensure that all images are copied to the interviewers' devices when you deploy your application.

If you wish to manage image storage yourself, uncheck the box **Include in pen file as image resource?** when choosing the image file. When this box is not checked, CSPro does not make a copy of the image file but instead stores the relative path from your application directory to the folder that contains the image file. This means that if you move or copy the application you will also need to move/copy the image file in such a way that the relative path does not change. In general, it is easier to allow CSPro to include image files as resources. However, if you want to share the same image file among multiple applications you may consider managing the image file locations yourself.

Adding high-resolution images, such as digital photos, will increase the size of your application which may make it slow to install and update the application over the internet. You should consider resizing large images using an image editor before adding them to the question text.

Insert Link In Question Text

To insert a hyperlink into question text, click on the **Link** button on the question text toolbar. A dialog will appear with two options:

- **Text to display**: The text that will appear in the question text.

- **URL**: The URL that should be accessed if a user clicks on the linked text.

See also: Change Formatting

HTML Question Text

Overview

For those with experience using HTML, it is possible to format the question using HTML tags and CSS styles. To view the HTML code, click on the **Code** button on the question text editor toolbar.

CSPro supports all tags and attributes in standard HTML5, including images, tables, and CSS styles.

The HTML for each question does not include the <html> or <body> tags. The HTML question text is inserted into to a complete HTML document that already has those tags.

Styles

All styles defined in the Edit Styles dialog are included as CSS classes in the HTML document by CSPro. These styles can therefore be used in the HTML question text by setting the class attribute of a tag to the name of one of the styles. For example:

```
<p>A question with <span class="instruction">a style applied to this part</span>.</p>
```

The **normal** style is applied to the <body> tag by CSPro so that all unstyled text in the question text uses the **normal** style.

In addition, the document created by CSPro sets the background color to yellow on Windows Desktop and to gray on Android.

Finally the document adds some basic styling to tables such as borders and padding.

External Files

References to external files such as images and stylesheets may be included in the HTML question text. All paths must be relative paths based on the location of the application (.ent or .pen) file. Absolute paths are not supported. When deploying your application, make sure to copy the external files as well or include them in a resource folder.

JavaScript Interface

A JavaScript interface exists to facilitate the invocation of user-defined functions from question text. This feature allows advanced users to take advantage of the power of JavaScript to allow some interaction between question text and CSPro logic. For example:

```
<p>You are entering person ~~curocc()~~. When all persons have been recorded,
<a href="javascript:CSPro.runLogicAsync('EndPersonRoster();');">click here</a>.</p>
```

See also: HTML in CSPro, Change Formatting, Question File (.qsf)

Question Text Macros

There are a few tasks that you can perform on a data entry application's question text using the question text macros functionality. To access these tasks, either right-click on a form file in the tree and select **Question Text Macros** or, with the form active, select **Question Text Macros** from the **CAPI** menu. A dialog box will appear with the following options:

Audit Undefined Text: This creates a list of fields and blocks that do not have any question text. For most applications, you will want to define question text for all fields and blocks that an enumerator will visit. This option shows you which entities may require question text.

Remove Unused Text: If you define question text for a field or block but then remove that entity from a form, its question text may remain in the .qsf file. This option will remove any unused—not connected to a form entity—question text from that file.

Harmonize Occurrences: Increases the maximum occurrence number of a field or block's question text to match that of the controlling record in the dictionary. This is useful if you increase the number of record occurrences; rather than having to manually increase each question's occurrences values, you can use this option.

Initialize as Dictionary Label: For any field that does not have question text defined, the question text will be set to the field's item dictionary label.

Paste from Clipboard: Using one of the two formats specified, a field or block's question text will be set to the text specified on the clipboard. The allowable formats are:

```
NAME<tab>Question Text
NAME<tab>LANGUAGE_NAME<tab>Question Text
```

See also: Dictionary Macros

Resource Folders

There may be times, when deploying an application, especially to a mobile device, that you will want to include some auxiliary files that the application uses. Such files might include lookup files or value set images. To simplify the deployment of such an application, these files can be packaged in with the data entry .pen file. The files will be compressed in the .pen file and then decompressed when the user opens the application. CSEntry will recreate the folder structure and files contained in the resource folders. Files that already exist on the device will be overwritten only if the timestamp of the .pen file is newer than the timestamp of the file on the device.

To specify a resource folder, select File -> Add Files and then select a resource folder. Your application may have more than one resource folder. All files in the resource folder, and its subfolders, will be included in the .pen file, so it is important to think about how to structure your data entry application so that only relevant files are bundled together in the .pen file. You might, for example, have an Images folder where you put all the value set images that your application uses. When decompressing data on mobile devices, files will only be saved in the csentry folder or in any of its subfolders.

If you no longer want to mark a folder as a resource folder, select File -> Drop Files and select the folder that you no longer want to use.

See also: Lookup Files, Value Set Images

Use Multiple Languages

If your application supports multiple languages, then two CAPI question text windows will be shown. Use the drop down list in the toolbar above each question text window to select a language. Each question can have text for each language, although this is not required.

If your CAPI application supports multiple languages, consider starting by entering text for all the questions in the first language. Then use can translate each text into the other languages by putting the first language in the top question window, and translating in the bottom question window. You can also copy and paste between question text windows.

Create Conditional Questions

By using conditions, it is possible to determine which question to display at runtime based on previous responses.

The bottom window of the question editor is used to add conditions.

Select an item for which you want to add a condition (from the left-hand tree).

1. Right-click on the blue highlight bar in the condition window.
2. Choose Add Condition.
3. Enter a condition.
4. Enter text for this question in the window above.
5. Add more conditions, and then question text for each.

When the entry application comes to this question, it will evaluate the first condition. If it is true, then the question text for that condition will be shown to the user. If not, the next condition will be checked, and so on. If no conditions are satisfied (i.e., they are all false), then no question text will be displayed. It is not necessary to add an actual condition; a blank condition will always evaluate true.

Note: Conditions can be any CSPro logic expression that evaluates to a numeric value including dictionary items or variables, and functions. Conditions may be combined using logic operators such as `and` and `or`.

For example, to display different question text for a variable in the first row of the roster than in the remaining rows of the roster simply set the first condition to `curocc() = 1` and leave the second condition blank. When the application is on the first row of the roster where the occurrence number is equal to one, the first question text will be shown. On subsequent rows, where the current occurrence is greater than one, the first condition will be false so the second question text will be shown.

Structure Movement

Decide what movements within the case, an interviewer will be allowed to make. For example, will the interviewer only be allowed to move backwards or forward through the interview questions or will they be allowed to jump around. If they can jump around, what jumping movements will they be allowed to make. Are there sections that must be completed first? Are there optional sections that don't have to be filled in? Are there sections that can be completed in any order?

Decide what happened when the interviewer tries to stop the interview before it is completed? Will this be allowed? If it is, how will the interview be ended? Through an end section, which may collect when the respondent will be free to continue the interview and where interview notes may be inspected? Or by just remembering the current question being asked?

If an interview can be continued at a later time, how will the interview be restarted? By just advancing to the last question completed? Or by having the interviewer ask a few questions to reorient the interviewer and respondent.

Create Helps for Fields

In addition to question text, a CAPI application can have help information for each question. During entry, the **F2** key is used to display a question's help text.

1. From the **View** menu, press **CAPI Questions** (or press **Ctrl+Q**). The question editor will be displayed.
2. Select an item from the tree on the left. Any text already defined for this item will be displayed in the editing window on the right.
3. Switch from editing questions to editing helps by pressing the ? button.
4. Enter help text into the editing window. You can format the text using the toolbar, or use the formatting choices shown in the **CAPI** menu.

When you have finished entering the question's text, you can select a different item or switch to the form view or logic view.

Show Values for Selection

You can display a list of response values for a field, so that the interviewer can select a response. The list of response values comes from the first value set of the field in the data dictionary.

To turn on the display of values for selection during the interview, use the following statement:

```
set attributes(field-1[, field-2, ...]) assisted on;
```

The interviewer can turn the responses on or off during the interview by pressing **Ctrl+C** or going to the options menu and selecting **Show Responses (This Field)**.

Handle Multiple Answers

Some questions allow for multiple responses. For example a question such as "Which of the following items do you have in your household?" allows for multiple responses.

There are three methods that could be used to collect this data in a CAPI application. Two involve defining in the dictionary a data item with multiple occurrences. For example if there are 10 possible responses, then the item, usually a single character, would occur 10 items. On the data entry form, the item would be displayed as a roster with 10 occurrences. Response labels can be given on the form by changing the occurrence labels from numbers from 1 to 10 to the text of the responses, so the enumerator will know, and can read the possible responses.

One way to collect the responses is to create a roster with free movement, or spreadsheet like behavior, so that enumerator can move the cursor between occurrences to mark the ones to which the respondent gives positive answers.

Another way to collect the data is to protect the roster and have a separate data item to collect the number of the response. When the number is entered, the corresponding occurrence of the item is marked. If the number is entered again the corresponding occurrence is unmarked.

A final way to collect the data is to set the item as an alpha item and use a check box to collect data for the item.

Choose Topic Sections

Often in a CAPI questionnaire, the enumerator needs to be able to move from one section of the questionnaire to another.

To make this happen, a global navigation key needs to be established and handled by the onkey function. The `accept` function can be used to collect the enumerators choice of what section to move to and define the first field in the section to move to. Movement, however, must be made from a location, a field, before any of the sections are encountered in order maintain the path structure. This field is protected so that no data need to be entered into it.

Below is code for the onkey function to support using F9 to move to another section in the application. The NAVIGATE data item provides the location of movement.

```
function numeric OnKey(numeric keystroke)

    if keystroke = 120 then // F9 to navigate among sections

        numeric selection = accept( "What Section?",
                                    "Section A",
                                    "Section B",
                                    "Section C",
                                    "Section D",
                                    "Section E");

        if selection = 1 then // Section A
            JumpField = "Q_A1";
        elseif selection = 2 then // Section B
            JumpField = "Q_B1";
        elseif selection = 3 then // Section C
            JumpField = "Q_C1";
        elseif selection = 4 then // Section D
            JumpField = "Q_D1";
        elseif selection = 5 then // Section E
            JumpField = "Q_E1";
        else // Escape
            OnKey = 0;
            exit;
        endif;

        JumpFlag = 1;
        move to NAVIGATE;

    else
        OnKey = keystroke;

    endif;

end;


PROC NAVIGATE

onfocus

    $ = "Z"; // need to have a value in the field

postproc

    if JumpFlag = 1 then
        JumpFlag = 0;
        move to JumpField;
    endif;
```

Create General Helps

Often a specialized general help is needed for a CAPI application. Such a help can be established in the following manner.

1. Put a form with one alpha item at the end of all forms.
2. Make question text for the item be the help text.
3. Code in the OnKey function a key that will jump to the help form.
4. Code the alpha item on the help form that will be used to jump back.

In the code below, **Ctrl+H** is used to jump to the general help. The data item `X_HELP` is on the form with the help text.

```
PROC GLOBAL

string lastField;

function numeric OnKey(numeric keystroke)

    string thisField = getsymbol(); // current field location

    if keystroke = 2072 and thisField <> "X_HELP" then // Ctrl+H
        lastField = thisField;
        move to X_HELP;
    endif;

    OnKey = keystroke;

end;


PROC X_HELP

    reenter lastField;
```

Test Application

CAPI applications require even more testing than data entry applications from paper forms because of the enumerator's need to be able to move easily and quickly around the questionnaire.

Here are some suggestions on how to test an application.

- Go through the case in intended sequence to test skip patterns, range checks, and edit checks.
- If breaking off and restarting the interview are allowed, try doing that from different locations. Can you get back to where you left off easily?
- Test moving around within the application, moving backward and forward.
- Test moving between sections, if that is allowed.
- Test field helps and general helps.

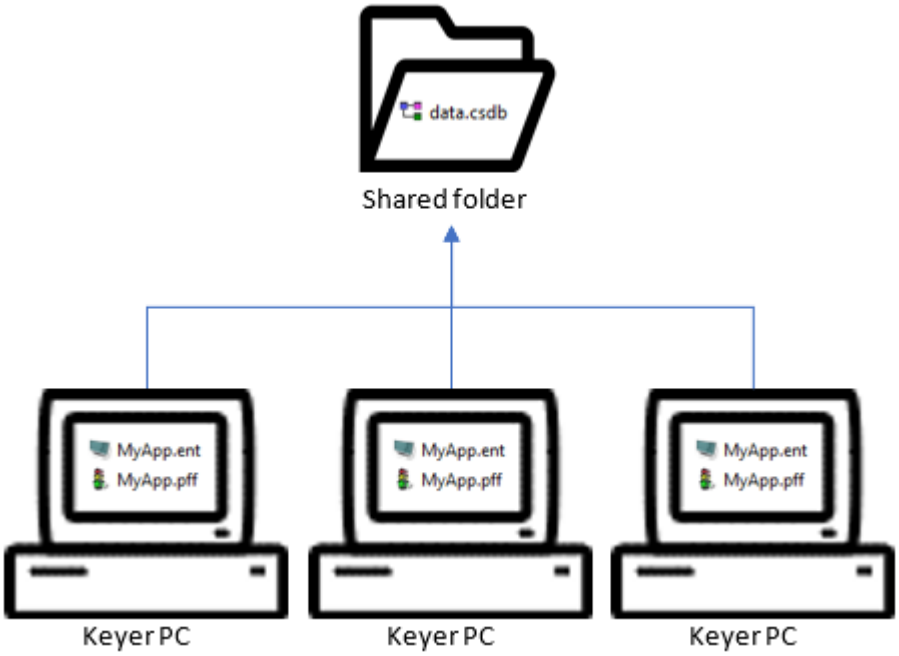
Network Data Entry

Network Data Entry

When keying large numbers of questionnaires using a team, it is possible to take advantage of a local area network (LAN) to make it easier to manage data files. There are multiple options for using a LAN to facilitate data entry. In all cases, CSPro must be installed on each of the computers that will be used for keying.

Using a Shared Data File on a Network Drive

It is possible to have multiple keyers, each using their own computer, key to a single data file that is located on a shared network folder. When launching the data entry application, enter or browse to the path on the network drive where you want to save the data. After having done this once, the path to the data file will be saved in a PFF file. You can then place a copy of the data entry application and the PFF file on each computer and have the keyers run the PFF file to start data entry. All keyers will enter data into the same shared data file.



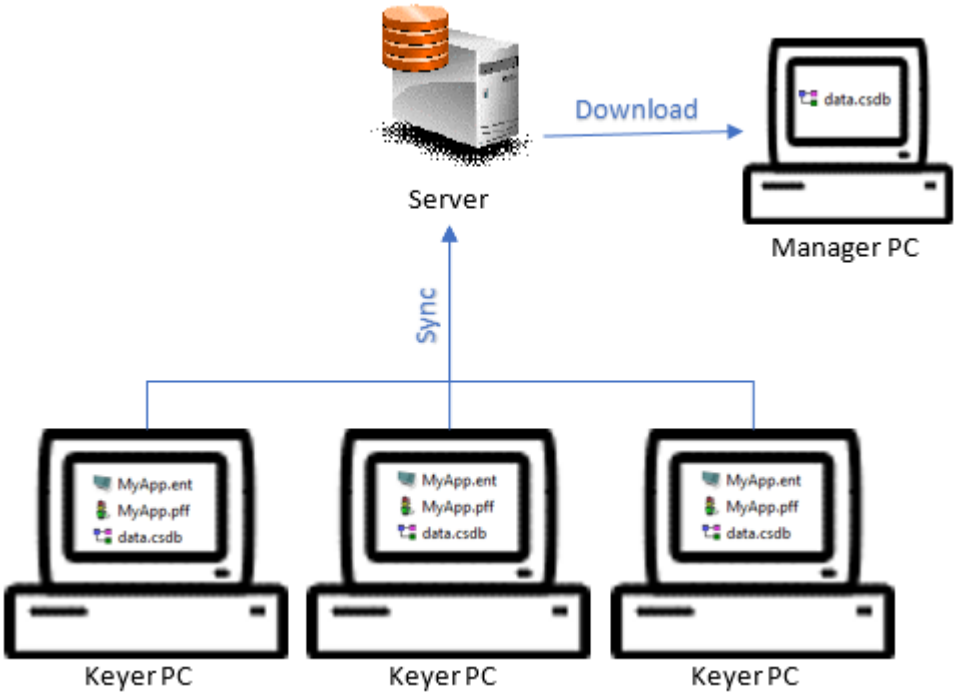
Sharing a single file between multiple simultaneous users is only possible when using non-text files such as CSPro DB files. Data files in text format cannot be shared.

By default, each keyer will be able to view, modify, and verify cases in the shared data file entered by other keyers. This can be restricted by setting the CaseListingFilter in the PFF filter. See Run Production Data Entry for more information.

Using Data Synchronization on the Local Network

CSPro supports the transfer of data files between computers using the synchronization feature. This can be used to transfer data between the keyers' computers and a server. Normally synchronization is used to transfer data over the Internet to a central server, but it can also be used with a server that is connected only to the local network.

When using synchronization, each keyer enters data to a separate data file on their computer. After entering data, they use CSEntry's synchronize functionality to transfer the data that they keyed over the network to a server. The combined data from all keyers can then be downloaded from the server.



CSPro synchronization supports three types of servers: CSWeb, Dropbox, and FTP, any of which can be used for network data entry. CSWeb is a web server based on Apache, PHP, and MySQL. It can be run on any PC running Windows or Linux. For information on how to set up and configure CSWeb, see the see the CSWeb help documentation. If you are not familiar with setting up and managing web servers, then it may be easier to use Dropbox or FTP instead. Dropbox does not require a server at all. It simply requires a free account on Dropbox. However, it does require a reliable Internet connection since files are stored on servers in the cloud. There are various free FTP server programs that you can use to set up an FTP server on your local network, including FileZilla Server and Microsoft IIS FTP Server. More information about these options can be found online. Note that for network data entry, neither CSWeb nor FTP require a powerful server nor do they need to be connected to the Internet. They can both be run on a standard computer connected to the local network.

To enable synchronization in your data entry application, use the Simple Synchronizations documentation. After entering data, keyers select **Synchronize** from the **File** menu in CSEntry to send data to the server. To download data from the server, use the Download function of the Data Viewer tool. This will automatically combine the data from all the keyers into a single data file that can be used for further processing.

Data synchronization is only possible when using the CSPro DB and Encrypted CSPro DB file formats. Data files in text format cannot be synchronized.

Copying Data Files between Computers and Network Drives

In older versions of CSPro that did not support data synchronization or sharing files over a network, it was common to have each keyer enter data into a file on their computer and then use either a CSPro program or a DOS batch script to copy files from the keyers' machines to a shared network folder. Following that copying, the Concatenate Data tool was used to combine the keyers' files into a single data file for further processing. While this is still possible, it is more complex and error prone than the above options.

Android Data Entry

Android Development

Overview

CSEntry, CSPro's data collection tool, is available as an Android application from the Google Play store. This application can be used to run data entry applications on devices such as phones or tablets running Android version 4.0 (Ice Cream Sandwich) or higher. CSPro data entry applications can be designed on Windows machines and then run on either Windows or Android devices. This is a "build once, deploy many" system, whereby the running behavior across the two operating systems is very similar, so you have the freedom to decide what device you want to use to conduct CAPI surveys using CSEntry.

In the Android environment you will only be able to run binary data entry applications (.pen files). You will not be able to execute .ent files or any other kind of CSPro application (concatenation, batch processing, etc.). This .pen file, along with the .pff runtime script, constitute the two files needed to run a data entry application on an Android device.

All applications on the Android device must be stored in the *csentry* folder on your device. Connect your Android device to your computer and, via USB transfer, copy your .pen and .pff files to the Android device. Generally the device's external storage is its SD card. The first time you run the CSEntry application, the *csentry* folder will be automatically created for you. The application searches subdirectories located in the *csentry* folder, so it is possible to add multiple applications while maintaining a clean directory structure.

Starting with CSPro version 7.5, the *csentry* directory is located at **<external storage>/Android/data/gov.census.cspro.csentry/files/csentry**. In earlier versions it was located at **<external storage>/csentry**. If you previously had an older version of CSPro on your Android device, the *csentry* will not be moved when you upgrade.

You can also transfer applications to your device by using the Deploy Application tool.

Data Security

As in the Windows environment, the Android application does not encrypt data files by default. Fortunately, it is easy to encrypt an Android device so that all files stored on it are encrypted. On your Android device:

1. Go to Settings.
2. Under the Personal group of options, select Security.
3. Select the Encrypt Phone option and follow the subsequent instructions.

Encrypting the phone will require that you supply a PIN or password to protect the data on the phone.

Differences Between Windows and Android

The Android application only displays one question per screen (unless using blocks). A future release may allow the displaying of rosters and multiple fields on a single screen. In addition, though most functions operate identically on the Windows and Android platforms, there are a few functions that behave differently, such as `execsystem`. Remarks about differing functionality are noted in the help documents about each function.

See also: Android Limitations, Multimedia Features

Android Limitations

As much as possible, CSEntry and other CSPro programs run in the same fashion on Android devices as they do on Windows machines. However, there are some differences:

CSEntry (Data Entry)

- Only one question per screen is shown at a time (unless using blocks).
- Only .pen files, not .ent files, can be executed.
- The execsystem function has different command options.
- Only one instance of a data entry application can be open at any given time (so the `execpff` function has different behavior).
- Some other logic functions behave differently, with remarks about differing functionality noted in the help documents about each function.

Index Data

- The interactive modes (*View duplicate cases* and *Prompt to delete duplicates*) are not supported.

Other Programs

The following tools do not have any differences: Compare Data, Concatenate Data, Paradata Concatenator, Reformat Data, and Sort Data. However, only production PFFs can be executed on Android.

All other programs or tools that make up the CSPro suite are not supported on Android.

See also: Android Development

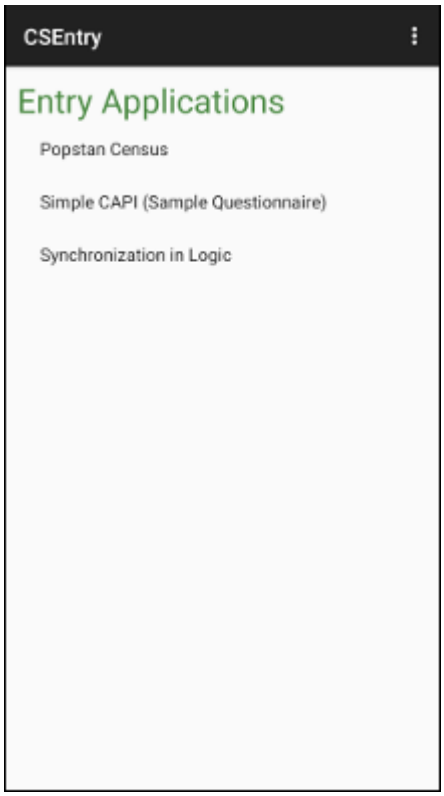
Android Navigation

Overview

The CSEntry for Android application is designed to be straightforward to use, with features and options that interviewers can easily control.

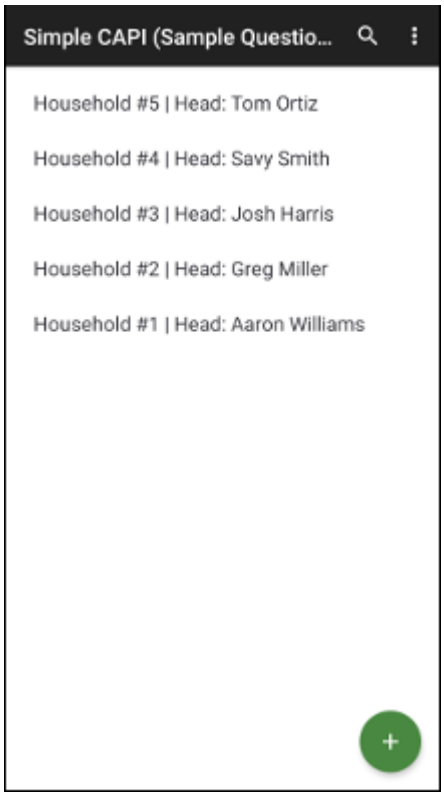
Entry Applications Screen

Upon opening CSEntry, all data entry applications on the device are displayed. Click on an application to open it.



Case Listing Screen

The case listing displays all cases that have already been added to the data file. Clicking on a case will open it for modification. Clicking on the add icon (green circle with a plus), will add a new case. To delete a case, hard-press on the case until a deletion menu appears. The menu on this screen has options to sort the cases alphabetically, or to display only incomplete cases.

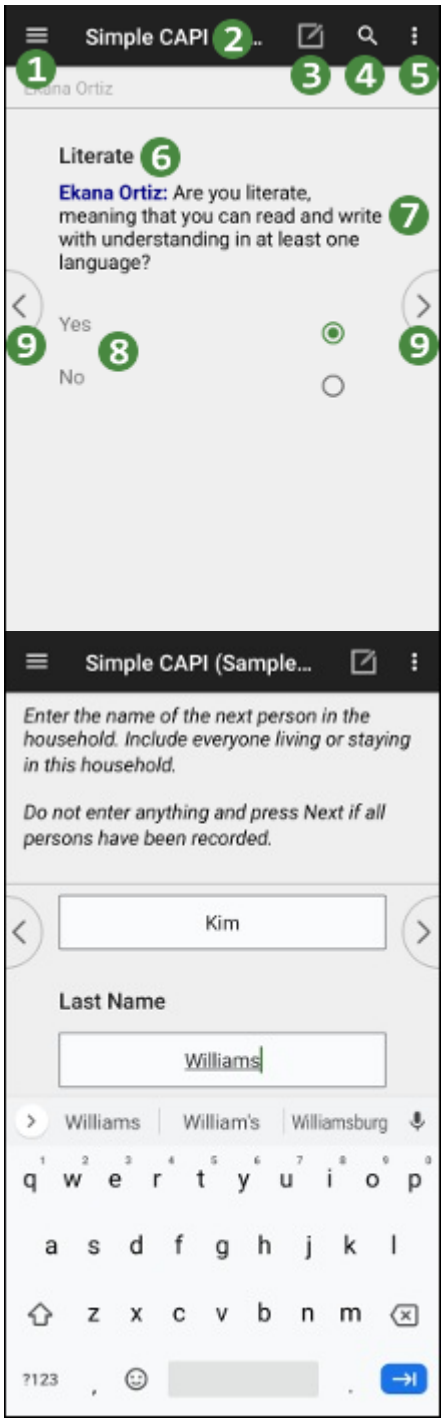


Screen Elements

This illustration showcases the features of the data entry screen.

1. Clicking on the menu icon displays the case tree.
2. This is the name of the survey.
3. Clicking on the pencil and paper icon allows field-specific notes to be taken.

4. Clicking on the search icon and typing in a query allows you to filter the response listing.
5. Clicking on the more options icon displays a menu with additional options.
6. The field label is a short description of the field that you are currently entering.
7. This is the question text for the field that you are currently entering. This will be more descriptive than the field label.
8. These are the response labels. CSEntry can display text boxes, radio buttons, check boxes, combo boxes, or date selectors.
9. These are the left and right navigation buttons, which correspond to moving backwards or forwards in the data entry application. Alternatively, swipe left or right to move backwards or forwards. Clicking on the Android back button will close out of adding or modifying a case.

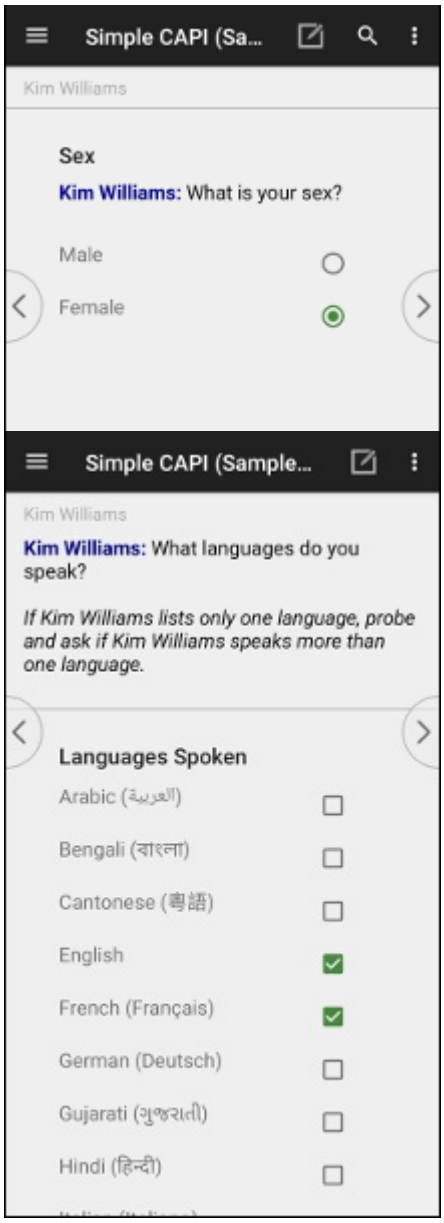


User Inputs: Text Box

To enter data in a text box field, simply type the response using the keyboard. If a keyboard does not appear automatically, click on the field and a keyboard will appear. If your keyboard has a Next key, you can click on it to move to the next field.

User Inputs: Radio Buttons

When presented with a list of radio buttons, you must select one, and only one, response. Click on the response label, or the corresponding radio button, to make your selection.

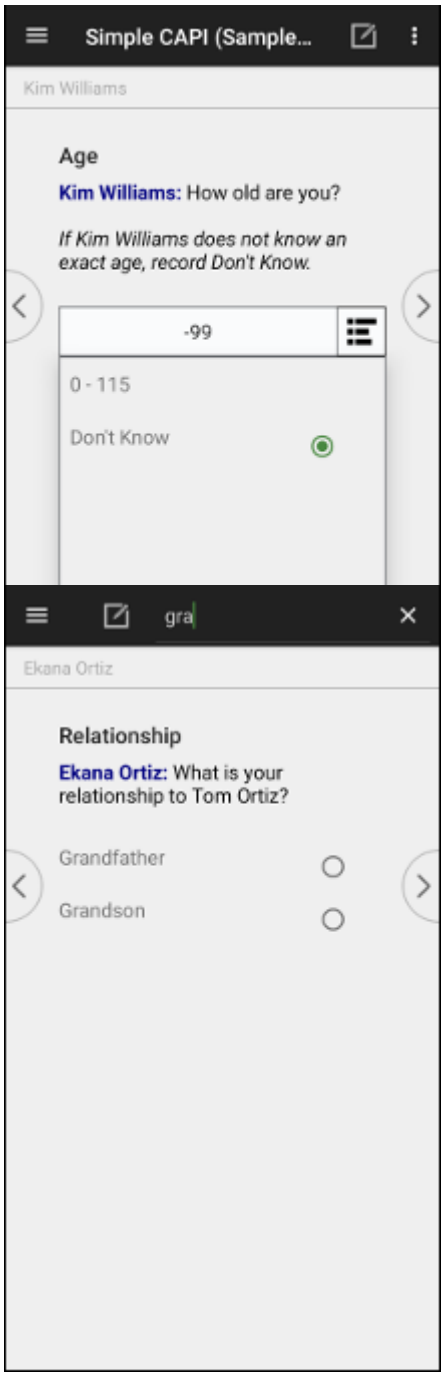


User Inputs: Check Boxes

When shown checkboxes, you can select none, one, or multiple responses. Click on the response label, or the corresponding checkbox, to make a selection. Once you are finished making all selections, you can proceed to the next field.

User Inputs: Combo Box

With a combo box, you can either enter a value within the defined range (as in a text box field), or choose a discrete value (as with radio buttons). To bring up the discrete values, click on the button to the right of the text field.



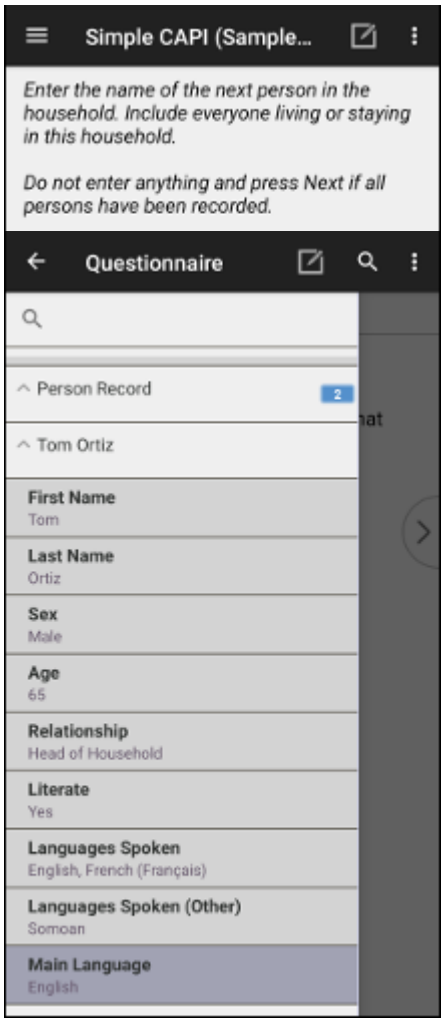
Search Filter

After clicking on the search icon, a field will appear where you can type a search query. The list of responses will automatically filter based on your search query. The full text of each response is searched, not necessarily starting from the first letter of the response.

Field Notes

After clicking on the pencil and paper icon, you will have the opportunity to type a field-specific note. Click the icon again to close the note.

Case Tree



Clicking on the menu icon brings up the case tree. The case tree displays all fields that have been entered in the data entry application, showing the field label as well as the field response. If you click on a field, you will be taken to that field. This allows you to quickly move from one part of the questionnaire to another.

To insert or delete a repeating occurrence, hard-press on the occurrence label until you are presented with a menu that allows for insertions and deletions. You can dismiss the case tree by using the Android back button.

CSEntry Settings Modification

It is possible to customize the menus presented to the user when running CSEntry on an Android device. By using the savesetting function to save a setting, you can override the default behavior of a menu option. You must execute this function call in logic before the menu setting takes effect, so if you are interested in immediately applying your customized settings, you can distribute your application with the PFF's CommonStore attribute set to a settings file with your predefined settings.

Settings List and Description

{ General Options }

"CSEntry.Setting.HelpUrl"

A website URL or filename that is opened when the operator clicks on the "Help" menu.

"CSEntry.Menu.Help"

Whether to show the "Help" menu option. Values: "Yes" (show) or "No" (hide)

{ Application Listing Options }

"CSEntry.Setting.LaunchSingleApplicationAutomatically"

Whether CSEntry should automatically open an application if there is only one PFF on the device. Values: "Yes" (open) or "No" (do not open)

"CSEntry.Menu.AddApplication"

Whether to show the "Add Application" menu option. Values: "Yes" (show) or "No" (hide)

"CSEntry.Menu.Settings"

Whether to show the "Settings" menu option. Values: "Yes" (show) or "No" (hide)

{ Settings Options }

"CSEntry.Menu.ShowHiddenApplications"
Whether to show the "Show Hidden Applications" option in the settings. Values: "Yes" (show) or "No" (hide)

{ Data Entry Options }

"CSEntry.Menu.AdvanceToEnd"
Whether to show the "Advance to End" menu option. Values: "Yes" (show) or "No" (hide)

"CSEntry.Menu.ReviewAllNotes"
Whether to show the "Review All Notes" menu option. Values: "Yes" (show) or "No" (hide)

"CSEntry.Menu.ShowCaseTree"
Whether to show the "Show Case Tree" menu option on tablets. Values: "Yes" (show) or "No" (hide)

"CSEntry.Menu.ShowRefusals"
Whether to show the "Show Refusal Options" menu option. Values: "Yes" (show) or "No" (hide)

"CSEntry.Setting.ShowNavigationControls"
Whether to display the navigation controls. If this value is set, then the "Show/Hide Navigation Controls" menu option is hidden. Values: "Yes" (show) or "No" (hide)

{ Case Tree Options }

"CSEntry.Menu.AddInsertOccurrence"
Whether to show the "Add Occurrence" and "Insert Occurrence" menu options when clicking on a group. Values: "Yes" (show) or "No" (hide)

"CSEntry.Menu.DeleteOccurrence"
Whether to show the "Delete Occurrence" menu option when clicking on a group. Values: "Yes" (show) or "No" (hide)

"Setting.ShowCaseTreeInOverlay"
Whether to always show the case tree (as opposed to it being displayed only on user demand). Values: "BasedOnScreenSize" (generally yes for tablets, no for phones), "Yes" (always), or "No" (hidden until user demand)

Examples

```
// prevent the user from adding new applications
savesetting("CSEntry.Menu.AddApplication", "No");

// override the help menu so that it opens the survey documentation
savesetting("CSEntry.Setting.HelpUrl", pathname(Application) + "Enumerator Manual.pdf");
```

See also: LoadSetting Function, SaveSetting Function, User and Configuration Settings

Batch Editing Applications

Introduction to Batch Editing

Over the years, the questions of whether or not to edit data, and if so, how, have generated a great deal of discussion and even controversy. Today, it is generally recognized that data from any survey or census will require at least minimal editing (and correction) in order to be usable. The following sections present a more detailed discussion of these questions, and the tools available in CSPro to carry out these tasks.

The Batch Edit Designer module allows you to create and modify batch edit applications. A batch edit application is used to clean (via editing and imputation) your data files.

For examples and methodology on how to develop your edit routines, refer to the United Nations Handbook on Population and Housing Census Edits.

This section contains the following information:

- Create a New Batch Edit Application
- Order of Executing Batch Edit Events
- Methods of Correcting Data
- Manipulate Automatic Reports
- General Batch Editing Issues

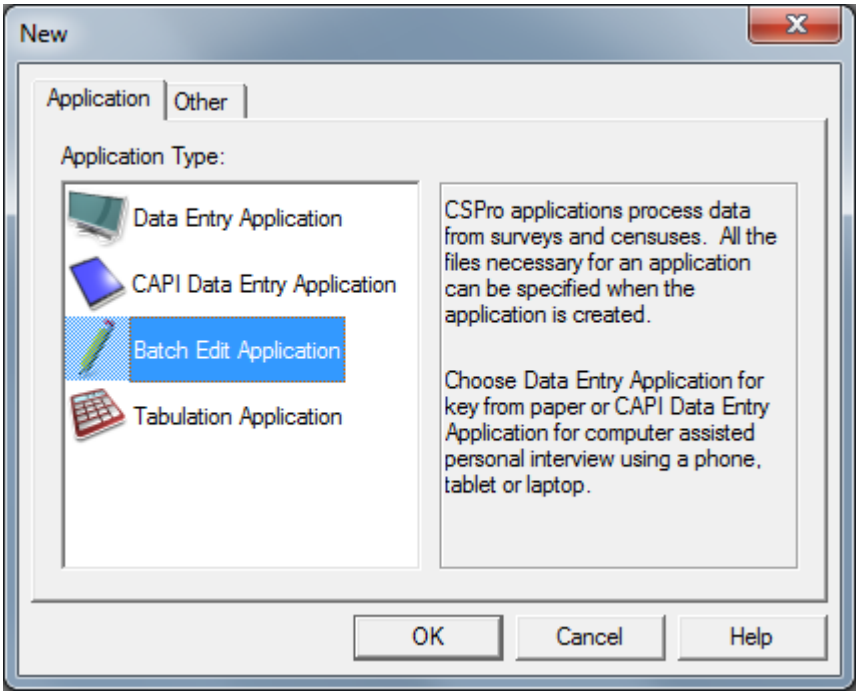
Create a Batch Edit Application

Create a New Batch Edit Application

To perform batch editing, you will need a data dictionary to describe the data file you are editing. If you already have a data dictionary for the file, you can specify this dictionary when you create a new batch edit application.

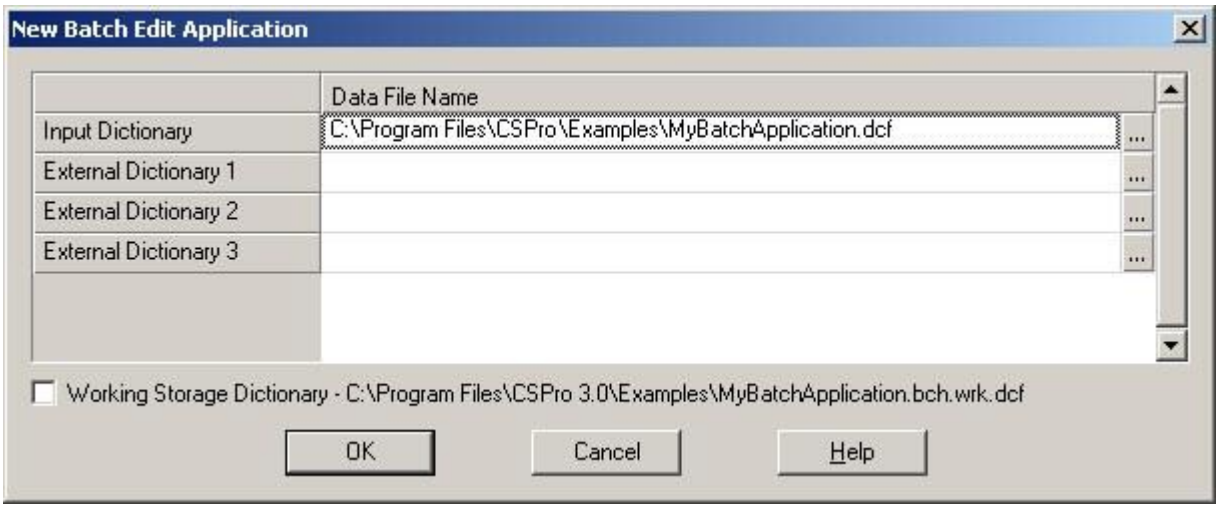
To create a new batch edit application:

• Click on the toolbar, or from the **File** menu, select **New**. The following dialog box will appear.



• Select **Batch Edit Application** and press **OK**.

• A file dialog box will appear. Enter the name of the application file. Make sure you are located in the folder where you want to place the application files. Then press **Create**. The following dialog box will appear.

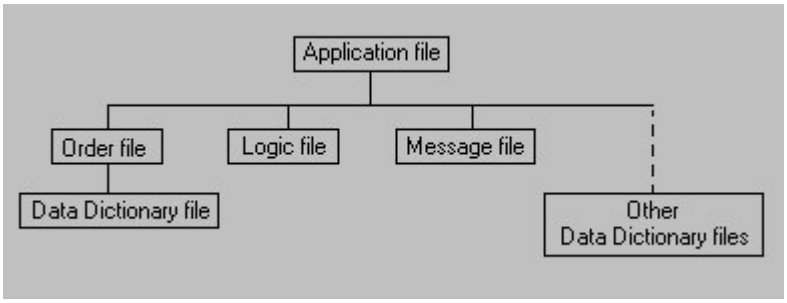


• A default name of the data dictionary describing the data file is given. You can use this name or change it. If you give the name of a dictionary file that already exists, that data dictionary will be used by the application. If you give the

name of a dictionary that does not exist, a new data dictionary will be created.

If you are using an existing CSPro data dictionary, you may begin creating batch edit procedures. If you are creating a new CSPro data dictionary, you will need to enter information into the dictionary about records, items, and values before you can create edits.

Batch edit applications consist of the following files:



- Batch Edit Application File (.bch)
  Specifies all other files contained in the application and includes other application information.

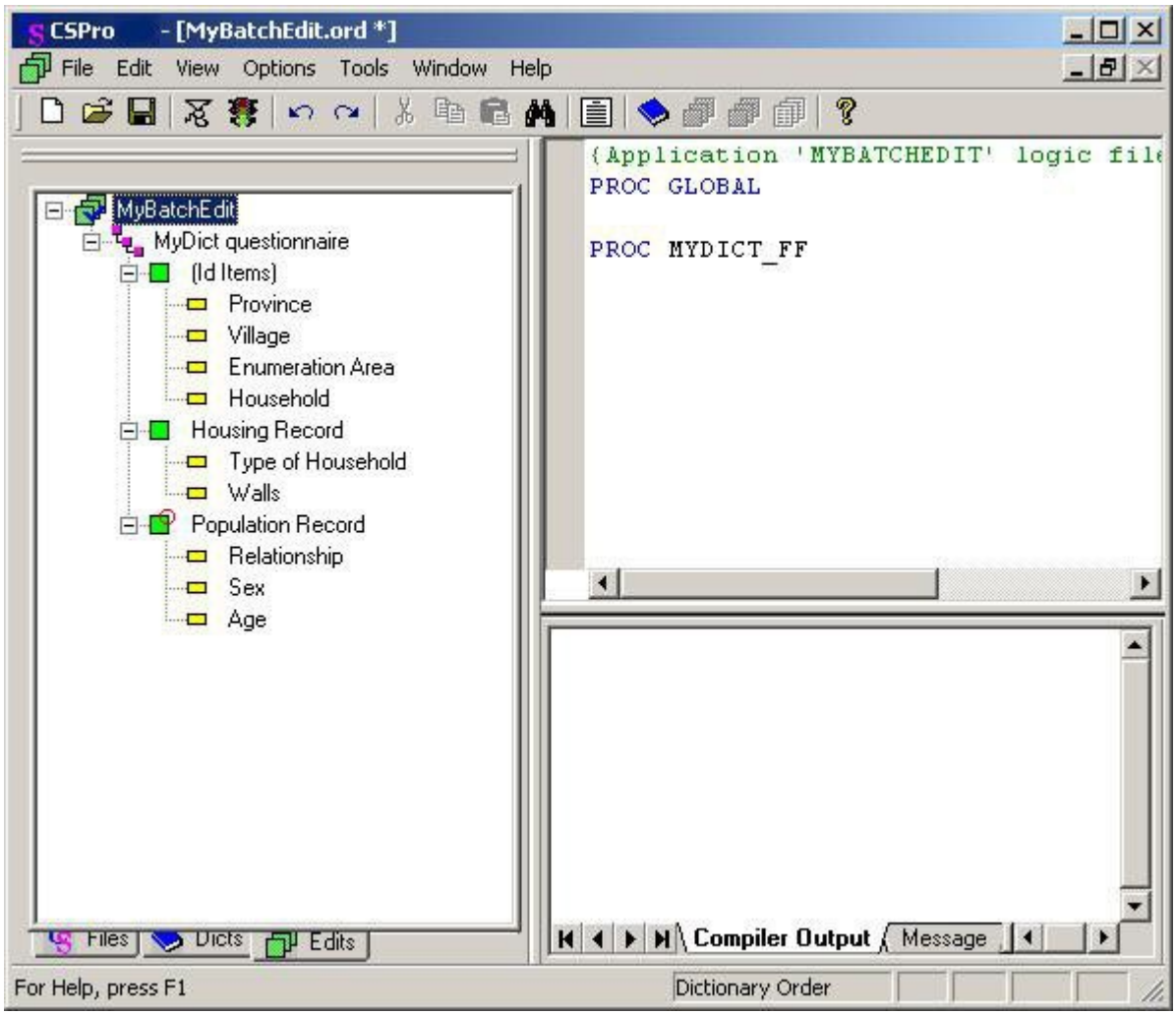
- Edit Order File (.ord)
  Specifies the order in which logic in the application is executed. There is usually one order file per application, but there may be multiple order files. Each order file contains one Data Dictionary file (.dcf) that represents the primary data file that is being read and/or written.

- Logic File (.apc)
  Contains CSPro language statements

- Message file (.mgf)
  Optional file, it contains text for messages displayed on the output listing

- Other Data Dictionary Files (.dcf)
  Optional, it represents secondary data files (such as lookup files) that are read and/or written to during the batch run.

Batch Application Screen Layout



The screen is divided into three main work areas: the Tree View, the Logic View and the Message View.

- **Tree View**
  The window on the left half of the screen displays the batch edit tree with the root node ( ) selected.

- **Logic View**
  This is the window block in the upper portion of the right half of the screen. It is the programmer's "clean slate, to which may be added logic for any part of the data file: any item, any section, any record, even the file as a whole. It is up to the programmer to determine the correct placement and sequence of execution for each logical element. The initial screen will display:

```
{Application 'MYBATCHEDIT' logic file generated by CSPro }
PROC GLOBAL

PROC MYDICT_FF
```

  These two lines of code will **always** be in your application file. You can delete them, but they will always be regenerated and placed in your file on **open**, **save**, or **exit**. This is the beginning of your program. You write the declaration statements under `PROC GLOBAL`, then the procedures for the event.

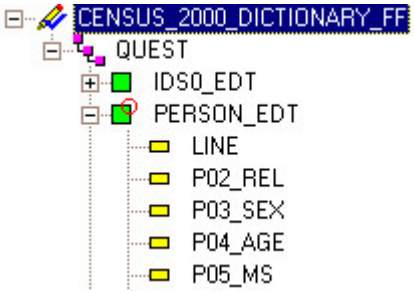
- **Message View**
  This is the window block in the lower portion of the right half of the screen. It is devoted to messages (user-created and system-generated). As with the Tree View, tabs are available to the programmer; clicking on one of them will make the contents of that view active. The **Compiler Output** tab displays errors found during compilation of your program; if the code compiled successfully, it will state "Compile Successful." The **Message** tab is used to type in error messages that will be used in the execution of the program.

If you wish to modify the size of any of these three work areas, just place the mouse over one of the separating bars, grab it, and drag to resize.

See also: Moving Around a Logic Application

Batch Edit Tree

When you create a batch edit application, the edit tree will be identical to the dictionary tree; that is, edit items will be listed as named and ordered in the dictionary. However, there are a few distinctions to make, as follows:



- **BatchEdit File:**
  This is the highest level node, i.e., the root node. It is the owner of all code, which is to say [1] level-, record-, and item-related code, [2] user-defined functions, and the [3] global routine.

- **BatchEdit Level:**
  This is the second-tier tree node, just below the root. It has a 1-to-1 correspondence with the same-named dictionary level.

- **BatchEdit Record:**
  This is the third-tier tree node, just below its level. It has a 1-to-1 correspondence with the same-named dictionary record.

- **BatchEdit Item:**
  This is the terminal or "leaf"-node; i.e., the lowest accessible level. It has a 1-to-1 correspondence with a dictionary item.

You are free to rename any of the above the unique names via the properties dialog box, but it is recommended that you retain the original name, so that it is easier for you to see which dictionary entity is being referenced. The batch edit tree represents the order in which the logic associated with each edit item is executed.

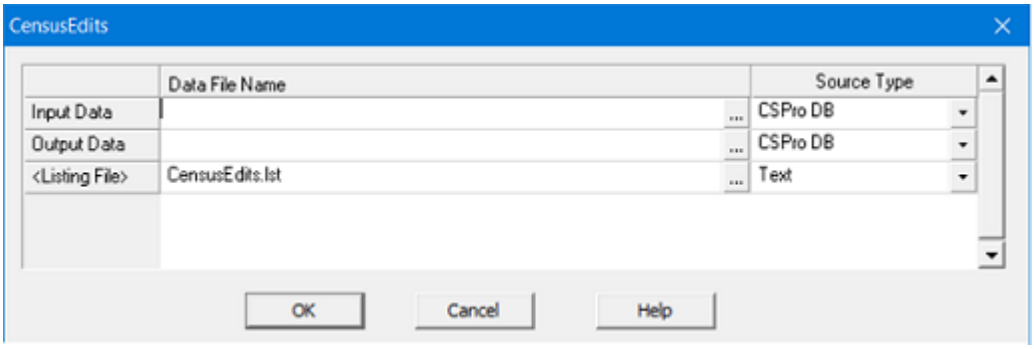
If code has been written for a given edit level, record, or item, a check mark will appear superimposed on the icon for that entity. This is how, at a quick glance, you can see where you have placed programming logic. Once one line of code has been written anywhere in the program, a check mark will appear on the root node.

You can never delete edit levels, records, or items (i.e., the entries shown on the **edit** tab). However, you can change the order of the logic execution by dragging the items within the Batch Edit tree view. When selecting a new edit item, the contents of the logic view will change to display the logic for the selected entity.

Pressing **Ctrl+T** in the batch edit tree will allow you to switch between the labels and the names of the items.

Run a Batch Edit Application

Press ; or press **Ctrl+R**; or select **Run** from the **File** menu to run a batch application. If you've made changes since you last compiled, CSPro will first compile your application. If your program compiles successfully, you will see a screen that looks similar to this:



The following rows are required:

- **Input Data**: This asks for the name of the data file against which you wish to run your batch application. This data file will not be modified in any way; it will only be opened, read, and closed. You can also select none as your input data file; useful, for example, when writing menu applications that do not require an input data file.
- **Output Data**: The output file is where the results of correcting your data will be written. If you are not making any corrections in your program, then the generated file will be an exact copy of the original data file. If you are making corrections to your data file, then this will be the corrected data file. If you do not need a corrected data file, and are simply running the program to generate a report, you can select none as the output.
- **<Listing File>**: This asks for the name of the file to which you want to write the results of the run. Results from `errmsg` functions will be written here. This file will always be generated, regardless of whether or not your program includes `errmsg` commands. Listing files generally have the extension .lst.

The following rows are optional, based on the logic in your application:

- **<Freq File>**: If your program contains any `freq` statements, the tabulated frequencies will be written to this file. If you do not specify a frequency file, the frequencies will be written to the listing file (if possible).
- **<Impute Freq File>**: If your program contains any `impute` statements, the results of this command will be written to this file. The default file extension is .impute_freq.lst, but you can use whatever you'd like. This field is optional; therefore, if your program contains impute commands, but you forget to specify a frequency file, no file will be generated. Similarly, if you indicate a frequency file but your program does not contain any impute commands, no file will be generated.
- **<Impute Stat Data>**: If your program includes one or more `impute` statements that use the `stat` command, the imputation statistics will be saved to the data file specified.
- **<Write File>**: If you have any `write` function calls in your program, the results will be saved to this file. This field is optional. If no write file is specified, the `write` function's output is saved to the listing file.

In addition to these rows, there may be rows where you can specify the names of lookup files, external files, and the paradata log.

Order of Editing

Order of Executing Batch Edit Events

CSPro executes the procedures in an application one case at a time. There are two main procedures for each element: a `preproc` and a `postproc`. If the type of procedure is not named, `postproc` is assumed. (There are three additional procedures that are not common in batch applications: onfocus, `killfocus`, and `onoccchange`.)

The following diagram illustrates the order of operations for a two-level batch edit application. Level 1 has two records (1 and 2) and level 2 has one record (3).

```
Application File preproc
    Level 1 (case) preproc
        Record 1 preproc
            Item 11 preproc
            Item 11 postproc
            ...
            Item 19 preproc
            Item 19 postproc
        Record 1 postproc
        Record 2 preproc
            Item 21 preproc
            Item 21 postproc
            ...
            Item 29 preproc
            Item 29 postproc
        Record 2 postproc
    Level 2 preproc
        Record 3 preproc
            Item 31 preproc
            Item 31 postproc
            ...
            Item 39 preproc
            Item 39 postproc
        Record 3 postproc
    Level 2 postproc
    (repeat level 2 procedures for other level 2 nodes)
    Level 1 (case) postproc
Application File postproc
```

See also: Batch Edit Order, CSPro Program Structure, Order of Executing Data Entry Events

Batch Edit Order

Eventually you'll reach the point where you have written edits for many variables and you will begin to wonder, just how do you control the order of execution? It's in the batch edit tree. The order of the items listed in the BatchEdits tree tab shows you the order of logic execution. (If there is no associated logic for an edit item, then the order is of course not important.)

What if you don't like the order that's given? Change it. As mentioned above in "The Batch Edit Tree" you can re-order items and records (but not levels) on the batch edit tree. When developing edit specifications, the edit of one variable might depend on another edit having already been completed (say, relationship before sex). If the dictionary wasn't designed in the order you need, then when a batch edit application is generated, the order will be incorrect.

Having said all this, there are a few nuances to the editing process that you may wish to note, specifically with regard to the execution of **preproc**s and **postproc**s execution:

- For BatchEdit items, there is no benefit to writing and maintaining **preproc** and **postproc** code blocks. Because a BatchEdit item is at the lowest level in the order tree, no other code would be executed in the interval between a **preproc** and **postproc** code block. Therefore it is suggested that only one code block exist for each item. If you do not preface the instructions with either "**preproc**" or "**postproc**", the code by default will be in the **postproc** block. This is the recommended approach; that is, accept the default.

- If a BatchEdit item is within a record that repeats, the logic will be repeated for each occurrence of the record. For example, if you have a population record that allows 30 occurrences, the logic for each of its member items can repeat up to 30 times, depending on the actual number of persons in the household. Suppose you have a household with three members: the head, the spouse, and a child. The logic for each data item (such as "relationship," "sex," and "age") will be executed three times.

- If a record repeats, the associated logic for that record will NOT repeat; instead, it will be executed once and only once for questionnaire. For example, take that population record again that allows 30 occurrences. Whether there are 1 or 5 or 30 people in the household, the associated logic for the BatchEdit record will execute only once. Therefore, if you have logic that must occur for each person in the household, we suggest you place that code under the first BatchEdit item in the record.

- Logic written for Level 1 will only execute once for a questionnaire/case. Logic written for Levels 2 or 3 will execute for each node, i.e., for each set of records contained in that level.

- Finally, logic written at the BatchEdit File node will execute only once for a data file. Therefore, if you have global variables that you need to initialize, etc., this is the place where that should take place.

Change Edit Order

By default, a new batch edit application fixes the order of editing to the order of items in the dictionary. If new items are added to or rearranged in the dictionary, the editing order is determined by the new dictionary arrangement.

To make your own custom order of the editing items within records, you need to do two things: from the **Options** menu, select "Custom Order"; then drag and drop items in the batch edit tree into the order you wish the edits to be performed. If you rearrange items within a record in the dictionary, the custom order will not change. If you add new items to a record, the new items will be placed at the end of the record for purposes of editing. If you unselect "Custom Order," the edit order will return to the order of items in the dictionary.

See also: Batch Edit Order

Correcting Errors

Methods of Correcting Data

Before you can correct your errors, you need to know what kind of errors you have. You have two methods of finding these errors: manually or automatically. Manually checking large quantities of data is an extremely time-consuming and error-prone task and not recommended. By contrast, automated searches for your errors is quick and, if done properly, a reliable method to use. If you wish to use CSPro to automate the search for errors, you must create a Batch Edit application.

Using a Batch Edit application to identify errors is a relatively easy task, though care must be taken to do so correctly. Improperly identifying errors can waste precious personnel resources, so a precise set of rules should be developed with input from subject-matter specialists.

Just as you have two methods available to you when searching for errors, you have two methods available to you for correcting errors: manually or automatically.

Manual Correction

Manual correction of a census could take years, and the possibility of human error is great. When large volumes of data are collected in censuses and surveys, it is not always practical to refer to the original document in order to correct an error. Often the data recorded on the original questionnaires are wrong or inconsistent.

Automatic Correction

With computer editing, both time and the possibility of human error is reduced significantly (just how much depends on how well your logic is written!). The high degree of accuracy and uniformity in computer editing cannot be obtained in manual editing. In computer editing, range checks and within-record consistency checks can easily be made, between-record edits can be done, and unknown information can be allocated (imputed) automatically. If an allocation method is used, you should strive to retain as much of the original information as possible. With a computer editing and imputation system like CSBatch, erroneous data can be corrected immediately and reports can be generated of all errors found and all changes made.

The programmer should plan and design computer edits to inspect the data and have the computer correct them according to specifications supplied by a subject-matter specialist. It would most likely be an extension of the original program written to find the errors—when you reach the point where there is an error, instead of (or in addition to) printing out an error message, you should now correct it with an appropriate value.

Actual methods of making corrections vary depending upon the item. In most instances, data items can be assigned valid codes with reasonable assurance that they are correct by using responses for other data items within the record, or in other records in the questionnaire. When recorded responses are missing, impossible, inconsistent, or unreasonable and cannot be determined from other responses in the same questionnaire, the hot deck technique can be used to assign entries.

Guidelines for Correcting Data

The purpose of editing is to make the data as representative of the real life situation as possible; do this by eliminating omissions and invalid entries, and by changing inconsistent entries. Below are some important principles that should be followed:

- The fewest number of changes should be made to the originally recorded data. You are only trying to make a record or questionnaire acceptable, not make it conform to what you think should be acceptable.
- If you must change a data value, do so only once. If you change a person's age, then later find this age doesn't work for another edit, then you didn't write the original edit correctly. Go back and review the first edit.

- For certain items it may be acceptable to have a "not reported [NR]" or not stated [NS]" category. Thus, in case of an omission or an inconsistent, impossible, or unreasonable entry, a code for "NR" or "NS" can be assigned.
- Obvious inconsistencies among the entries should be eliminated.
- Providing corrected values for erroneous or missing items should be supplied by using other values as a guide; (for example, entries for the housing unit, person, or other persons in the household or comparable group), and always in accordance with specified procedures.
- Specifications for editing the questionnaire data should be developed at the same time as the questionnaire itself. Information to the computer programmer concerning what checks and imputations should be made in the data are provided through editing instructions or specifications. A subject-matter specialist, in collaboration with the computer specialist, should write editing instructions. The instructions describe the action to be taken on each data item. The editing instructions should be clear, concise, and unambiguous, since they serve as the basis for the CSPro editing program. Specifications for machine editing may be written instructions, decision tables, flowcharts, or pseudo-code. Pseudo-code (IF/ELSE logic) is recommended because it can be easily translated to CSPro code. Pseudo-code can be prepared using basic word-processing or text-editing software and can be easily modified.

If the hot deck method of imputation is used, it is important that the edit specifications indicate where, during the processing, hot decks are to be updated, that is, at which points in the logic the data items can be considered valid.

Imputation

Imputation refers to the process of providing values for missing, erroneous, or inconsistent responses. For example, if a person's sex code is invalid (i.e., out of range or otherwise unacceptable) or missing; then an appropriate response should be substituted.

You may decide that for missing data, you'd rather just keep it "missing" and publish your tables with an extra column (or row) for unknown values. This is a very cumbersome method, however, as the number of missing values will vary for each data item, and so the number of missing entries will vary from table to table, making the data difficult to analyze.

Inconsistent responses occur when a response yields an impossible situation with respect to another response. For example, if a 5-year-old female reports having children, either her age is wrong, or her fertility data are wrong (i.e., that section should be blank). This type of error must be corrected, as your users will place very little faith in the quality of your data if this type of condition becomes evident in the tabulations. Many users also do not look kindly on "missing" or unreported data. Of course, nothing can correct for bad data, and if you find that a significant amount of your data are bad (poorly designed questionnaire, inadequate field procedures, inattentive coders and keyers, etc), you may want to reconsider whether the data should be released at all.

Procedures have been developed to provide the missing information, thereby avoiding discrepancies and the need to determine percentages twice (with and without unknowns). For a detailed discussion on using imputation and the methods available to you, please refer to the United Nations Handbook on Population and Housing Census Edits.

Essentially, two methods of imputation are available: static and dynamic.

Static Imputation

Static imputation means providing a value from a predetermined set of values. Suppose a person's sex is missing or invalid (out of range). If we decide to change the response using static imputation, there are two basic methods to use: hard coded or from a "cold deck."

Hard Coded

Using our example above, we would programmatically set SEX to the value we think it should be. For example:

```
PROC GLOBAL

numeric sexToggle = 1;

PROC SEX

    if not SEX in 1:2 then
        SEX = sexToggle;
        sexToggle = 3 - sexToggle;
    endif;
```

What we've done above is a very primitive form of imputation. Essentially, when we encounter a bad value for sex, 50% of the time the variable SEX will be assigned the value "male," and 50% of the time the value "female." Note that no accommodation was made for other responses; for example, if fertility data were present, you might not wish to make this person "male." Or if this were an enumeration of a prison where the entire population is male, you would probably not want to be adding females to this group! So while this method can be used, you need to take into account other responses. We attempt to do this in our next method of static imputation, where we use a cold deck.

Cold Deck

With the cold deck approach, known information about individuals with similar characteristics (sex, age, relationship to household head, economic status, etc.) is used to determine the "most appropriate" response to be used when some piece (or pieces) of related information is invalid.

For example, suppose a person's age is missing or invalid. We might have a table as follows, where the row indices represent the person's sex (1 = male, 2 = female), and the column indices refers to the person's relationship codes (1-5) (this table assumes that the relationship and sex codes have already been corrected):

	Head (1)	Spouse (2)	Child (3)	Other Relative (4)	Non-Relative (5)
Male (1)	35	50	10	41	65
Female (2)	32	48	10	37	68

In the event that a female child was found to have a missing age, she would be given the age of 10. If a female head of the household had a missing age, then her age would be given as 32. This method is acceptable if you do not need to use it often; that is, if very little of your data is missing or invalid. Also, if your population is fairly homogeneous (for example if you were correcting for religion and 90% of the population is Muslim), then this will not result in an unrealistic portrayal of your country.

However, if you find yourself referring to this table often, or you have a very diverse population where a few static values do not give an accurate portrayal, then your data will end up skewed. For these reasons (and others), dynamic imputation is the preferred method.

Dynamic Imputation (Hot Deck)

Dynamic imputation refers to the concept of using constantly changing values for your allocation routines. It is similar to static (cold deck) imputation, except that instead of creating a table and assigning allocation values that remain fixed, the tables are continually updated with valid and consistent values taken from the population being edited.

For example, assume, for a given person, that the age, relationship, and sex codes appear correct and that consistency checks validate these items. You can use the values from this person to update your "cold deck," thus making it a "hot deck." Below is the table given in the cold deck example:

	Head (1)	Spouse (2)	Child (3)	Other Relative (4)	Non-Relative (5)
Male (1)	35	50	10	41	65
Female (2)	32	48	10	37	68

If the person in question is a male 6-year-old child, the table can be updated with new information, giving the following:

	Head (1)	Spouse (2)	Child (3)	Other Relative (4)	Non-Relative (5)
Male (1)	35	50	6	41	65
Female (2)	32	48	10	37	68

You would proceed in this way for every person in the household who had correct age, sex, and relationship values. Then, when you encounter a person with an invalid or missing age, you can extract from the table, using the sex and relationship of the person, a value for age. This value is more likely to be appropriate for the person than would be a purely random value. (The preceding example is clearly a simplification of the hot deck technique, which requires great care in constructing and updating the tables used for allocation.)

See also: Use Hot Decks

DeckArrays

Using "DeckArrays" in CSPro is a simplified way of working with hotdecks. It is important to understand the hotdeck process well before using DeckArrays because the functions getdeck and putdeck automatically handle recodes and thus hide much of the behavior of hotdecks from the programmer. DeckArrays eliminate the need for the programmer to recode variables to fit within the boundaries of an array as the recodes are processed automatically based on the values in a value set.

If using DeckArrays it is very easy to change the parameters of a hotdeck. For instance, if a subject matter specialist decides that a variable should be hotdecked based on five-year intervals of age but later changes the specification to be ten-year intervals, programming such a change is very quick if using DeckArrays, whereas with standard hotdecks reprogramming the recodes may be time-consuming.

Declaring a DeckArray

Declare a DeckArray in PROC GLOBAL as you would a normal array but instead of using numeric dimensions, specify the name of a value set. The size of the dimension will match the size of the value set. If you make changes to the value set, the size of the array will automatically reflect the changes.

```
array Age_HD_from_Sex(SEX_VS); // same as array Age_HD_From_Sex(2);
array Age_HD_from_Sex_Relationship(SEX_VS,RELATIONSHIP_VS);
```

To create a DeckArray, at least one of the dimensions must be a value set, but it is permissible that the other dimensions are declared with a numeric size. Because the program can only recode values for the dimensions based on value sets, it is necessary to specify the indices of any numeric dimensions when calling getdeck or putdeck.

```
array Age_HD_from_EA_Sex(200,SEX_VS); // same as array Age_HD_From_EA_Sex(200,2);
```

Accounting for "Else" Values

Occasionally a programmer wants to create a hotdeck with a "leftover" row for any values not contained in the value set. Sometimes these are invalid values, other times the value set can be created so that this leftover, or "else," row consists of values valid for the census or survey. For instance, if a user wants to hotdeck a variable based on Christian sects, the programmer would create a value set with all the Christian sects, and then would create a DeckArray with a leftover row for all people who do not belong to the Christian sects identified. This is indicated by adding a (+) after the value set name.

```
array CEB_HD_Religion(RELIGION_CHRISTIAN_SECTS_VS(+));
```

Example 1

Simplified edit specification: If the head of household's age is missing or invalid, the value should be imputed with a hotdeck based on the head's sex and the spouse's age. If there is no spouse or no valid age for the spouse, hotdeck based on the head's sex and the household size, which will be assumed to have a maximum value of 10+.

```
PROC GLOBAL

array ageFromSpouseHotdeck(SEX_VS,AGE_VS);

numeric maxHHSize = 10;

array ageFromHHSizeHotdeck(10,SEX_VS); // household size is from 1 - 10+
```

```
PROC AGE

    universe RELATIONSHIP = 1; // we're only editing head of households here

    // assumption: by now RELATIONSHIP and SEX have been edited for the head of
household

    numeric ptrSpouse = seek(RELATIONSHIP = 2);
    numeric minSpouseAge = 12;

    if SEX = 2 then
        minSpouseAge = 15; // males cannot marry until 15
    endif;

    if AGE in 15:95 then  // the head of household has a valid age

        if ptrSpouse <> 0 and AGE(ptrSpouse) in minSpouseAge:95 then  // the spouse has
a valid age

            putdeck(ageFromSpouseHotdeck,AGE,SEX,AGE(ptrSpouse)); // update the hotdeck

            // because SEX belongs to the current record being examined (the head of
household)
            // it can be left blank and CSPro will pick the correct SEX; the age of the
spouse must always
            // be passed because we are updating the hotdeck based on the age of the
spouse, not the age
            // of the head (the age of the head is invalid after all)

            putdeck(ageFromSpouseHotdeck,AGE, ,AGE(ptrSpouse)); // this is the same as
above

        endif;

        putdeck(ageFromHHSizeHotdeck,AGE,low(totocc(),maxHHSize),SEX);  // update the
hotdeck based on HH size
        putdeck(ageFromHHSizeHotdeck,AGE,low(totocc(),maxHHSize));  // this is the same
as above


    else  // the head of household's age is invalid so we must impute it

        if ptrSpouse <> 0 and AGE(ptrSpouse) in minSpouseAge:95 then  // the spouse has
a valid age
            impute(AGE,getdeck(ageFromSpouseHotdeck, ,AGE(ptrSpouse)));

        else
            impute(AGE,getdeck(ageFromHHSizeHotdeck,low(totocc(),maxHHSize)));

        endif;


    endif;
```

Example 2

VS_SEX has two values: 1 (Male) and 2 (Female). VS_EDUC has three levels: 1 (No schooling), 2 (Primary schooling), 3 (Secondary schooling).

```
// Command:                                        // The hotdeck looks like
                                                   // this after the command:
```

```
array ageHotdeck(VS_SEX,VS_EDUC(+));            //  00   00   00   00
                                                //  00   00   00   00

SEX = 1;   EDUC = 1;   AGE = 20;                //  20   00   00   00
putdeck(ageHotdeck,AGE);                        //  00   00   00   00

SEX = 2;   EDUC = 3;   AGE = 50;                //  20   00   00   00
putdeck(ageHotdeck,AGE);                        //  00   00   50   00

SEX = 1;   EDUC = 8;   AGE = 64;                //  20   00   00   64
putdeck(ageHotdeck,AGE);                        //  00   00   50   00

SEX = 2;   EDUC = 3;   AGE = notappl;           //  20   00   00   64
AGE = getdeck(ageHotdeck); // AGE = 50         //  00   00   50   00

SEX = 2;   EDUC = 0;   AGE = 10;                //  20   00   00   64
putdeck(ageHotdeck,AGE,,);                      //  00   00   50   10

SEX = 2;   EDUC = 0;   AGE = 11;                //  20   00   00   64
putdeck(ageHotdeck,AGE,SEX,);                   //  00   00   50   11

SEX = 2;   EDUC = 0;   AGE = 12;                //  20   00   00   64
putdeck(ageHotdeck,AGE,SEX);                    //  00   00   50   12

SEX = 2;   EDUC = 0;   AGE = 13;                //  20   00   00   64
putdeck(ageHotdeck,AGE,,EDUC);                  //  00   00   50   13

SEX = 2;   EDUC = 0;   AGE = 14;                //  20   00   00   64
putdeck(ageHotdeck,AGE,SEX,EDUC);               //  00   00   50   14

                                                //  20   00   28   64
putdeck(ageHotdeck,28,1,3);                     //  00   00   50   14

putdeck(ageHotdeck,28,0,3); // returns          //  20   00   28   64
// DEFAULT because 0 is not in VS_SEX           //  00   00   50   14
// and no (+) specified
                                                //  20   00   28   64
AGE = getdeck(ageHotdeck,2,300); // AGE = 14    //  00   00   50   14
```

See also: Getdeck Function, Putdeck Function, DeckArray Leftover Rows

DeckArray Leftover Rows

When using unedited variables as parameters of your DeckArray, it may be useful to include a "leftover" (spillover) row for cases in which the variable is not valid. This functionality allows the user to simulate the use of many hotdecks using a single array. An example will best explain this functionality.

This following is a hotdeck for housing type, based on the source of drinking water (H8), the type of toilet (H9), and whether or not the household has electricity (H10).

```
array housingTypeHD(H8_VS1(+),H9_VS1(+),H10_VS1(+)) save;
```

At the point that housing type, H6, is edited, none of the dependent variables in the hotdeck have been edited, so the values can be blank or invalid. That is why, in the array definition, a (+) has been added after the value set names. The (+) gives an extra row for any value that is not in the value set.

When updating the hotdeck in cases where the housing type value is valid, adding a (+) after the DeckArray name will update the "leftover" rows.

```
putdeck(housingTypeHD(+),H6);
```

This means that when the hotdeck is called upon for an edit or imputation, having valid values for H8, H9, or H10 is not necessary to return a valid H6 value. This fundamentally makes the housing type DeckArray eight hotdecks combined into one. The table below explains what happens when executing a getdeck function call:

```
impute(H6,getdeck(housingTypeHD));
```

H8	H9	H10	Take the H6 value from the nearest neighbor with the same values for...
Valid	Valid	Valid	H8, H9, and H10
Valid	Valid	Invalid	H8 and H9
Valid	Invalid	Valid	H8 and H10
Valid	Invalid	Invalid	H8 only
Invalid	Valid	Valid	H9 and H10
Invalid	Valid	Invalid	H9 only
Invalid	Invalid	Valid	H10 only
Invalid	Invalid	Invalid	The previous valid household's value

Or in words:

- If electricity is blank, use a hotdeck based on water and toilet.
- If electricity and water are blank, use a hotdeck based on toilet.
- If water is blank, use a hotdeck based on electricity and toilet.
- If all three are blank, use a hotdeck based on the previous valid household.
- Etc.

See also: DeckArrays, Getdeck Function, Putdeck Function

Types of Edits in Batch Editing

Overview

Errors in data can be (roughly) categorized as problems in either structure or consistency. Problems with structure may require a different approach to correction than will problems with validity or consistency. For this reason, many users choose to implement a two-stage edit, where the data are first rendered structurally correct, and then passed through a second edit to ensure that all items are valid and consistent. However, it is expected that while the subject-matter specialists will have established rules for validity and consistency, the computer specialist may have equally important input into the definition of structural validity.

After keying or scanning your data, there will be errors in the data file. This is unavoidable, and will be a combination of human and computer error. It will therefore be necessary to correct the data by writing a series of edit routines (procedures) to systematically and consistently clean your data files.

The **Batch Edit Designer** module allows you to create and modify batch edit applications. A batch editing application contains logic which you can apply against one set of files to produce another set of files and reports. Batch editing applications can be used to gather information about a data file.

To create these edit routines, you will use CSPro to develop the batch editing application based on the dictionary that describes your data files. If you received this data file from someone else and do not have a dictionary that describes it, you will need to create a dictionary before you are ready to develop programming logic for it. You use CSBatch to run the application. For small surveys and for testing applications, you can run CSBatch directly from CSPro, on the same computer. For large surveys and censuses, which require a production environment, you can transfer the application files to other computers and run CSBatch on them.

You can have the following runtime features in your batch editing application:

Check Case/Questionnaire Structure

These checks ascertain that all records that should be present for a particular questionnaire (case) are supplied and that no extra records are included.

Validate Individual Data Items

These checks are designed to determine whether a response has a value that is inside or outside the valid limits for that response. Although these checks are normally performed during data entry, you can also perform them with CSBatch.

Test Consistency Between Items

In these checks, two or more responses in a questionnaire are compared for consistency. For example a male person reporting having any children is an inconsistency between the response to **sex** and the response to **children born**. (Only females report **children born**.) The responses being compared may be in the same record or in different records within a questionnaire.

Automatic Modification

When a census or other large survey is being processed, it would be unduly cumbersome to make most data corrections by visually examining the errors. CSPro provides the facility for not only finding incorrect or inconsistent data, but also making modification to the data. Needless to say, modifications are not always corrections. Moreover, any modifications to data that have been collected must be carefully thought out and monitored through CSPro's edit statistics reports.

Generate Edit Reports Automatically

CSPro generates comprehensive statistics about the edit tests carried out and the number of changes made to the data. The user may also create a customized report including or excluding any of the information generated by CSPro during the editing process.

Generate Reports

You can write customize reports to a file.

Match Files

CSPro allow for matching two files and gathering information from both. The feature is useful, for example, when a file must be created which has a combination of data from two other files.

Modify Data Value

During the editing process, values that are invalid, inconsistent, or otherwise unacceptable will need to be replaced with correct values.

Use Arrays for Imputations

Whether you choose to correct data using hotdeck or cold-deck methodology, the arrays are easily defined, accessed, and updated.

Generate Imputation Statistics

CSBatch will automatically keep track of changes, but users my choose to format reports to their specifications. By specifying a denominator variable, users may obtain rates of imputation, including rates for individual values (e.g., **male** or **female** when imputing the variable **Sex**).

Create Additional Variables

Recoded or composite variables may be created during the editing process. The only requirement is that space be allocated (via the dictionary for the file) in the output record. The updated output file is automatically created when a file name is specified at the time of execution.

Reference Multiple Lookup Files

An essentially unlimited number of secondary or auxiliary files may be attached to an application and used as reference or lookup files. The application may read from and write to any of these files.

How to ...

Manipulate Automatic Reports

During the testing and debugging stages of developing your application, you'll want to write out a lot of error messages to help find problem areas, and keep statistics on the number of times certain code blocks are being executed (or values are being imputed). You may begin to notice that you're using the same error message in several places. Rather than write out the message every time it's needed, you can define it once and refer to it whenever needed. For example, suppose you have the following error message scattered throughout your code:

```
errmsg("Current age after imputation is %d", AGE);
```

Why bother retyping it each time? You can simply define it once and reference it over and over. To do so, select the **Message** tab in the **Message View**. You will see one line that has been generated for you; it reads: `{Message code file generated by CSPro}`. Beneath this simply add your error message (we'll give it number **100001**):

```
100001  Current age after imputation is %d
```

Then, whenever you want to use this message in your code, simply use the message number:

```
errmsg(100001, AGE);
```

Besides simplifying your work, after you run your program, a convenient summary statistic will be generated for each user-defined error message, indicating how often it was used. A sample listing is shown below:

```
Number    Freq  Pct.  Message text                                        Denom
------    ----  ----  ------------                                        -----
100001       1    -   Current age after imputation is %d                    -
```

See the `errmsg` function for a detailed explanation of all the options available to you.

Create a Specialized Report

Manipulate Automatic Reports showed how to add messages to the default report which is automatically generated after a run. This type of report may be useful for debugging, but in general, the subject-matter staff responsible for edit review will usually prefer a more "user-friendly" report - that is, one that presents the information in a format dictated by the users.

Such a "custom" report can be generated by using the write function. This command will put the information you want where you want it in your report. For example, for each questionnaire, you'll want to know the identifying ID values. If this were a population census, the case ID would likely be composed of levels of geography [Province, District, Village, EA, etc.] attached to a household identification. The "errmsg " command could display the Ids as follows:

```
Case [010117100110870031] has 12 messages (0 E / 0 W / 12U)
```

As you can see, this may be difficult for the non-programmer to decipher. But by using the **write** command, you can more clearly display this. One way would be to put the following **write** statements in the preproc of the first level (in this way it would only get written out once per questionnaire):

```
PROC QUEST
preproc
  write ("***************");
  write (" Province: %3d", PROVINCE);
  write (" District: %3d", DISTRICT);
```

```
write (" Village : %3d", VILLAGE);
write (" EA : %3d", EA);
write ("***************");
write (" "); { blank line }
```

After the execution of the program, the .wrt [report] file would show the following (of course, actual values will vary depending on the questionnaire IDs):

```
***************
Province: 1
District: 7
Village : 30
EA : 4
***************
```

Additional **write** statements can be included in the batch edit program to generate a customized report.

Use Hot Decks

Hot decks in CSPro are implemented using arrays. First, you must declare the array under the `PROC GLOBAL` section. Then you need to set the initial values for the hot deck array. There are two ways to initialize and maintain arrays in CSPro: inline (in program logic), or using saved arrays.

Once the array has been declared and initialized, you need to add logic to check each occurrence of the variable you wish to impute using the hot deck. If the value of the variable is valid, update the hot deck by assigning this value to the appropriate cell in the array. If the value is invalid, set the value of the variable based on the corresponding cell in the hot deck. For example:

```
PROC AGE

    if AGE = notappl then
        // if the value for age is not valid...
        // assign the value from the hot deck based on sex and relationship
        impute(AGE,AgeSRDeck(SEX,RELATIONSHIP));

    else
        // otherwise...
        // update the value of the hot deck
        AgeSRDeck(SEX,RELATIONSHIP) = AGE;

    endif;
```

When an age is missing during the data file's processing, we will use a value from the array `AgeSRDeck`; if the age is present, we will "refresh" the age for the person using the current sex and relationship codes as indices into the array.

For a more detailed explanation of what hot decks are, refer to the United Nations Handbook on Population and Housing Census Edits.

See also: Dynamic Imputation (Hot Deck)

Initialize Hot Decks in Program Logic

In this technique, the hot deck is initialized in program logic. Hot deck values are updated during the program's execution but the values are not saved externally.

The following is an example of code that will initialize age values based on sex and relationship:

```
PROC GLOBAL

// declare an array and set the initial values
array AgeSRDeck(2,6) =
                        27, // male head of HH
                        30, // male spouse of head
                         6, // male child of head
                        58, // father of head
                        11, // male other relative of head
                        24, // male non-relative of head
                        32, // female head of HH
                        25, // female spouse of head
                         8, // female child of head
                        60, // mother of head
                        10, // female other relative of head
                        27; // female non-relative of head

PROC AGE

    if AGE = notappl then
        // if the value for age is not valid...
        // assign the value from the hot deck based on sex and relationship
        impute(AGE,AgeSRDeck(SEX,RELATIONSHIP));

    else
        // otherwise...
        // update the value of the hot deck
        AgeSRDeck(SEX,RELATIONSHIP) = AGE;

    endif;
```

See also: Dynamic Imputation (Hot Deck), Use Hot Decks, Initialize Hot Decks Using Saved Arrays

Initialize Hot Decks Using Saved Arrays

In this technique, the hot deck is initialized by running the program twice and using the results of the first run to initialize the hot deck for the second run. The array is declared using the `save` keyword. No initialization of the array is done in the program logic. The program is run twice. During the first run of the program, the array has not been initialized so it is likely that some imputations from the hot deck will use these uninitialized values. The results (the output file) from the first run of the program are therefore discarded. However, during the first run of the program, the hot deck is filled with valid values. Since the array has been declared as a saved array, these values are written to the saved arrays file. When the program is run a second time, these values are read from the saved array file and used as initial values from the array. The advantage of this approach over initializing the array in the program logic is that all values used in the hot deck are taken from the data file.

The following example uses a saved array to initialize a hot deck:

```
PROC GLOBAL

// declare an array initialized from the saved array file
array AgeSRDeck(2,6) save;

PROC AGE

    if AGE = notappl then
        // if the value for age is not valid...
        // assign the value from the hot deck based on sex and relationship
        impute(AGE,AgeSRDeck(SEX,RELATIONSHIP));

    else
        // otherwise...
        // update the value of the hot deck
        AgeSRDeck(SEX,RELATIONSHIP) = AGE;

    endif;
```

See also: Dynamic Imputation (Hot Deck), Use Hot Decks, Initialize Hot Decks in Program Logic

Interpret Reports

After specifying your file(s), a progress dialog bar will be displayed as CSBatch works its way through your data file, then TextViewer is launched and generate the following listing report:

```
Application      C:\Program Files\CSPro 2.5\Examples\CSPro_Test.bch
Type             BATCH
Input Data       C:\Program Files\CSPro 2.5\Examples\Popstan Census 2000.dat
<Output>         C:\Program Files\CSPro 2.5\Examples\Popstan Census 2000.out

Date             May 04,2004
Start Time       10:24:48
End Time         10:24:51

CSPRO Process Summary
          +-------------------------------------------------------+
          |   29143 Records Read ( 100% of input file)            |
          |       0 Ignored (       0 unknown,        0 erased) |
          |   47063 Messages (      36 U,     9004 W,   38023 E) |
          +---------+--------------+--------------+-------------+
          |  Level  |  Input Case  |  Bad Struct  |  Level Post |
          +---------+--------------+--------------+-------------+
          |     1   |       4872   |          0   |       4872  |
          +---------+--------------+--------------+-------------+



Process Messages

*** Case [0107058028204601 91] has 12 messages (8 E / 2 W / 2U)
    W 88870 Value '01' out of range - check P16_IND(1)
    W 88870 Value '01' out of range - check P16_IND(5)
    U   -69 Household tenure is vacant, but there are 5 person records.
    E 88203 6 inconsistent fields detected following a 'skip to H13_PERSONS'
          command in Var H06_TENURE PostProc
    E 88212 ... H07_RENT should be blank (currently '000')
    E 88212 ... H08_TOILET should be blank (currently '9')
    E 88212 ... H09_BATH should be blank (currently '5')
    E 88212 ... H10_WATER should be blank (currently '6')
    E 88212 ... H11_LIGHT should be blank (currently '6')
    E 88212 ... H12_FUEL should be blank (currently '7')
    U   -84 Number of persons is 0, number of person records is 5.
    E 88230 Unexpected 'reenter' command reached in Var H13_PERSONS PostProc
```

The "CSPro Process Summary" gives you information about the records and cases. In this example:

- 29143 records were read from the input file, which represent 100 % of the input file
- "Ignored" is the sum of "Unknown" and "Erased" cases.
- "Unknown" represents number of cases with invalid record type code.
- "Erased" represents number of records with a "~" character
- The list report contains 47063 messages of which: 36 messages were defined by the user; 9004 are warning messages generated by the system; and 38023 are error messages generated by the system
- "Level 1" indicates that the following statistics are for Level 1 only. If the application had more than one level, the statistics will be displayed for each level.
- The input data file included 4872 cases (questionnaires) of which: there were "0" case with bad structure, meaning that no required records were missing; and 4872 cases were written to the output file ("Level Post").

The "Process Messages" gives you information on each case:

- *** Case (010705802820460191) has 12 messages (8 E/ 2 W/ 2 U)
  Indicates that questionnaire 010705802820460191 (codes correspond to the ID items in the Dictionary) has 12 messages: 8 error messages generated by the system; 2 warning messages generated by the system; and 2 error messages defined by the user.

- W 88870 Value '01' out of range – check P16_IND(1)
  Indicates that field P16_IND in first person (1) has a value "01" which is out of range because it has not been declared in the Dictionary. This is Warning 88870, generated by the system.

- U -69 Household tenure is vacant, but there are 5 person records.
  Message defined by the user in line 69 of the program

- E 88212 … H07_RENT should be blank (currently '000')
  Indicates that field H07_RENT has a value '000' but it should be blank. (This is a vacant unit)
  This is Error 88212, generated by the system.

If a write (.wrt) or frequency (.frq) file was generated, then they will also be loaded into TextViewer for display; to rotate between the various files, select the "Window" option from TextViewer, and choose from among the files listed at the bottom. If TextViewer was already running when you launched your application, it will be refreshed with the latest run results.

See the explanation given under Run an Application

Run Production Batch Edits

You can customize CSBatch's behavior by creating a PFF file. You can then use the PFF file as a command line parameter for CSBatch.exe. For example, if you name your PFF file MyEdits.pff, then you run the batch edit application by invoking:

```
"C:\Program Files (x86)\CSPro 7.7\CSBatch.exe" MyEdits.pff
```

This assumes that CSPro was installed in the default directory. Your PFF file must have a .pff extension.

You can create a PFF file in one of two ways: either [1] create one with a text editor (such as Notepad or Wordpad), or [2] have it generated automatically for you by launching your batch application from within the CSPro Designer. The file will have the same name as your application, but with a .pff extension instead of .bch. For example, if your batch application was named MyEdits.bch, the generated PFF would be named MyEdits.pff. You can also manipulate PFF files in logic by using the pff object.

The following section shows the options available to you in a CSBatch PFF file. A PFF file is not case sensitive, so you can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Batch

[Files]
Application=.\MyEdits.bch
InputData=.\MyInputData.dat
OutputData=.\MyOutputData.dat
Paradata=.\MyParadataLog.cslog
Listing=.\MyEdits.lst
Freqs=.\MyEdits.tbw
ImputeFreqs=.\MyEdits.impute_freq.lst
ImputeStat=.\MyEdits.impute_stat.csdb
WriteData=.\WriteMessages.txt
SaveArray=.\MyEdits.bch.sva
CommonStore=.\MySettings.db
HtmlDialogs=.\MyDialogsDirectory
BaseMap=.\MyMap.mbtiles

[ExternalFiles]
LOOKUP_DICT=.\LookupFile.dat

[Parameters]
ViewListing=Always
ViewResults=Yes
ListingWidth=120
MessageWrap=No
ErrmsgOverride=No
InputOrder=Sequential
Parameter=your choice
Language=LN
OnExit=.\Menu.pff
```

[Run Information]

The **[Run Information]** block is required and must appear exactly as shown in the example above.

[Files]

The **[Files]** block is required. A description of the files, not all of which have to be specified, is as follows:

- Application=the name of the batch edit application that you created.
- InputData=the data file(s) against which the batch edit program will run. These files will not be modified during the run.
- OutputData=the revised/corrected input data will be saved as this file(s).
- Paradata=the name of a file where events logged during the application's run are saved.
- Listing=a report of the batch operation.
- Freqs=a frequencies file containing the results of `freq` statements.
- ImputeFreqs=if you have any `impute` function calls in your program, the results of these statements will be written to this frequencies file.
- ImputeStat=if you have any `impute` function calls in your program that use the `stat` command, the results of these statements will be written to this data file.
- WriteData=if there are one or more `write` function calls in your program, the text of these statements will be written to this file.
- SaveArray=if your program contains one or more saved arrays, the contents of these arrays will be read from and written to this file.
- CommonStore=if you use the `loadsetting` or `savesetting` functions in your program, this file overrides the default location where these settings are saved.

- HtmlDialogs=the name of a directory that contains custom HTML dialogs that override the default CSPro ones.
- BaseMap=the name of the online map or the filename of an offline map used for CSPro mapping.

[ExternalFiles]

If the **[ExternalFiles]** block is present, it means that a second (or more) dictionary was linked to the batch application. In the example above, LOOKUP_DICT is the dictionary name, and LookupFile.dat is the name of the data file that contains the lookup codes.

[Parameters]

The **[Parameters]** block is optional. This section defines parameters for the batch run.

- ViewListing=determines whether you see the batch run report. If this entry is missing or set to **Always**, then you will see the generated report. Other available options are **OnError**, in which case you will see the listing report only if an error occurred during the run, or **Never**, in which case you will never be shown the generated report.
- ViewResults=determines whether or not the impute and/or write file are displayed in Text Viewer at the end of the run. The available choices are **Yes** or **No**. If the entry is missing, the resultant file(s) will be displayed by default. For more information on these files, see Run a Batch Edit Application.
- ListingWidth=allows you to control the number of characters outputted to the listing file before forcing the start of a new line. This is set to 120 characters by default. If your screen resolution and/or printer permits, it may be useful to increase the width of the listing file. This only applies to text listing files.
- MessageWrap=determines whether or not summary messages displayed at the end of a listing file will be outputted on several lines when they are too long to fit on one line. The default option is **No**, in which case the messages are truncated so that they fit on one line. This only applies to text listing files.
- ErrmsgOverride=allows you to override the default behavior of the `errmsg` function. The override only affects `errmsg` functions used in the code without a `case` or `summary` specifier. The option **No** maintains the default behavior (displaying the messages for each case, as well as in the summary). The option **Summary** only displays the message in the summary. The option **Case** displays the message for each case but not in the summary.
- InputOrder=allows you to override the default order in which cases are processed in a data file. The default option, **Sequential**, means that cases will be processed from the first case to the last case, in file order. The option **Indexed** overrides this order and cases will be processed in alphabetical order of the case IDs. Text data files can contain duplicates case IDs if processed in sequential order, but not in indexed order.
- Parameter=allows you to pass in a string to your program. The parameter can be any length, although the string that retrieves the value in your program (via the `sysparm` function) must be large enough to accommodate it. Once the parameter is retrieved, it can be parsed by your program for further usage.
- Language=specifies the initial language of the program. The parameter must match the name of a language specified in the dictionary or message file.
- OnExit=specifies a PFF file to run after the application closes. This can be useful, for example, if you want to relaunch a menu program after processing data.

Steps in Developing a Batch Editing Program

General Batch Editing Issues

This guide has described the CSPro language and has presented examples of various types of edits. However, the job of writing and testing a CSPro program to perform many edits is still a large one. Because of the structure of the CSPro language, this job can be distributed among several people if it is well coordinated.

Before coding begins, a complete set of edit specifications and a formatted listing of the data diction ary should be available. A team leader should be appointed; this person should have a thorough understanding of the CSPro language. The leader should develop the naming conventions and standards that are to be followed by other members of the team. During the course of program development, questions will arise about the edit specifications, and it is important to involve subject-matter specialists to resolve these issues.

This section outlines a step-by-step approach to developing a CSPro program. It assumes that a data dictionary has been developed. If this is not the case, then the first step is to build the data dictionary using the Data Dictionary module and to check the dictionary carefully against original specifications.

An editing system may contain more than one CSPro program. It is not unusual to have at least two CSPro programs in the edit system: the first is used to find and report structure errors which must be resolved before the data moves on to the next stage. These include missing records, duplicate records, or other incompatibilities with requirements which are considered essential to the processing. The second program is the more traditional value-validation and consistency edits.

Review Edit Specifications

In order to determine the total editing task, the user should understand what edits are to be performed. Are the edits fairly simple or mostly complex? Are the edits merely to find and report errors in the data file, or will the program correct the errors as well? Is a non-response value being allowed for any or all data items? Is a look-up file going to be needed? Are there any auxiliary files to be produced? What types of reports will be needed and how will they be organized?

The three types of edit statistics reports are:

- **Case**: Output listing shows case by case.
- **Summary**: Output listing shows (1) a summary of the number of times each error message was generated, and (2) the number or errors as a percentage of total cases in which the edit test was invoked.
- **Frequency**: This report is generated when at least one `impute` statement is coded in the program. It assigns a value to a data item and logs the frequency of assignments.

Typically, during the testing phase, the report by case is very useful, because it permits detailed examination of the effects of the logic coded. After testing, it is usually not used because of the volume of reports generated.

Define Coding Standards

Without a good set of coding standards, it will be difficult to incorporate the various subroutines into one program. Consistent naming standards and coding techniques must be considered; specific suggestions include the following:

- Names for variables to be defined in the program should have unique prefixes. The standard could be that variables

needed for editing LINE_NUMBER be N01 through N10; those needed for editing AGE would be N11 through N20, and so forth.

- Names for hot-deck arrays should also be standardized. For example, an array for hot decking age by marital status, relationship, and sex would be defined as HD_AGE_MAR_REL_SEX. The variable whose values are in the array (in this case, age) should be named immediately after the prefix (in this case, HD).
- It may be necessary for a procedure to call a function, and standards should be defined for naming them.
- Standards for coding impute statements should be set. If it is desirable to relate an imputation to the original edit specifications (and this is usually helpful), then case numbers should be assigned to every imputation case in the edit specifications document.
- Standards for code indentation should be established so the program is easily readable. All the examples in this manual are indented consistently and can be used as guides for establishing a standard.
- File naming conventions also should be established.

See also: Programming Standards

Code Edits of Individual Data Items

Edit specifications need not be assigned to team members in any specific order. It is recommended that the simpler edits be coded first, especially if the coding is being done by inexperienced program mers. In this manner, they will gain experience and will later be able to handle more complex edits.

Team members should code the edit specifications and review the code with the team leader. The team leader should then incorporate the new code into the CSPro program. All syntax errors should be corrected before more code is added to the main program. It is important to maintain earlier versions of the program as insurance against problems that may creep in.

Develop Comprehensive Test File

A comprehensive test file should be created; it can be made at any time after the edit specifications are finalized, and it can be done by anyone familiar with the data file. Each member of the team should create records to test all paths of his/her subroutines. These records can be combined into one test file in a logical sequence.

Test CSPro Program

The CSPro program should be thoroughly tested using the file with "invented" data. Earlier in this guide is a discussion of the `errmsg` statement and a technique for listing records before and after imputation. This technique (or a similar one) should be used to verify that imputations are being done properly and under the right circumstances. Each record that contains imputed items should be examined closely. Records containing no imputed items also should be checked to ensure that no imputations were needed.

When hotdeck imputations are used to correct invalid or inconsistent items, it is particularly important to ensure that the hotdeck arrays are being updated under the appropriate circumstances and with valid values. This is the most common error in program construction, and one way to avoid it is by requiring that the edit specifications explicitly indicate where, in the sequence of actions, hot decks should be updated.

The output file produced by the **Run** program step should be "clean," assuming imputations were performed to correct all invalid and inconsistent data. To verify that the CSPro program does not contain contradictory logic, that it corrects errors properly, and that it does not introduce any new errors, the **Run** program step should be rerun with the "clean" (output) data file as input. In this rerun, no errors should be found and the edit statistics should reflect this. If errors are found, then the logic of the CSPro program must be corrected.

Re-Test with Live Data

The CSPro program should be tested using real data with perhaps 2,000 to 3,000 records. The edit statistics should be examined closely. The number of imputations should be a very small percentage of records. If a large number of imputations are being done for any particular edit test, the code should be examined, and the imputations accounted for before proceeding. The edit statistics should be examined by subject-matter specialists also, so they can determine whether or not the rate of imputation is too high. If appropriate coding techniques are used with the `impute` statements, the imputations can easily be related back to the edit specification document.

Remember: Just because the CSPro program is free of syntax and logic errors is no guarantee that all the required editing is being done and is being done correctly! An error-free CSPro program cannot compensate for incomplete edit specifications. Among the test most commonly omitted tests when creating edit specifications are: making sure there is one and only one head of household, and making sure the head of household is of an acceptable age.

Begin Production Editing

If no changes to the CSPro program are needed, only the **Run** program step needs to be executed for each batch of data to be edited. If the data dictionary is changed, then you must re-compile the program before running it again. During production processing, edit statistics for each run should be checked to ensure that the number of errors and imputations are within the normal range.

As soon as enough batches have been edited to form a larger geographic area (perhaps all enumeration areas within a district), tabulations should be run at the higher geographic level (in this case, district) to see if any inconsistencies are apparent. If so, it will be necessary to pause and discover the reason for the inconsistencies. This may involve modifying and re-testing the CSPro program until the data are producing satisfactory results. Each time the CSPro program is modified, all previously-edited batches must be re-edited (always starting with the same original data files!) so that final data will all have been edited and corrected following the same logic.

Tabulation Applications

Introduction to Tabulation

Tabulation applications in CSPro allow you to tabulate data quickly and easily, producing basic frequency distributions and complex cross-tabulations from one or more data files. The only requirements are that:

- All the data files to be used in a single run share the same format
- A data dictionary exists that describes the data file(s)

When you create your application, you can use an existing data dictionary or you may create one as you create the application. You can produce most tables by way of the menu-driven, point-and-click, drag-and-drop interface, i.e., without using the CSPro programming language. The procedural language is available for the most complex kinds of tables.

This section contains the following information:

- Common Uses of Tabulation
- Capabilities of Tabulation
- Creating Tables
- Creating Tables by Geographic Area
- Parts of a Table
- Formatting Tables
- Printing Tables
- Tabulation Preferences
- Saving and Copying Table Data
- Table Post Calculation
- Run Production Tablulations
- Advanced Table Topics
- Table Tips and Tricks

Parts of a Table

Shown below are the terms used by the CSPro software to reference the parts of a table.

These are in "Table Print Format"

Left header | Center header | Right header

Table: This is the title - first line of the table

Subtitle: This is a subtitle - a text line below the table title

Stub Head - this is the text above the stub descriptions	**Sex - this is a spanner - a text line above several column headings**		
	Total - this is a column head	Male	Female
Relationship to Head			
Total	16,556	7,958	8,598
Head	3,312	2,169	1,143
Spouse	1,518	73	**1,445**
Child	5,097	2,429	2,668
Grandchild	1,832	908	**924- this is a data cell**
Parent	140	19	121
Other relative	3,671	1,812	1,859
Non-relative	810	463	347
Institutional member	176	85	91
Citizenship - this is a caption - a row without any numbers			
Total	16,553	7,958	8,595
Popstan-this is a stub - a row with numbers	**16,283**	**7,814**	**8,469**
Kiribati	119	71	48
Vanuatu	7	4	3
Tuvalu	94	43	51
Other	50	26	24

A Page note is a text line at the bottom of each page

An End note is a text line that only appears on the last page of a table

These are in "Table Print Format"

Left footer | Center footer | Right footer

Design/Data View Contents

- **Title:** Top line of table that usually describes its contents.
- **Subtitle:** An optional informational title below the (main) title.
- **Spanner:** An informational text box 'spanning' several columns.
- **Column Head:** Text box describing the contents of a column of data.
- **Stub Head:** Text box describing the general contents of the stubs (rows).
- **Stub:** Text line describing the contents of a row of data.
- **Caption:** Text lines interspersed with Stub lines but which do not have data associated with them.
- **Page Note:** A text line that appears at the bottom of each page of the table [a footnote on each page]

- **End Note:** A text line that appears at the bottom of the LAST page of a table [a footnote at end of table].
- **Area Caption [Not shown]:** When the area option is used for tabulations, this is a caption line that will be replaced by the text in the 'area name file'.
- **Boxhead:** This is the union of the Stub Heads, the Spanners, and the Column Heads

Additional Contents for Print View

- **Left, Center, Right Header:** Optional text printed at the top left, center, and/or right of each printed page (above the title).
- **Left, Center, Right Footer:** Optional text printed at the bottom left, center, and/or right of each printed page (below any footnotes).
- **Secondary Stub Head [Not shown]:** When stub text is displayed on both sides [left and right] of the print table, this text is displayed.

Additional Table Terms

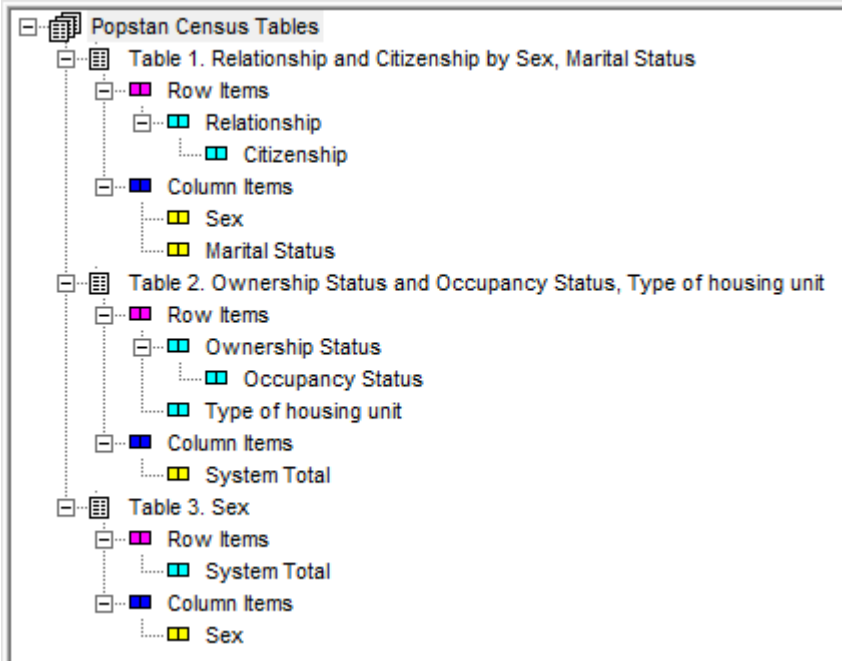
- **Tally:** Characteristics associated with the numbers that appear in a table. These include percents, averages, medians, etc.
- **Subtable:** When more than one set of categories is crossed with another characteristic, then each is considered a subtable. A table with only one set of stubs and column headings can also be considered as a subtable. (See below: a comma in the table title separates subtable item labels.). A subtable is a portion of a larger table that can have its own settings for universe, weights, value tallied and unit of tabulation different from other subtables in the same table. When hidden parts in the view menu is enabled, a colored box will be drawn around each subtable.

Table 1. Sex, Citizenship by Birthplace

	Birthplace		
	Total	Popstan Provinces	Countries
Sex	SUBTABLE		
Total	16,528	16,204	324
Male	7,946	7,796	150
Female	8,582	8,408	174
Citizenship	SUBTABLE		
Total	16,528	16,204	324
Popstan	16,259	16,079	180
Kiribati	118	117	1
Vanuatu	7	5	2
Tuvalu	94	3	91
Other	50	-	50

Tabulation Application Tree

When you create a tabulation application, the tables tree tab will be nearly empty—only a single table will exist, with the Row Item and Column Item icons there as placeholders. However, as you drag dictionary items over to the table workspace to build your tree, you will begin to see it populate. The tree below shows such progress after three tables have been developed.



Tabulation Application

This is the highest level node, also known as the root node. It represents the tabulation application.

Table

This is the second-tier tree node, just below the root. It represents each table defined in the application. There will always be at least one table in a tabulation application, although there is no maximum number. There are three tables in the tree above.

Row Item (parent)

This is a third-tier tree node, which will appear just below a table tree entry. It will always be present, even if no child row item has been placed below it. However, if a child column item appears on the table, then a "System Total" entry will appear as a placeholder, informing you that the column totals will appear in the row (see Table 3's Row Items for an example of this).

Row Item (child)

This is a fourth-tier tree node. It can appear in the tree either just below a Row Item parent, or beneath a Row Item child. This second scenario indicates that the data item is nested—in Table 1, **Citizenship** is nested below **Relationship**, and in Table 2, **Occupancy Status** is nested below **Ownership Status**. A Row Item can not have a depth of more than two data items. However, it can have any number of data items in sequence. For example, in Table 2 the totals for the **Ownership Status** Row Item will appear first, followed immediately by the **Type of Housing Unit** row totals. The values of these two Row Items are not dependent on one another, and more data items could have followed in the row.

Column Item (parent)

This is a third-tier tree node, which will appear just below the Row Item tree entries. It will always be present, even if no child column item has been placed below it. Similar to the behavior of the Row Item parent entry, once a child row item appears on the table, then a "System Total" entry will appear as a placeholder, informing you that the row totals will appear in the column (see Table 2's child Column Item for an example of this).

Column Item (child)

This is a fourth-tier tree node. It can appear in the tree just below a Column Item parent, or beneath a Column Item child. The behavior of sequential or nested Column Items is identical to the arrangement explained above for Row Items. There are no nested Column Items in the example tree above, but there are two column data items in sequence in Table 1, **Sex** and **Marital Status**.

Pressing **Ctrl+T** will switch the tabulation tree view between labels and unique names for the items.

Common Uses of Tabulation

Tabulation applications may be used for a number of different purposes. The statistical organization may generate tables for dissemination to outside users of the data, and the same organization may also generate tables to be used internally, as a tool for control and analysis of the data themselves. Some other common uses of tabulation applications are:

Design Table Stub Groups

In the initial stages of designing the tabulations that will eventually be generated from the final edited data, it is important to optimize the category breakdowns. Certain variables present few problems in this respect, either because the number of categories is inherently limited (e.g., sex) or the number of categories is limited to those pre-coded on the questionnaire (e.g., relationship). Other items, how ever, may require that a choice be made (e.g, age, occupation, educational levels, etc.) in grouping the available values. Tabulation applications may be used to examine the effects of various groupings before the final design is approved.

Examine Quality of Data and Design Edits

The design of proper consistency and validity edits is a difficult task, but the use of tabulation applications can help in determining the accuracy and completeness of the edit program. In fact, effective quality control requires that users verify and document the results of the edits. By using tabulation applications to generate the same tables from the pre- and post-edit data, the user can compare the cell values to ensure that all anomalies have been corrected and that the edits have not distorted the distribution of data values across categories.

Test Edits

Although CSBatch automatically produces edit statistics which report the extent of the editing it performed on a file, the subject-matter specialist and/or computer programmer has the option of using tabulation applications to analyze a data file before and after editing to examine the extent of the changes to a file, and to review relationships between data items in the edited file. Tabulation applications can be used to determine whether the edits did the job they were intended to do.

Test Tables

Tabulations that are destined for publication and for distribution to outside users must be carefully verified to ensure that the values presented are without error. Tabulation applications are currently designed for use as a custom tabulation package but it can also be useful as a means of comparing the results produced by other tabulation software. In this case, the relative rigidity and transparency of tabulation specifications means that there will be virtually no errors attributable to programming faults; if differences are encountered between the tables produced by CSPro and tables produced by other software, the tables produced by CSPro are likely to be closer to the "true" counts. In any case, tabulation applications can be an effective tool to create publication-quality tabulations or for cross-checking numbers produced by other software.

Capabilities of Tabulation

Cross Tabulations

Cross tabulations can display relationships between two or more data items. You may have both dependent and independent variables in each dimension [row and column].

Table 1. Mother living and Literacy by Urban/Rural and Sex

Mother Alive and Literacy	Urban/Rural								
	Total			Urban			Rural		
	Sex			Sex			Sex		
	Total	Male	Female	Total	Male	Female	Total	Male	Female
All persons									
Literacy									
Total	16,528	7,946	8,582	9,707	4,595	5,112	6,821	3,351	3,470
Literate	13,890	6,836	7,054	8,830	4,229	4,601	5,060	2,607	2,453
Illiterate	2,638	1,110	1,528	877	366	511	1,761	744	1,017
Mother living									
Literacy									
Total	12,064	5,834	6,230	7,202	3,435	3,767	4,862	2,399	2,463
Literate	10,481	5,106	5,375	6,690	3,189	3,501	3,791	1,917	1,874
Illiterate	1,583	728	855	512	246	266	1,071	482	589
Mother deceased									
Literacy									
Total	4,464	2,112	2,352	2,505	1,160	1,345	1,959	952	1,007
Literate	3,409	1,730	1,679	2,140	1,040	1,100	1,269	690	579
Illiterate	1,055	382	673	365	120	245	690	262	428

For example, this table has Urban/Rural and Mother Alive as independent variables; Sex and Literacy as dependent variables. In addition, this table could be produced by 'area'; usually this is a geographic item but it could be any item that meets certain conditions. The 'area' item is basically an extra tabulation variable.

The user can elect to display tabulations in terms of actual counts or as percentages of the total, and has the option of counting special and/or undefined values. The counts in tabulations may be unweighted or weighted. Users can produce a table on a subset of a file using the universe option, or select an item or numeric value to be used as a weighting factor during tabulation.

The selection of an item as a row or column variable will affect the table layout.

See also: Create a Table

Tabulate Counts or Percents

In addition to the quantitative numbers generated in your tabulations, you can choose to show the distribution of values for an item as a percentage of either row or column totals, or as a percentage of the table total. You can also choose to

show values only as percentages and suppress the actual numbers. This is useful when you have a small data set and prefer not to display values which might identify individual cases.

See also: Include Percents

Tabulate Values and/or Weights

Tabulation allows the use of a data item or a constant value as a weighting factor during tabulation. This is particularly useful in the case of a survey, where the weight assigned to each case or observation in the sample must be taken into account in order to produce numbers representative of the whole. If no weight value is specified, weight is assumed to be "1".

In the same fashion, values (rather than counts) may be tabulated; like weights, values are numeric data items or constants with or without decimal positions. When a value is specified for tabulation, the effect is that of cumulative addition of the specified value into the cell at the intersection of the row and column coordinates. If you leave the value item blank, a value of 1 will be added into the appropriate cell during tabulation. Tabulating values can be used, for example, when a variable such as "number of children born" has been collected for each woman in the population and you wish to tally the total number of children born rather than tallying the number of women with a particular number of children. In this case you would use the "number of children" variable as the value to tally so that for each woman, the number of children born to that woman would be added to the appropriate cell.

If both value and weight are specified for a given table, the specified value is first multiplied by the specified weight and the **product** of this multiplication is then added to the cell in question. This feature is useful in the case where the unit of observation has a weight, and one of the data items to be tabulated is a quantitative value. An example might be a fertility survey, where each female of child-bearing age carries a sampling weight, and where some of the data items might be "Number of children ever born," "Number of children surviving," etc. Tabulation applications easily permits the accumulation of weighted values to provide the total number of children ever born for all females in the sample.

If the weight or value includes decimals, the decimal may be explicit or implicit, but the dictionary definition should reflect the correct situation. The weight may be assigned to just the current subtable, the entire table or all the tables currently defined in the application.

See also: Add Weights to a Table, Tabulate Values Instead of Frequencies

Produce Summary Statistics

Tabulation applications allow the inclusion of summary statistics in the tables. Available statistics include Mean, Mode, Median, Minimum value, Maximum value, Standard Deviation, and Variance, as well as Percentiles (N-tiles). Most of these give information about the distribution of values for the data item. These do not make sense for some items so you should only use them if you have some understanding of their meaning and purpose. For example, none of these measures have any meaning for an item such as sex.

Mean, Mode and **Median** are measures of central tendency. These are measures of the location of the middle or the center of a distribution. The definition of "middle" or "center" is purposely left somewhat vague so that the term "central tendency" can refer to a wide variety of measures. The mean is the most commonly used measure of central tendency. Mean is the average value of the item, mode is the most common value occurring in the data, and median is the middle value (half of the values occur above this value and half below, e.g., 50th percentile).

Minimum and **Maximum** are the lowest values and highest values, respectively, found for the data item.

Standard deviation is a statistic that tells you how tightly all the various values are clustered around the mean in a set of data. This measure is the square root of the variance.

Variance is a measure of how spread out a distribution is. It is computed as the average squared deviation of each number from its mean.

N-tiles are values that divide the data values into N groups, each of which contain 1/N of the values. If N is two, then this measure is equivalent to a median.

See Also: Tally Attributes for a Variable

Restrict a Universe

When you define a universe, CSPro will only tabulate data records in the questionnaires that meet the conditions stipulated by you. The universe specification acts as a filter, as the tables produced use only a subset of the data file's records. Therefore, values in the table may be lower than they would be with no universe specified, since the universe restricts the data available for tabulation. Note that each table has its own universe definition, but that a given universe specification may be extended to all tables in an application.

See also: Define a Universe for a Table

Format Tables for Printing

The layout and presentation of the text and numbers in a table is generally referred to as the "Table Format". CSPro, through various menus, gives the user almost complete control over all aspects of the Table Format. These include the text font, its colors, its indentation, its alignment, its borders and the text itself. Other formats control the presentation of numbers, headers and footers on the printed page, and margins.

See Also: Formats for Parts of a Table, Formatting Row, Column, or Cell Data, Formats for a Table, Formats for a Tabulation Application, Formats for Printing

Load and Save Formatting Preferences

CSPro has a predefined set of formatting standards built into the system; these would be considered "Defaults." These include the text font, its colors, its indentation, its alignment, etc. Users have the option of creating their own defaults called "Preferences." These Preferences can be saved and reused for other applications. They may also be shared among multiple users working on the same publications in order to keep consistent formatting for all tables produced by that team.

See also: Preferences and Default Formats, Modifying Preferences, Load and Save Formatting Preferences

Produce Tables by Area

The "Area Processing" feature groups table data according to areas that you define. For example, if you are producing tables for industries, you may want to output the data according to industry type (Agriculture, Mining, Fishing, etc.).

One of the most common uses of area processing is for geography. Whatever items you choose to use for your area, they must be part of the questionnaire identification or items on a singly- occurring record. Area processing is in addition to any row/column items selected.

To use this feature, you must first create an area names file, which defines the levels of the area and assigns text names to the numeric code. The Area Processing applies to all tables in your application.

See also: Area Processing, Create an Area Names File, Area IDs Dialog Box, Area Names File

Save Tabulations in Different Formats

CSPro lets you select an entire table or parts of a table. The tables can be saved in several formats:

- **CSPro Tables (.tbw)**: Lets you save tables so they can be used later by the CSPro Table Viewer.
- **Rich Text Format (.rtf)**: Lets you save your tables so they can be used later by word processors such as Word.

You can open the (*.rtf) in your word processor, and the table will appear in the word processor's table format.

- **HTML files (.htm)**: Lets you save your tables so they can be later incorporated into Internet applications in table format.
- **Tab-delimited (.other)**: Lets you save your tables so they can be used later by a spreadsheet such as Excel. You can open the file in your spreadsheet, and the table will appear as a matrix of cells with columns lined up as you would expect.

See also: Save Tables for the Table Viewer, Copy Table to Other Formats

Copy Table to Other Formats

In addition to saving a table file in a particular format (as described in the preceding section), the user can use standard Windows copy and paste functions to copy all or part of a table directly into a word processor or spreadsheet. If only a portion of a table (i.e, not all rows and/or columns) is highlighted, CSPro will automatically include stub and column header text for only the affected rows/columns.

See also: Save Tabulations in Different Formats, Select and Copy Table Data to Other Applications

Copy and Paste Table Specification

CSPro lets you copy the entire table specification (before you run it against a data file) to the clipboard. The table specification includes the variables in the table, tally attributes and formatting information. You can then paste the table into the current tabulation application or into a different tabulation application.

This feature is very useful for moving tables from one application to another. If you have many tables to produce, you can divide them into several applications each with a small number of tables, then use copy and paste table specification to assemble them into a single application at the end. If you do this, you must be careful to use the same data dictionary in each application.

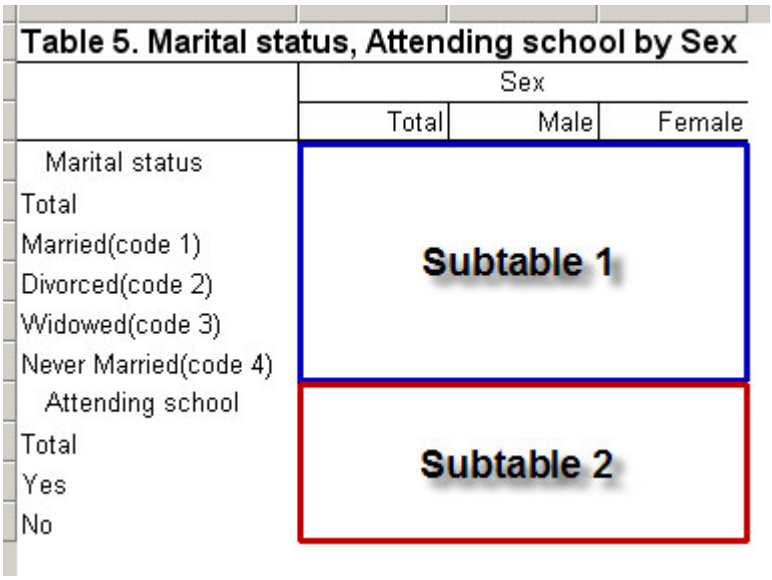
This feature is also useful if you need to add a table in the same application which is very similar to an existing table.

To copy or paste a table specification:

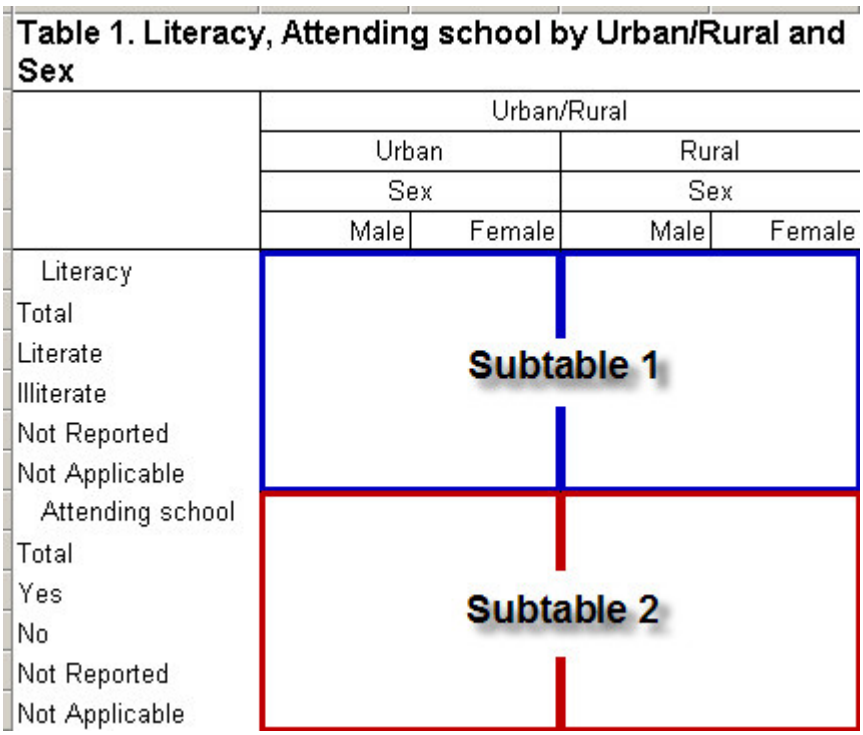
- From the Tabulation Tree tab, right-click over any of the tree entries. A popup dialog box will appear. Select **Copy Table Spec** or **Paste Table Spec**.
- From the **Tabulation** menu, select **Edit**, then **Copy Table Spec** or **Paste Table Spec**.
- From the table itself, right-click anywhere over the cells of a table. A popup dialog box will appear. Select **Copy Table Spec** or **Paste Table Spec**.

Create Multiple Subtables

Subtables are themselves tabulations nested within a larger tabulation -- hence the prefix 'sub', meaning part of a larger object. Each subtable can be made up of one to four variables (two for row and two for column). Each subtable has one independent variable in the rows and one independent variable in the columns. It may optionally have one dependent variable in the rows and one dependent variable in the columns. In the first example, each subtable has only two variables, one row and one column.



In the second example, there are three variables in each subtable, one row variable plus two column variables (the independent variable Urban/Rural and the dependent variable Sex).



Subtables are created automatically as the user builds the table. Every time an independent variable is dropped onto the rows or columns of the table, new subtables are created as appropriate.

Subtables are designated as part of the larger tabulation using colored outlines when viewing hidden parts. Each subtable has the same properties available to the entire tabulation, i.e., universe, weight, value tallied and unit of tabulation. These properties may be set differently for each subtable within a table.

See Also: Add a Variable to a Tabulation, Using Subtables

Change Unit of Tabulation

For most tabulations the **unit of tabulation** is easily determined. For example, the unit of tabulation for a table of **Age** by **Sex** would be the **Person** record. A table of **Householder Tenure** and **Number of Rooms** would be tallied for each **House** record. In cases where all items in the table are contained on the same type of record, the unit of tabulation could only be that record itself. At times, tables require inputs from two different record types (or two different groups of data). The choice of the unit of tabulation will affect the numbers in the table. CSPro allows this selection if there is any "cross-group" tallying in the table.

See also: Tally Attributes for a Table, Changing the Unit of Tabulation

Tally Items from Related Records

Data items on different records are often linked or related via codes on each of the records. For example, in a labor force survey there may be an initial roster of information about each household member followed by an additional set of questions for persons 15 years old and over. If a table involves age and sex from the initial roster and an economically active recode from the supplemental form, the data must be somehow linked so that the tally is done correctly. CSPro allows the definition of relations in the data dictionary, which will link these two related records for the purpose of creating cross tabulations.

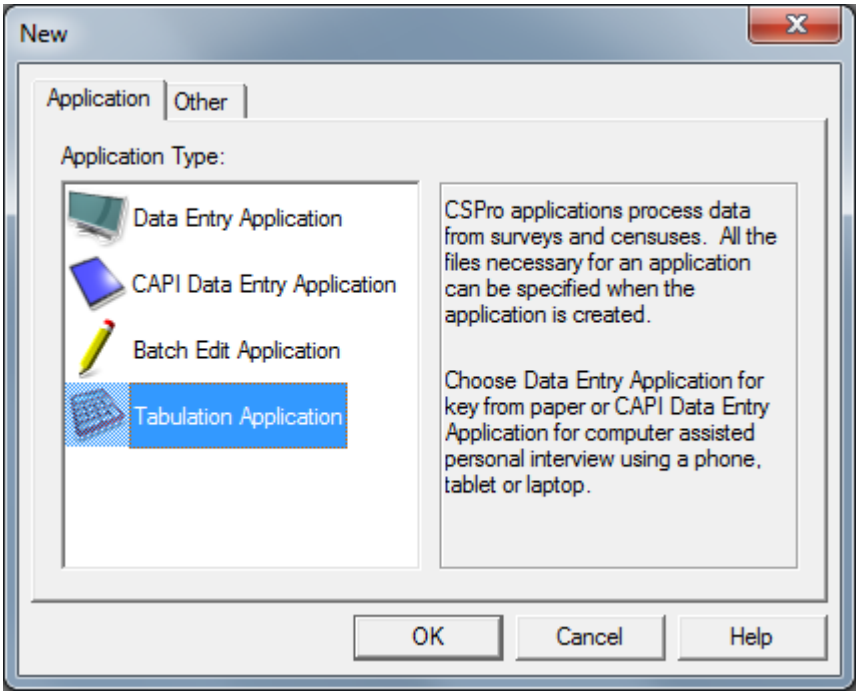
See also: Relation Description, Tabulations Using Relations

Creating Tables

Create a New Tabulation Application

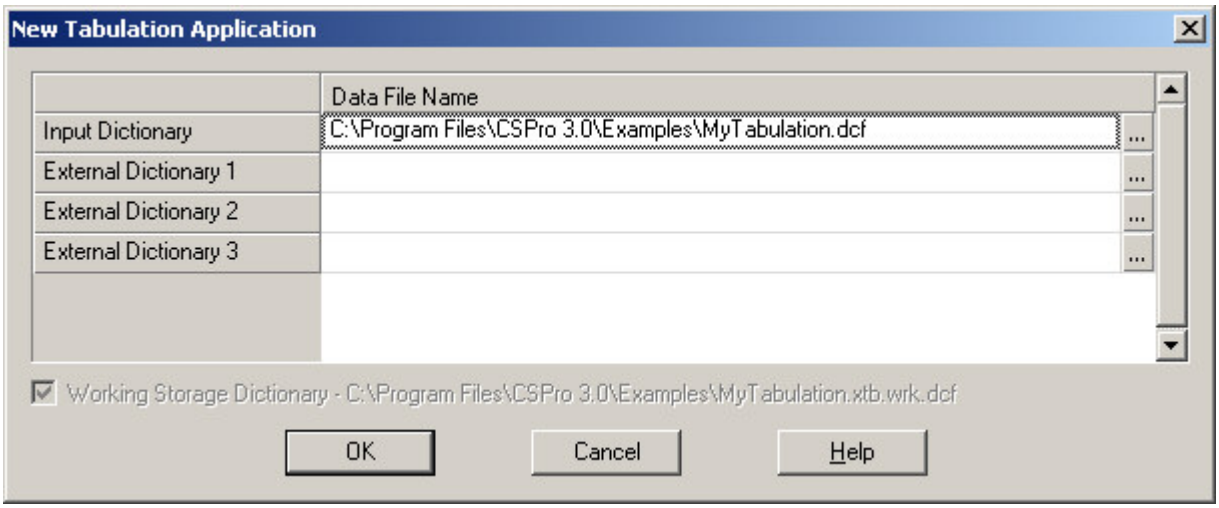
To perform tabulations, you will need a data file, and a data dictionary to describe the file. If you do not have a data dictionary, you will need a written description of the data file structure and organization. You can create the CSPro data dictionary as you create the new Tabulation application.

- Click on the toolbar, or from the **File** menu, select **New**. The following dialog box will appear.



- Select **Tabulation Application** and press **OK**.

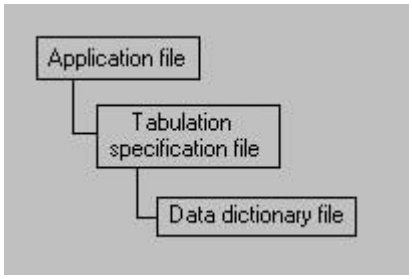
- A file dialog box will appear. Enter the name of the application file. Make sure you are located in the folder where you want to place the application files. Then press **Create**. The following dialog box will appear.



- A default name of the data dictionary describing the data file is given. You can use this name or change it. If you give the name of a dictionary file that already exists, that data dictionary will be used by the application. If you give the name of a dictionary that does not exist, a new data dictionary will be created.

- If you are using an existing CSPro data dictionary, you may now start creating tables. If you are creating a new CSPro data dictionary, you will need to enter information into the dictionary about records, items, and values before you can create tables.

Tabulation applications consist of the following files:



- Tabulation Application file (.XTB)
  Specifies all other files contained in the application and includes other application information.

- Table Specifications file (.XTS)
  Contains variable names and other parameters which define the tables in the application.

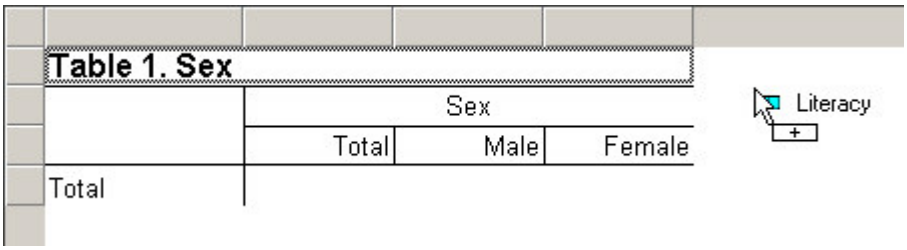
- Data dictionary file (.DCF)
  Contains the physical format of the data file(s) to tabulate.

See also: CSPro Initial Screen Layout, Introduction to Data Dictionary

Create a Table

To create a table, do the following:

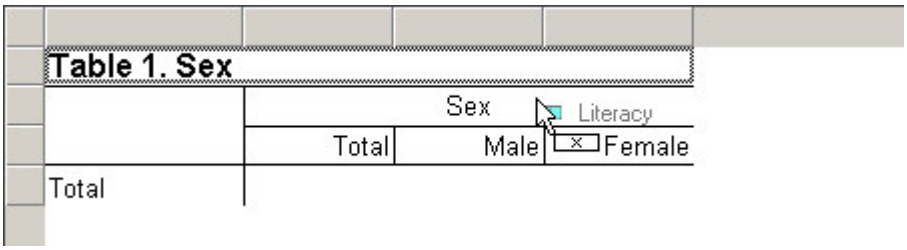
- Select the Dictionary [Dicts] tab to make the dictionary file structure visible.
- Expand the tree until the item(s) you wish to use for row or column variable(s) appears on the tree.
- Drag the desired dictionary item and drop it in the table window. Where you drop it on the table will determine whether it is used as a row or column variable. For a row item drop it on the left side below the generated "Table nn". For a column drop it on the top but a little below the generated "Table nn". If you imagine a diagonal line connecting the top-left corner of the table window to the bottom-right corner, dictionary items dropped below and to the left of the line will become row variables and items dropped above and to the right of the line will become row variables.
- This can be repeated with additional items. The key to placement for additional items is whether they are dropped next to or on top of existing items. This is indicated by the "+" or "x" displayed with the item while it is being dragged.



An item dropped next to an existing item is indicated by the "+" while dragging.

Table 1. Sex, Literacy						
	Sex			Literacy		
	Total	Male	Female	Total	Literate	Illiterate
Total						

The dropped item will be appended to the existing item, e.g., Total, Male, Female followed by Total, Literate, Illiterate.



An item dropped on top of an existing item is indicated by the "x" while dragging.

Table 1. Sex and Literacy									
	Sex								
	Total			Male			Female		
	Literacy			Literacy			Literacy		
	Total	Literate	Illiterate	Total	Literate	Illiterate	Total	Literate	Illiterate
Total									

The dropped item will be crossed with the existing item to make sub-groupings, e.g., Total, Literate, Illiterate under Total, Male and Female.

- To delete a row/column variable, left-click on one of its value names, then drag and drop it back on the dictionary tree. In the 'x' example above, if "Sex" is deleted, then "Literacy" will also be deleted. However, "Literacy" can be deleted without affecting "Sex".

The rows/columns that are created when you add an item to a table come directly from the value set defined in the dictionary for that item. If you would like to have different rows/columns created for a particular item, you can either modify the value set in the dictionary or create an additional value set for the item and use the new value set instead. To use a different value set, rather than dragging the item itself onto the table, drag the value set instead. For items with more than one value set, the value sets appear underneath the item in the dictionary tree. For more information see Implications of Data Dictionary Value Sets.

Optional Definitions

• Define the universe for the table.

• Define the area for the table.

• Select the value Item or constant you want to tabulate.

• Select the weight Item or constant

• Add percentages.

When two variables are selected for the same dimension (row or column), the first one selected becomes the independent variable and the second one becomes the dependent variable. (See the discussion on row/column variables for more details.)

See also: Tabulations, Add a Variable to a Tabulation, Remove a Variable from a Tabulation

Create Tables with Multiple Variables

Row variables are those dictionary items or value sets which appear in the rows of the table. Column variables are those dictionary items or value sets which appear in the columns of the table. Every table must have at least one row or column variable specified; any given subtable can have a maximum of two row variables and two column variables. The number and disposition of row and column variables will affect the type of table generated.

- **1 Row/0 Column Variables**
  If a table consists of one row variable and no column variables, the system will produce a table with the variable's categories as rows and frequency counts as the only column.

- **0 Row/1 Column Variable**
  If a table consists of no row variables and one column variable, the system will produce a table with the tabulation categories in the columns and totals in the single row of the table.

- **1 Row/1 Column Variable**
  If a table or subtable consists of one row variable and one column variable, the system will produce a normal cross-tabulation, with the tabulation categories of each variable in row or column, as appropriate. By default, totals will appear in the left-most column and in the top-most row (it is possible, however, to change the position of the total).

- **2 Row/Column Variables**
  When a table or subtable is designed with two variables or value sets in the row and/or column, one of each pair is considered to be the independent [major] variable, and the other is considered to be the dependent [minor] variable. The tabulation categories of the dependent variable appear nested within the categories of the independent variable. Totals for a dependent variable appear as the topmost row or left-most column within each tabulation category of its independent variable.

Because there are effectively no limits on the number of rows and columns in a table, the combination of two variables/value sets can produce tabulations which will be extremely difficult to view and to understand. Users should give careful thought to the placement of variables and value sets in rows and columns, particularly when one or more of the items has a large number of tabulation categories. It is almost always easier to manipulate tables with large numbers of rows than those with the same number of columns.

Whenever the area processing function is invoked for a table set, the area levels are included as additional row categories within which the other row variables are displayed.

Subtables are created by adding variables to the bottom of the existing rows or to the right of the existing columns. When a subtable is created it is usually outlined in color to designate this fact. This is an example.

Table 1. Marital status by Sex, Literacy

Marital status	Sex			Literacy				
	Total	Male	Female	Total	Literate	Illiterate	Not Reported	Not A
Total								
Married								
Divorced								
Separated								
Widowed								
Never Married								

Notice these are like two separate tables side-by-side; 1) Marital Status by Sex and 2) Marital Status by Literacy. There is a total of 6 columns (3 for Sex **plus** 3 for Literacy). Note also the wording in the generated title.

Compare the above to the following:

Table 1. Marital status by Sex and Literacy								
Marital status	Sex							
	Total			Male			Female	
	Literacy			Literacy			Literacy	
	Total	Literate	Illiterate	Total	Literate	Illiterate	Total	Litera
Total								
Married								
Divorced								
Separated								
Widowed								
Never Married								

This is a single table (1 row variable and two column variables) with a total of 9 columns (3 for Sex **times** 3 for Literacy). Note the wording in the generated title.

Implications of Data Dictionary Value Sets

Within the data dictionary, the user can define multiple value sets for each data item. For use in Tabulation applications, these sets may have value names for both individual values and value ranges. If one table requires five-year age groups but another uses single years of age, both requirements can be met by having two value set definitions for age.

If an item in the dictionary has more than one value set, you can choose which of the value sets to use when adding that item to a table. If you drag the item itself from the dictionary, CSPro will add the first value set for that item to the table. However, if you drag one of the value sets underneath the item in the dictionary tree window, CSPro will use the value set that you dragged.
When creating a value set, you should take into account the following:

- Value labels must be given for the value set since these appear as text (row or column) in the tables. If no label is given, the corresponding row/column label will be blank.
- Counts associated with the value labels are based on the item values defined for the label. This might be a single value, e.g., Male (sex=1); a range of values, e.g., Under 5 years old (age from 0 to 4); disjoint values, e.g., Teacher (occupation=23, 42, or 67); or some combination of these. That is, each value in the data file that falls into a particular range/value in the value set will cause the count to be incremented for the corresponding cell in the table.
- Counts are ONLY made in the table for item values defined in the value set. Values in the data file that are not defined in the value set are not counted. In the example below, the 'Marital Status' value set is missing codes 2 and 3 from its value set and the total reflects only the values listed in the value set. The exception to this rule is when Special Values are included in the Tally Attributes for the table or subtable.

N	Value Set Label	Value Set Name	Value Label	From
	Marital status	P05_MS_VS1		
			Married	1
			Widowed	4
			Never Married	5

N	Value Set Label	Value Set Name	Value Label	Fr
	Marital status	P05_MS_VS1		
			Married	
			Divorced	
			Separated	
			Widowed	
			Never Married	

Table 5. Marital status by Sex

Marital status	Sex		
	Total	Male	Female
Total	23,472	11,587	11,885
Married	6,985	3,439	3,546
Widowed	788	149	639
Never Married	15,699	7,999	7,700

Table 6. Marital status by Sex

Marital status	Sex		
	Total	Male	Fem
Total	24,271	11,787	12,4
Married	6,985	3,439	3,5
Divorced	799	200	5
Separated	-	-	
Widowed	788	149	6
Never Married	15,699	7,999	7,7

- In the current version of the software, alphanumeric items CANNOT be tallied.
- By default, certain summary statistics (median, n-tiles and mode) are calculated using the counts in the values/ranges in the value set. The accuracy of these summary statistics is highly dependent on the number and size of values/ranges in the value set. If the range for a particular variable is large and there are a small number of categories in the value set, these statistics will be inaccurate. For best results, when calculating median, mode and n-tiles, use either single values for discrete variables or small, uniformly sized ranges for continuous variables. Note that by setting the appropriate options for n-tiles and median, you can use different categories for the n-tile/median calculation than those in the value set. See Tally Attributes for a Variable for details.

See Also: Include/Exclude Special Values in a Variable, Tally Attributes for a Table, Value Sets Description, Create a Table, Add Summary Statistics to a Table, Tabulate categories with disjoint values, Debug Table Totals

Tabulate Items with Multiple Occurrences

If an item used as a row or column variable is defined as having two or more occurrences, the following conditions apply:

- If the parent item [i.e., the one described as occurring `n` times] is dragged to the table as a row or column variable, the variable name will be shown **without parentheses**.
- If one of the occurrences of an item is dragged to the table as a row or column variable, the variable name will be shown with a number **in parentheses**.

For example, `MyItem` contains two occurrences, `MyItem(1)` and `MyItem(2)`. If `MyItem` (the parent item) is dragged from the dictionary tree to the table as a row or column variable, no parentheses will appear with the name `MyItem` in the table heading because no specific occurrence of `MyItem` was requested. However, if `MyItem(1)` is dragged from the dictionary tree to the table, then the item name `MyItem` will be shown with the identifying number in parentheses —`MyItem(1)`—because the user selected a specific instance [occurrence] of the variable.

If you choose an item with occurrences as a row or column variable, **all** occurrences of that item will be tabulated across **all** corresponding records in the data file. For example, if you choose `MyItem`, both `MyItem(1)` and `MyItem(2)` would be tabulated across all records.

If you choose a specific occurrence of an item as a row or column variable (e.g., `MyItem(2)`), only that occurrence will be tabulated across all corresponding records.

You may also use items with occurrences as the value or weight item. Note that when the unit of tabulation is not the

repeating item, you will get errors when running your table unless you specify the occurrence number to use (e.g., `MyItem(2)`) or change the unit (e.g., to `MyItem`).

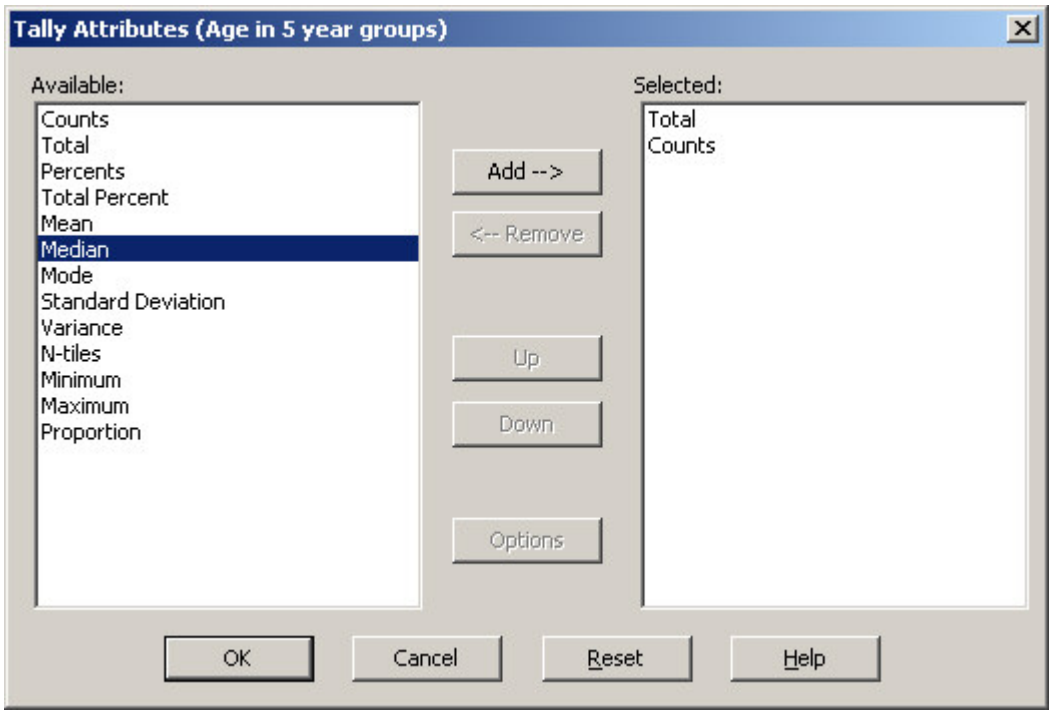
See Also: Create a Table

Tally Attributes for a Variable

The Tally Attributes (Variable) Dialog Box allows you to choose which additional calculations should be displayed for a variable in a table. Using Tally Attributes (Variable), you can display totals, percents and summary statistics for each variable in the table.

To bring up the Tally Attributes (Variable) Dialog Box, right-click on a stub or column header for the variable whose attributes you wish to modify and choose "Tally Attributes (<Variable Name>)".

Unlike the Tally Attributes (Table/Subtable) Dialog Box, the settings made in the Tally Attributes (Variable) Dialog Box apply to the chosen variable and not to the rest of the table or subtable.



The Tally Attributes (Variable) Dialog Box contains two lists: on the left is the list of available calculations and on the right is the list of calculations selected for the current variable.

To add a new calculation, select the type of calculation in the list on the left by clicking on it and then click the "Add" button. The selected calculation will appear in the list on the right.

Each calculation has its own set of options such as percent type for percents, number of tiles for n-tiles, etc. To modify the options for a calculation, first select the calculation by clicking on it in the selected list, and then click the "Options" button to bring up the options dialog box.
To remove an existing calculation, click on the calculation in the list of selected calculations on the right and click the "Remove" button. The calculation will be removed from the list of selected calculations.

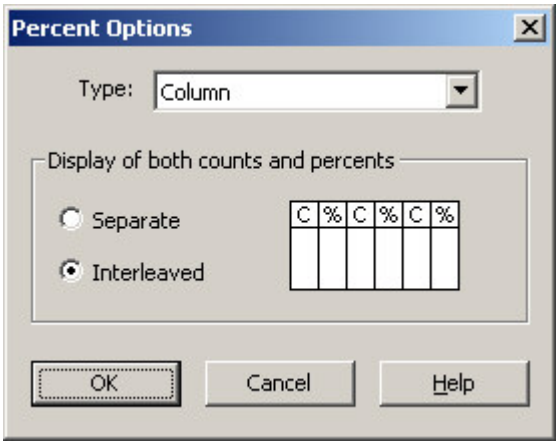
The order that the calculations appear in the selected list will be the same order in which they appear in the table. That is, if you have 'Total' before 'Median' in the selected list, then in your table, you will have a "Total" row or column before the row/column containing the median. To change the order of the calculations, choose a calculation in the selected list by clicking on it and click either the "Up" or "Down" button to change its position in the list.

The following calculations are available:

Counts: Displays the counts (frequencies) for each value in the value set. Adding counts to the variable will add a row or column to the table corresponding to each value in the value set for the variable.

Total: Displays a single row or column with the total for all values in the value set.

Percents: Displays the counts for the variable as percentages of the row total, column total or subtable total. Adding percents to the variable will add a row or column to the table for each value in the value set for the variable, but unlike 'Counts', the results are displayed as percentages. Percents have the following options:



Type: Selects how to calculate percents for the selected variable. Specifically, this gives the 'base' (100%) or denominator for the percent. One of three options can be selected:

- **Total** - use the total count for the subtable as a base for each percent.
- **Row** - use the total count for the subtable row as the base for the percents in each row. If there is a total column, it will contain '100 %'s.
- **Column** - use the total count for the subtable column as the base for percents in each column. If there is a total row, it will contain '100 %'s.

Interleaved/Separate: Selects whether or not the percents are alternated with the counts. For percents to be interleaved, counts must be present directly before or directly after the percents in the selected list. When percents and counts are interleaved, the order in which the counts and percents appear in the selected list determines whether the first column/row will be a percent or a count – i.e., if counts appear first in the selected list, then the first row or column will be a count followed by a percent.

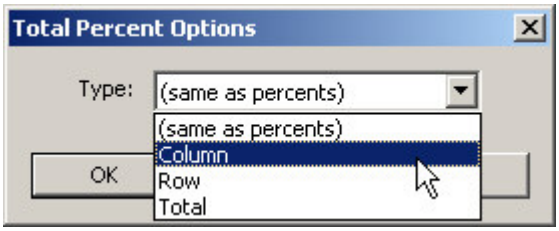
Sex					
Total	Percent	Male	Percent	Female	Percent

Interleaved percents

Sex					
Total	Male	Female	% Total	% Male	% Female

Separate percents

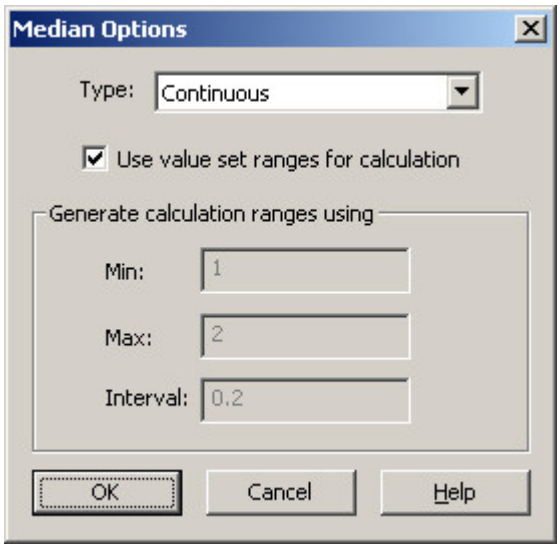
Total Percent: Displays a single row or column with the total for all values in the value set expressed as a percentage of the row, column or subtable total. Total percent has the following options:



- **Type:** The type of percent (row, column or total as for percents above). You may also choose '(same as percents)' to force the total percent to use the setting from the percents.

Mean: Displays the arithmetic mean (average) for each row or column.

Median: Displays the median for each row or column. Median has the following options:



- **Type:** Specifies the type of variable – either discrete or continuous. This should be set to "Discrete" for discrete variables (such as number of rooms) and to "Continuous" for continuous variables (such as age or income). The median is approximated using linear interpolation based on a frequency distribution for the variable. For discrete variables, 0.5 is subtracted from the result to reflect the fact that values in a category are all equal to the lower limit of the category whereas for continuous variables values in a category are in between the lower and upper limits of the category.

- **Calculation ranges:** Controls which frequency distribution is used to calculate the median. If "Use value set ranges for calculation" is checked, then the median will be approximated using a frequency distribution for the value set that was dropped onto the table. Otherwise, you can specify the ranges to use for the distribution by entering the min, max and interval. CSPro will generate ranges of width 'interval' starting at 'min' and going up to 'max'. For example, a "min" of 0, "max" of 20 and "interval" of 5 will generate the following four ranges:

  0 up to but not including 5
  5 up to but not including 10
  10 up to but not including 15
  15 up to but not including 20

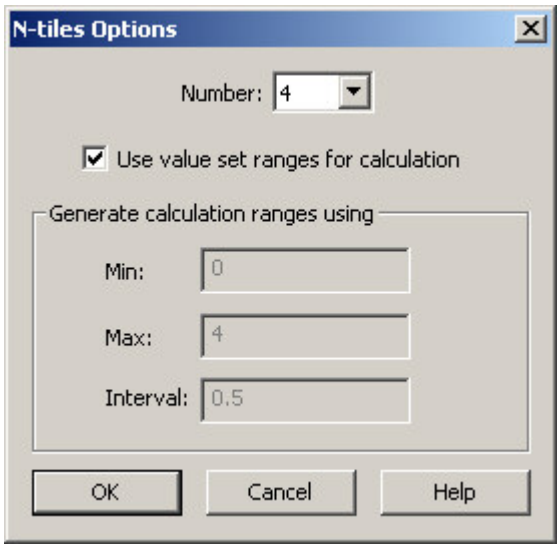
  If the value set you use does not have very many ranges, it is better to generate your own ranges in order to get a closer approximation of the median. For example, using a value set with only three categories for age such as 0-14, 15-49 and 49+ will generally result in a very poor approximation of the median, while using 5-year groupings will generally result in a good approximation. For an even better approximation, turn off "Use value set ranges for calculation" and specify a min of 0, a max of the largest age in the population and an interval of 1. For variables such as "income" or "price of a house", you may need to experiment with different sets of ranges to come up with a good approximation for the median.

Mode: Displays the mode (value with the greatest number of occurrences in the data set) for each row or column.

Standard deviation: Displays the standard deviation for each row or column.

Variance: Displays the variance for each row or column.

N-tiles: Displays percentiles for each row or column. N-tiles has the following options:



- **Number:** Sets the percentiles to show. Set it to 4 for quartiles, 5 for quintiles, etc.

- **Calculation ranges:** Controls which frequency distribution is used to calculate the n-tiles. If "Use value set ranges for calculation" is checked, then the n-tiles will be approximated using a frequency distribution for the value set that was dropped onto the table. Otherwise, you can specify the ranges to use for the distribution by entering the min, max and interval. CSPro will generate ranges of width "interval" starting at "min" and going up to "max". For example, a "min" of 0, "max" of 20 and "interval" of 5 will generate the following four ranges:

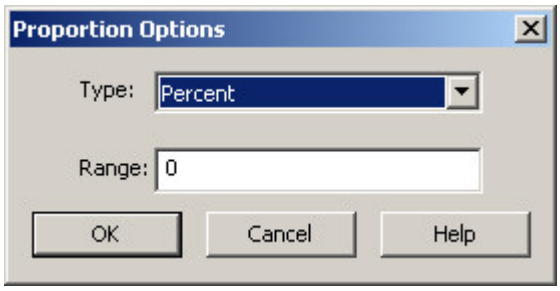
  0 up to but not including 5
  5 up to but not including 10
  10 up to but not including 15
  15 up to but not including 20

  If the value set you use does not have very many ranges, it is better to generate your own ranges in order to get a closer approximation of the n-tiles. For example, using a value set with only three categories for age such as 0-14, 15-49 and 49+ will generally result in a very poor approximation, while using 5-year groupings will generally result in a good approximation. For an even better approximation, turn off "Use value set ranges for calculation" and specify a min of 0, a max of the largest age in the population, and an interval of 1. For variables such as 'income' or 'price of a house', you may need to experiment with different sets of ranges to come up with a good approximation for the n-tiles.

Minimum: Displays the minimum value for each row or column.
Maximum: Displays the maximum value for each row or column.

Proportion: Displays the proportion of a specified subset of the counts for the variable as a percent or ratio of the total for the variable. You could also use proportion, for example, on the age variable, to display the percentage or fraction of the population with age between 15 and 49. You could also proportion on a yes/no variable such as 'owns a television' to show the percentage or fraction of the population that owns a television. The proportion is calculated by taking a subset of the values in the value set for a variable, computing the total for that subset and then dividing that total by the total for all values in the value set. The result may be displayed as a percent or as a ratio. The numerator used to calculate the proportion is the total number of values that match the specified subset of the value set and the denominator is the total number of counts for all values in the value set. For example, in the case of the proportion of people between 15 and 49, the numerator would be the total number of people between 15 and 49 and the denominator would be the total number of people of any age.

Proportion has the following options:



- **Type:** The type of proportion to display. It may be one of the following values:
  - Percent: Adds a single row or column to the table showing the counts for the values selected in the range as a percentage of the total.
  - Percent & Total: Adds two rows/columns to the total: the first showing the percent as above and the second showing the total number of counts for the values selected in the range (i.e., the numerator used to calculate the percent).
  - Ratio: Adds a single row or column to the table showing the counts for the values selected in the range as a fraction of the total.
  - Ratio & Total: Adds two rows/columns to the total: the first showing the ratio as above and the second showing the total number of counts for the values selected in the range (i.e., the numerator used to calculate the fraction).

- **Range:** Specifies the set of values whose counts will be used in the numerator of the percent or ratio and as the total. Range can be a single value (e.g., 3), multiple values separated by commas (e.g., 1, 2, 5, 17) and it may also include value ranges using colons (e.g., 1, 2, 5:10, 25:30).

To show the percentage of the population between 15 and 49 you would add "proportion" to the age variable, set the type to "Percent" and set the range to 15:49. To show the fraction of the population that owns a television, add "proportion" to the variable "owns a television", set the type to "Ratio" and the range to 1 (assuming 1 means "yes").

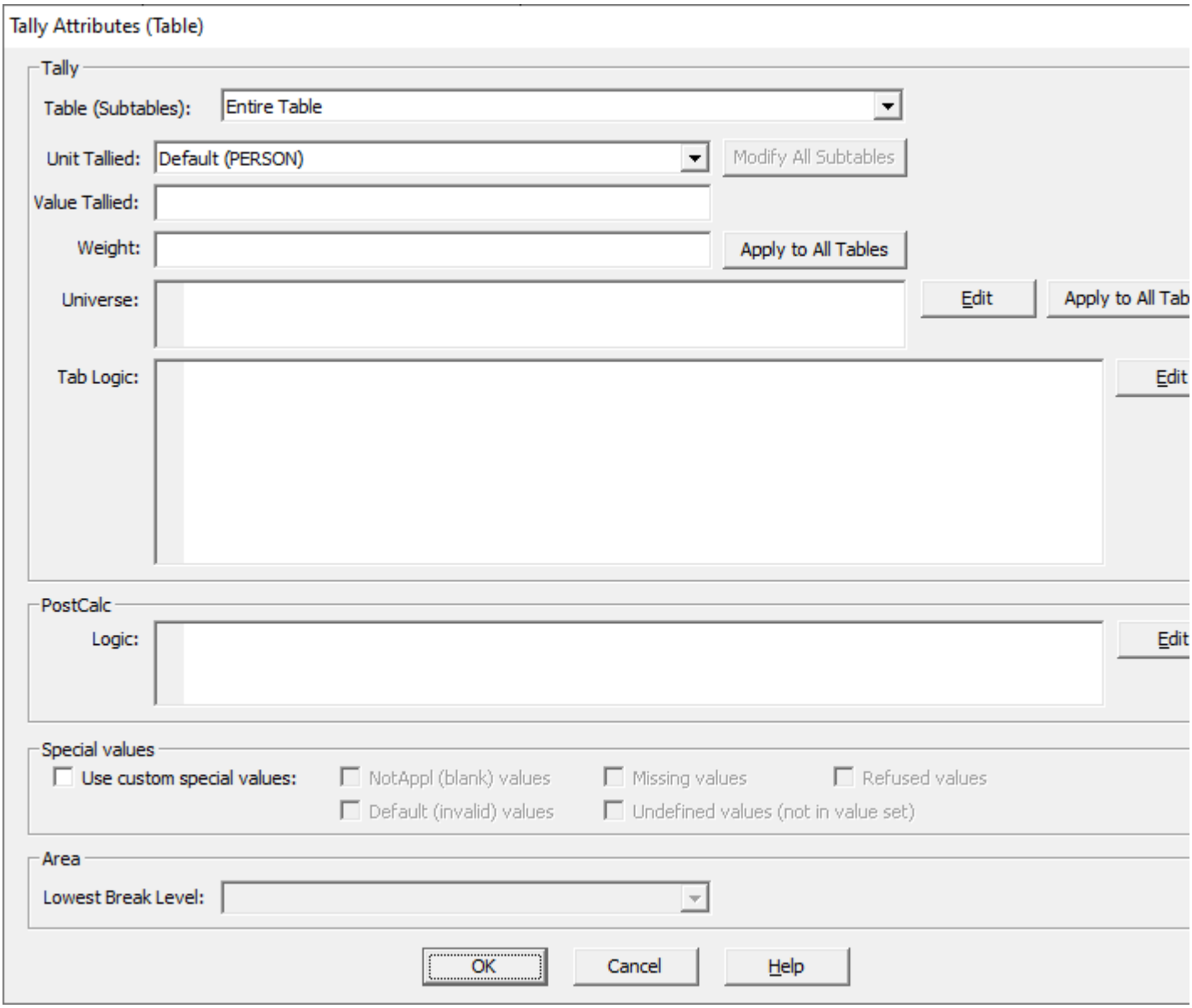
Note that the values calculated for median, mode and n-tiles are dependent on the value set used. For more information see Implications of Data Dictionary Value Sets.

See Also: Hide or Change the Position of the Total, Add Percents to a Table, Add Summary Statistics to a Table

Tally Attributes for a Table

Click on in the toolbar or right-click on a table or subtable and choose Tally Attributes (Table) or Tally Attributes (Subtable) to bring up the Tally Attributes (Table/Subtable) Dialog Box. This dialog allows you to change the results of the tabulation of a table or subtable by adding weights, restricting the universe of values, changing the values tallied, changing the unit of computation and changing the way special values and areas are considered.

By default, CSPro counts or tallies once each time it finds an occurrence of a variable in your data set. That is, each time an occurrence of a variable is found, it looks at the value of the variable found, determines which cell of the table this occurrence falls in and increments the value in that cell by one. By changing the tally attributes, you can have CSPro change the amount that it increments the cell by (using value tallied or weight). You can also make CSPro ignore certain occurrences of a variable and not count them (using unit tallied or universe).



The Tally Attributes (Table/Subtable) Dialog Box contains the following settings:

Table (Subtables): Allows you to choose to modify the tally attributes of either the entire table or one of the subtables. Any changes made to Unit Tallied, Value Tallied, Weight, Universe, and Tab Logic will only apply to the selected table or subtable. Changes made to Special Values, PostCalc and Lowest Break Level always apply to the entire table.

Each subtable may have its own unique settings for Unit Tallied, Value Tallied, Weight, Tab Logic and Universe.

- If a weight or value tallied is specified for both the entire table, and for a subtable, then the two weights or values (table and subtable) will be multiplied together to compute the final weight or value for the subtable.
- If a universe is specified for both a subtable and for the entire table, then the two universes will be combined (using the *and* operator) to compute the final universe for the subtable.
- If a table has more than one subtable, then you cannot set the unit tallied for the entire table. In this case, there is no unit for the entire table. You must set the unit for each subtable. However, you can set the unit for all subtables in one step by clicking on the Modify All Subtables button. Note that Modify All Subtables only changes the unit for existing subtables. Newly-added subtables will be given the default unit unless you modify them explicitly after they are created.
- If Tab Logic is specified for both a subtable and for the entire table, the logic for the entire table will be executed for each subtable in addition to any logic for the subtable. The logic for the entire table will be executed before the logic for the subtable.

Note that if you choose Tally Attributes (Subtable), the subtable that you clicked on will automatically be selected in this list.

Unit Tallied: Allows you to change the unit of computation for the selected subtable. The unit of computation is the level, record or item in the dictionary that is counted for the tally. For example, when tallying the rent variable of the housing record, by default, the unit would be the housing record so each household would be counted once. However, you could tally the rent variable on a per person basis by setting the unit to the person record. This would count the number of *people* paying a given rent rather than counting the number of *households* paying a given rent. In most cases, the unit is left as the default.

You can modify the unit for all subtables in a table at once by clicking on the "Modify All Subtables" button. This brings up the following dialog box:

This dialog displays all units that can be applied to all of the subtables. Choose the unit from the drop down and click "OK". Potential units will not be included if they cannot legitimately be applied to one or more of the subtables. For example, if one subtable contains an item from a multiple record (e.g., "sex" on the person record), then only the multiply occurring record, and multiply occurring items on that record, are considered legitimate units to be applied to all subtables.

Value Tallied: Optional name of a numeric item in the dictionary whose value would be tallied into the data cells instead of the unit (one) tally. For example, setting the value tallied to 'children ever born' would count the actual number of children instead of counting the number of cases with a given number of children. This can also be a numeric constant.

Weight: Optional name of a numeric item in the dictionary that contains the inflation factor (weight) for a survey case. For each tally, this value will be tallied instead of the unit (one) tally. This can also be a numeric constant.

Universe: An expression that restricts the data used for tabulation. When you define a universe, CSPro will tabulate only those data records in the questionnaires that meet the conditions stipulated by you. The "universe" specification acts as a filter, as the tables produced use only a subset of the data file's records. Therefore, values in the table may be lower than they would be with no universe specified, since the universe restricts the data available for tabulation.

Tab Logic: Optional CSPro logic statements that may be used to modify the values of variables in the case during tabulation. This is used mainly for recodes of existing main dictionary variables into new variables used in the table. For information see Table Logic (tablogic). Logic may be entered directly in the associated edit box or you can click on the Edit button to bring up a larger window in which to modify the logic.

PostCalc: Optional CSPro logic statements that may be used to modify values in the table after the tabulation is complete. This logic can be used to add values to the table calculated from the tabulated data such as ratios. For more information see Introduction to Table Post Calculation. Logic may be entered directly in the associated edit box or you can click on the Edit button to bring up a larger window in which to modify the logic.

Special Values: Allows you to customize which special values are counted in the selected table. By default, only special values included in the value sets being tallied are considered. By checking "use custom special values" you can specify which, if any special values to include in the tabulation regardless of the special values contained in the value sets. For example, if the value set used in a subtable contains the special values *notappl* and *missing*, by default *undefined* and *default* values will not be included in the tabulation. Enabling custom special values and checking *notappl* while leaving the others unchecked will cause only *notappl* values to be included in the tabulation.

Lowest Break Level: Allows you to limit the geographical areas used for area processing for just the selected table. The lowest break level is the lowest geographical area that will be used in the table. For example, if your file includes data at the province, district and village level but you set the lowest break level to district, then only province and district will appear in the selected table while all three levels will appear in other tables in the file.

See Also: Create Multiple Subtables, Define a Universe for a Table, Add Weights to a Table, Tabulate Values Instead of Frequencies, Include/Exclude Special Values in a Variable

Add, Insert, and Delete Tables

Add Table

Any table added to an existing set of tables will always be placed **after** the last existing table. If you want a new table to appear in any other position in the table set, you must insert the table.

To add a new table simply press the **Add** button on the toolbar. You'll notice a new table is created with the name "`Table #`" (where # represents the number of the table—if this is the 5th table in your table set, it will initially be named "`Table 5`"). Finish the definition of the added table by adding dictionary items and specifying any universe definitions or other tabulation parameters desired.

You can also add a table by right-clicking anywhere in a table and selecting **Add Table** from the popup menu. Alternatively, you can add a table from the **Edit** menu: select **Add Table**.

Insert Table

You can insert a table before another table by right-clicking anywhere in a table or on the table name in the table tree, then selecting **Insert Table** from the popup menu. Alternately, you can insert a table by clicking on the icon, or selecting **Insert Table** from the **Edit** menu.

Any table inserted into the existing table set will always be placed before the currently displayed table, which we will call "Table N". You'll notice a new tab is created with the name "Table #A" (where # is N-1). If you insert another table before this one, it will be created as "Table #AA". If you insert another table before "Table N", it will be created as "Table #B". In both cases # is N-1. Once created, finish the definition of the inserted table by adding dictionary items and specifying any universe definitions or other tabulation parameters desired.

Delete Table

The table that is currently on view in the table window is always the one affected when you choose to delete. So if the table you want to delete is not displayed in the table window, you must first make it visible. Either select it from the table tree on the left or by using the table paging arrows in the Tabulation toolbar.

You can then delete the table by right-clicking anywhere in the table or on the table name in the table tree, then selecting **Delete Table** from the popup menu. Alternately you can use the icon on the toolbar or select **Delete Table** from **Edit** menu. You will need to confirm that you actually want to delete the table.

Move Between Tables

To move through the tables press the or buttons on the toolbar.

To go to a specific table select the table title from the Table Tree on the left. If the Table Tree is not showing in the left panel, click on the **Tables** tab at the bottom of the panel.

Run a Tabulation Application

Once you have finished defining your table(s), you are ready to run them against your data files to produce the tables.

Press the Run Tabulation button or press **Ctrl+R**. If the application has been modified, you will need to save the changes in order to run.

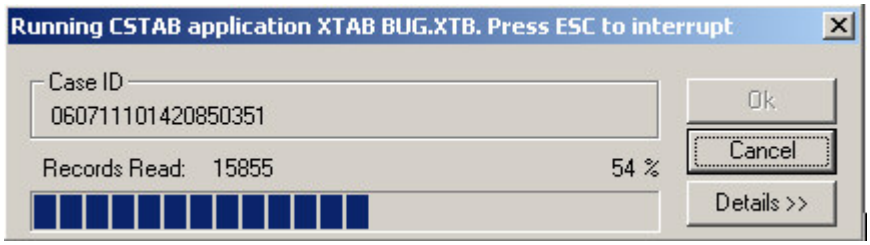
The run will request the following files as appropriate:
Input Data: The data file(s) being tabulated. There is NO required extension for CSPro data files. Multiple input data files can be selected using the browse button.

Area Names: Only used for applications with area processing. An Area Names File is used to associate the area codes

in the data file(s) with their descriptive text. The .ANM extension is required.

Output TBW: The Table Viewer file created by the run. The .TBW extension is required, but does not to be specified by the user.

During the run, a bar will be displayed showing progress in processing the data file (See below).

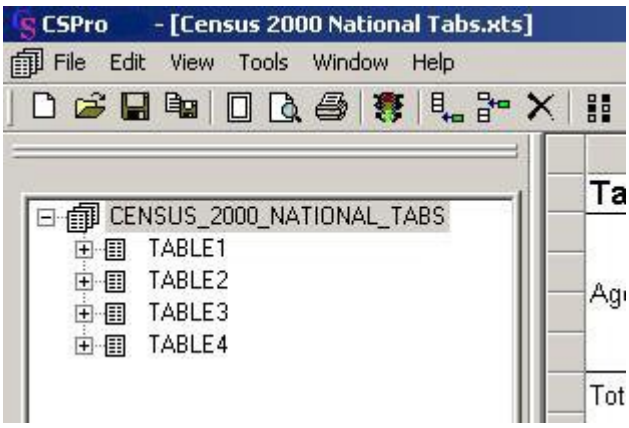


Note that you can also stop the processing by using the "Cancel" button.

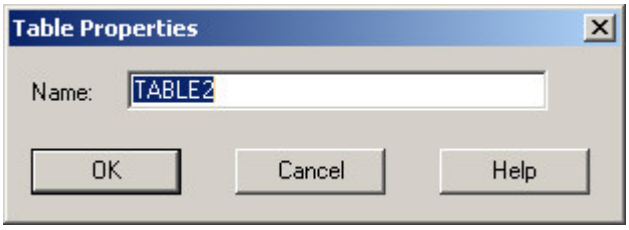
If there are no warnings or errors, the application will display the resulting tables. If your application has generated messages, these will be displayed via a listing in Text Viewer but the results will still be shown in your tables.

Renaming Tables and Table Applications

CSPro automatically assigns a name to each table in the application, such as TABLE1, TABLE2, etc. These names are displayed in the table tree (when "names in tree" is checked on the view menu), shown in certain error messages, and used in table post calculation.



You can change the name of a table using the table properties dialog. Right-click on the table in the "Tables" tree and choose "Properties". Edit the name of the table and click "OK". Note that the system will not allow you to have two tables with the same name.



You can also change the name and label of the table application by right-clicking on the table application in the "Tables" tree and choosing "Properties". The tables application is always located at the top of the tree. Edit the name and label of the application and click "OK."

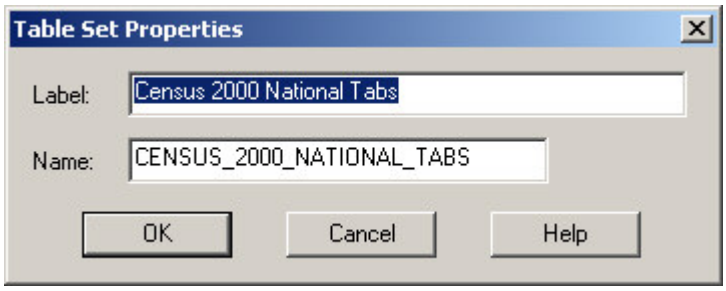


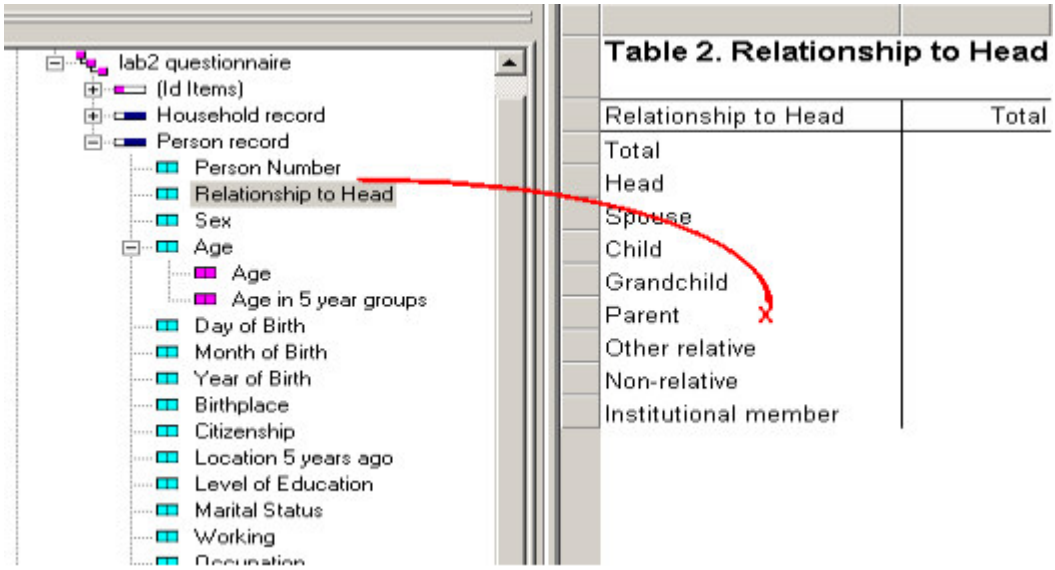
Table Set Properties
Label:
Census 2000 National Tabs
Name:
CENSUS_2000_NATIONAL_TABS
OK
Cancel
Help

How To ...

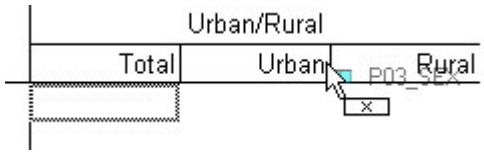
Add a Variable to a Tabulation

A table (or subtable) is created by selecting an item OR value set [if multiple value sets are available] from the data dictionary and then drag and dropping it into the table window on the right. Dropped in the upper right it becomes a column and in the lower left it becomes a row.

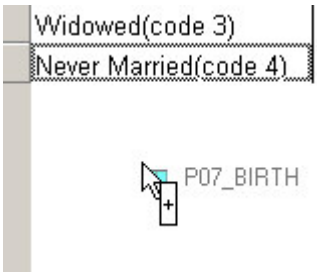
Here is an example of dropping a row.



In order to create subgroupings of categories already defined in the table, e.g., by sex, drag and drop the subgroup item (sex) anywhere on the existing value set. (See 'Sex' in column below)



In order to create an additional set of row/column categories, drag and drop the item or value set below the existing rows [for additional set of rows] or to the right of the existing columns [for additional set of columns]. (See Place of Birth and Martial Status below)



The resulting table would be as follows:

Table 1. Marital status, Place of birth by Urban/Rural and Sex

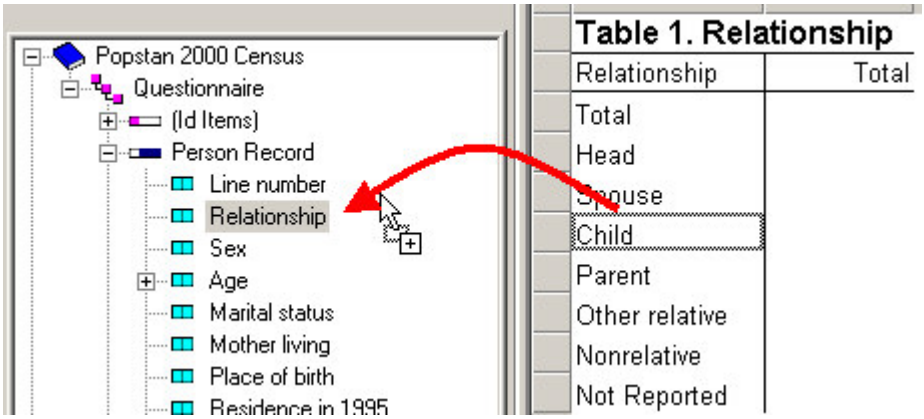
	Urban/Rural							
	Total			Urban			Ru	
	Sex			Sex			S	
	Total	Male	Female	Total	Male	Female	Total	
Marital status								
Total								
Married(code 1)								
Divorced(code 2)								
Widowed(code 3)								
Never Married(code 4)								
Place of birth								
Total								
Popstan								
Endar								
Victoria								
Africa								
Middle East								
Asia								
Europe								
Americas								

The title of the table is generated by combining the LABELS from the dropped items/values. The word "**and**" indicates that the second is dependent on the first. The word "**by**" separates the row labels from the column labels. A comma separates the subtable items in a row or column.

In example above: "Marital Status" and "Place of Birth" are two row items (separated by a ","). These two row items are separated from the column items using the word "by". The column item, "Urban/Rural", is subdivided by "Sex" (as indicated with "and").

Remove a Variable from a Tabulation

To remove a value set from a table, left-click on any of its value labels, then drag and drop it back in the tree panel. See below.



If an independent item is removed then the associated dependent item will also be removed.

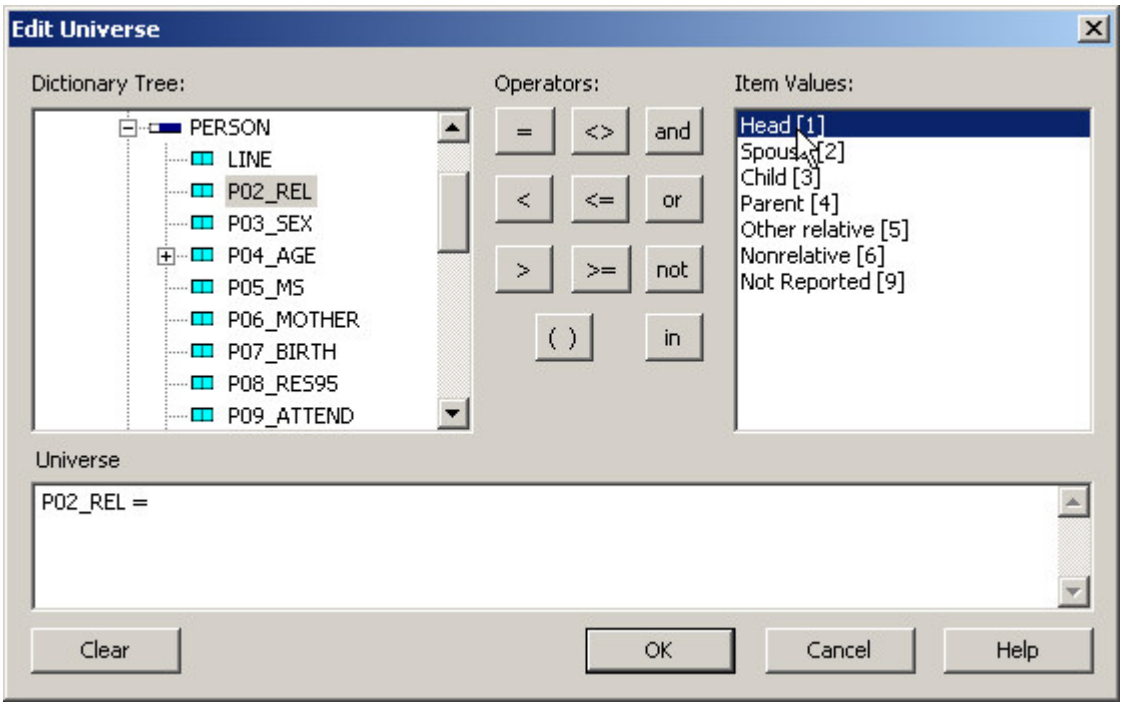
Define a Universe for a Table

The definition of a universe is available from the Tally Attributes for a Table dialog box. Universes are available for the entire table and/or individual subtables.

Experienced users, always remembering to use item names, can type the universe conditional directly in the box provided by the **Tally Attributes** menu.

Use the **Apply All** button to the right of the box if the universe is for the entire table and you want to propagate it to all tables currently defined. It will replace any existing Universe in other tables.

Less experienced users can access the universe "wizard" from the "Edit" button to the right of the universe box. This brings up the following:



You can type a universe condition statement, as CSPro logic, in the Universe box. Or, you can double click on an item name, press a relation button, and double click on a value.

- Double click on the item in the left-hand box that you want to use in the universe condition. The item name will appear appended to the text in the universe condition below.
- Click a relationship button (=, <>, >, >=, <, <=). The relation will appear appended to the text in the universe condition below.
- Double click on a value in the right-hand box. The value, not the text label, will appear appended to the text in the universe condition below.
- You may enter several conditions using the **and / or**. You can also add parentheses to modify the order of evaluation of the conditions.
- To delete the universe simply erase the contents of the universe condition.
- Press OK when you have completed the universe condition.
- You can apply a universe to all the tables, by pressing the **Apply All** button to the right of the Universe box in the **Tally Attributes** dialog box

Examples:

- To restrict your table to females of reproductive age, you might state:

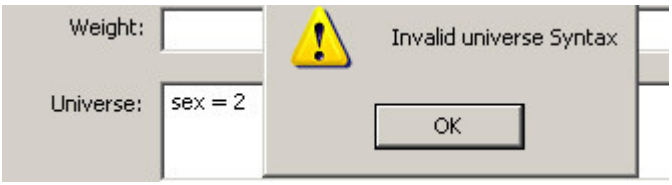
```
P03_SEX = 2 and P04_AGE in 12:49
```

- To restrict your table to heads of households who are economically active, you might state:

```
P02_RELATION = 1 and ECON_ACTIVE = 1
```

Notes:
If there is a universe condition for the entire table and another for a subtable, the two conditionals will be combined by "and" to determine tallies for the subtable.

If there is some type of error either in the typed or generated conditional, the system will display an "Invalid Universe Syntax" message and the problem must be corrected before the universe can be accepted. (In the example below, "Sex" is not the CSPro name of a variable.)



See also: Restrict a Universe, Tally Attributes for a Table

Add Weights to a Table

The declaration of weights is available from the Tally Attributes for a Table dialog box. Weights are available for the entire table and/or individual subtables.

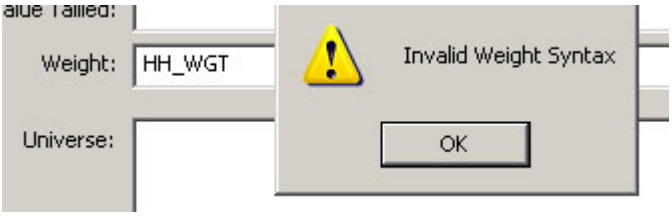
Weights can be a data item name, a numeric constant, or any arithmetic expression.

To add a weight to a table or subtable, simply type the expression the item name, numeric constant or expression directly into the box provided for weight.

Use the **Apply All** button to the right of the box if the weight is for the entire table and you want to propagate it to all tables currently defined. It will replace any existing weight in other tables.

Notes:
If the value of the weight data item or expression is not numeric when a tally is to be made, the tally is NOT done. This is equivalent to a tally of zero.

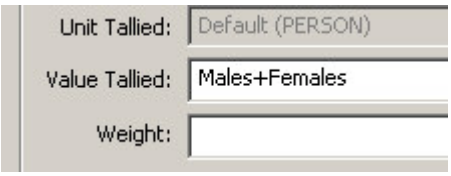
If the weight expression is not valid it must be corrected before the Tally Attributes dialog box can be closed.



See also: Tabulate Values and/or Weights, Tally Attributes for a Table

Tabulate Values Instead of Frequencies

The declaration of values to be tallied is available from the Tally Attributes for a Table dialog box. Tally values are available for the entire table and/or individual subtables.



A value to be tallied can be an individual data item, a numeric constant, or any valid expression that can be resolved into a single value.

Note:
If the value to be tallied is not numeric when a tally is to be made then the tally is NOT done. This is equivalent to a tally of zero.

If the tally value expression is not valid it must be corrected before the Tally Attributes dialog box can be closed. A message is issued if the expression is invalid.

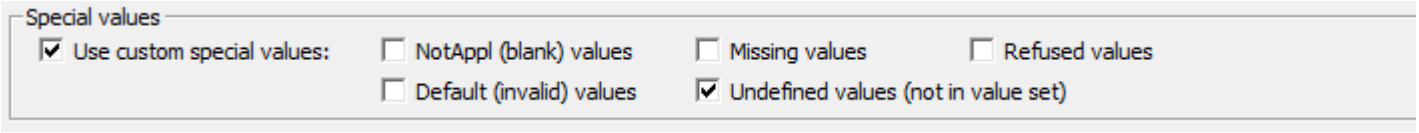
See also: Tabulate Values and/or Weights, Tally Attributes for a Table

Include/Exclude Special Values in a Variable

The addition of special values to be tallied is available from the Tally Attributes for a Table dialog box. It allows you to add rows and columns to your table to tally special values that are not contained in the value sets for your variables or to remove the rows and columns containing special values that are included in the value sets for your variables.

Use custom special values always applies to all variables in a table. You cannot change the special values for a single variable or single subtable.

In the Tally Attributes dialog box users have the following special value options:



To activate the Special Values portion of then menu, check the **Use custom special values** box (as shown).

After that select the types of special values you want to add to/remove from the value sets of each item in the table by checking the appropriate box.

- **NotAppl** values counts blanks occurring in the data item.
- **Missing** values counts special codes defined as "missing" in the data dictionary.
- **Refused** values counts special codes defined as "refused" in the data dictionary.
- **Default** values counts the invalid values (shown as asterisks "*" in the data).
- **Undefined** values counts any values for the item which is not otherwise counted.

Once **Use custom special values** is checked, this setting overrides any special values in the value sets for items in the table. If **Use custom special values** is checked, any of the special values that are checked in the tally attributes dialog will be displayed for every item in the table regardless of whether or not the value set for that item contains the special value. In addition, only the special values that are checked in the dialog box will be displayed in the table even if they appear in the value set for the an item in the table. For example, if your value set contains **Notappl** and you check **Use custom special values** but leave **Notappl** unchecked then the columns/rows corresponding to the **Notappl** value in your value set will be removed from the table. If your value set contains no special values, and you check **Use custom special values** and **Notappl**, then a row or column for **Notappl** will be added for each variable in your table.

Note: Undefined values count includes blanks if **Notappl** values is not checked. If it is checked then those counts are taken out of the "Undefined values" count and tallied separately. Similar statements apply to other special values.

Hide or Change the Position of the Total

By default, a total count is automatically associated with each value set placed in a table. The usual placement for rows is the top of the value labels; for columns it is to the left of the value labels. To modify this use the Tally Attributes for a Variable dialog.

To remove the total row or column, select the total by clicking on it in the list of selected calculations on the right hand side of the dialog, and then click on the **Remove** button to delete the total.

To move the total after the counts, select the total by clicking on it in the list of selected calculations on the right hand side of the dialog, and then click on the **Down** button to move it until it is after the counts.

The default options are set in the Preferences and Default Formats dialog.

See also: Hide or Show a Row or Column

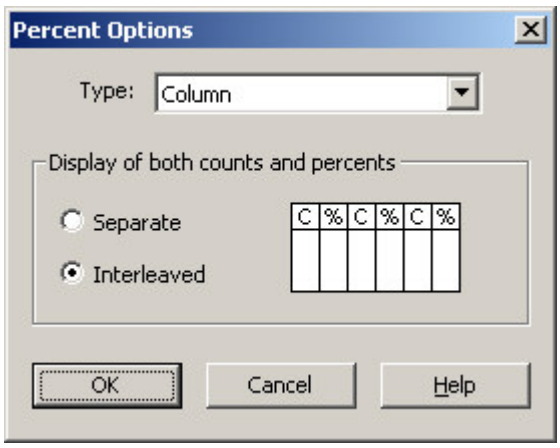
Add Percents to a Table

There are optional percentages available for each value set in a table. To create percentages, use the Tally Attributes for a Variable dialog.

To bring up the Tally Attributes for a Variable Dialog, right-click on the variable in the table and choose Tally Attributes (<variable>).

To add percents to the variable, select "Percents" in the list of available calculations on the left-hand side of the dialog and click the "Add" button.

There are two options for "Percents" which can be set by clicking the "Options" button while the "Percents" are selected:



Type

- **Total** – Percents are based on total cell for the table (% cell = (associated cell value * 100) / total of all the cells in the table).
- **Row** – Percents are based on total for the row (%cell = (associated cell value * 100) / total of all the cells in the row).
- **Column** – Percents are based on total for the column (%cell = (associated cell value * 100) / total of all the cells in column).

Interleaved/Separate
By default, percents and counts are interleaved; that is, for each value in the value set, the frequency for that value is listed in the column/row directly preceding or following the percent for that value. Percents may also be separate, in which case all the frequencies are grouped together and all the percents are grouped together.

Sex					
Total	Percent	Male	Percent	Female	Percent

Interleaved Percents

Sex					
Total	Male	Female	% Total	% Male	% Female

Separate Percents

For percents to be interleaved, the percents must directly follow or directly precede the counts in the list of selected calculations in the tally attributes dialog.

You can change the position of the percents relative to the other selected calculations (including the counts) by selecting the percents in the list on the right-hand side of the dialog and clicking the "Up" or "Down" buttons.

To show percents only (percents without counts), add the percents as described above and set the options to "Separate". Then select the counts in the list of selected calculations on the right-hand side of the dialog and click the "Remove" button to delete the counts. Finally, do the same to remove the "Total".

To show percents only with a number row, add the percents and remove the counts as above, but do not remove the "Total".

The Defaults are set in the Preferences and Default Formats.

By default, data in rows and columns containing percents will be displayed with one decimal place (e.g., 12.3). To change the number of decimal places see Change the Number of Decimal Places Displayed.

You can also show the percentage of the counts for a subset of the values in a value set (e.g., percent of people with age between 15 and 49) using proportions. See the section on proportions under Tally Attributes for Variable for more information.

Add Summary Statistics to a Table

There are optional summary statistics available for each value set in a table. To add summary statistics to a table use the Tally Attributes for a Variable dialog.

The choices for statistical measures are:

- **Mean** – average value of observations.
- **Median** - middle value (half of the observations are above this value and half below).
- **Mode** – for categorical (discrete) variables this is the most frequently observed value. For grouped variables, this is the lower limit of the modal class or the range in the value set in which the most observations lie.
- **Std Deviation** – Standard Deviation, a measure of how clustered the observations are around the mean (square root of the variance).
- **Variance** – another measure of dispersion.
- **N-tiles** - values that divide the values into N groups each of which contain 1/N of the total observations (N = 2 is equivalent to median).
- **Minimum** – smallest value found in all observations.
- **Maximum** – largest value found in all observations.
- **Proportion** – show counts for certain values in the value set as a fraction or percentage of the total for the variable.

It is assumed that users know the meaning and relevance of any statistics that are selected. They are only meaningful for data that represents a true numeric value, e.g., age, income, hectares, etc.).

Many of the statistics above have additional parameters that can set by selecting them and clicking on the "Options" button. These options are documented in Tally Attributes for a Variable.

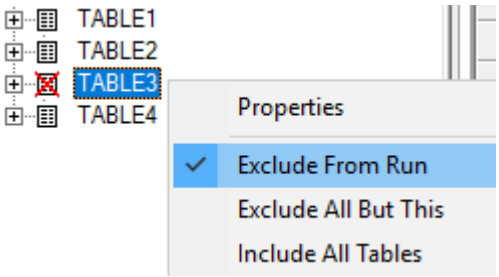
Note that in the case of grouped or continuous variables, the **median**, **n-tiles** and **mode** depend highly on the value set used. This is because these statistics are calculated using the frequency distribution for the value set rather than on the raw data itself. The **mode** will be the category in the value set that contains the most observations. The **median** is calculated by interpolation using the limits of the category in the value set within which the cumulative frequency reaches 50%. This means that the **mean**, **mode** and **n-tiles** will be more accurate for value sets with more smaller categories rather than fewer large categories. For best results, use value sets with a large number of uniform categories when calculating **median**, **mode** and **n-tiles**. For example, using single year of age or 5 year age groups rather than 15 or 20 year age groups will result in a more accurate **median** age. Using a single grouping for age (for example one category for 0-115 years) will result in a highly inaccurate result. Note that for **median** and **n-tiles**, it is possible to set groupings for the median/n-tile calculation which are different from the categories in the value set in order to get a more accurate result. See Tally Attributes for a Variable for details.

See Also: Implications of Data Dictionary Value Sets

Include/Exclude Tables from Run

By default, all the tables you define in a tabulation application will be produced when you run the application. However, there are times when you may not want all the tables to be produced. For example, if you already have fully tested many tables and you add another table that you wish to test, you may want to run only that table in order to save time. In this case you would exclude all the other tables from the run.

To exclude a table from the run, first make sure the table is selected and appears on the screen. From the **Edit** menu, select **Exclude Table from Run**, or right-click on the table name on the table tree on the left and select **Exclude from Run**. The icon next to the table in the tree on the left will show a red X when the table is excluded.



If a table is already excluded, follow the same steps to change the setting back to included in the run. The red X will disappear.

To exclude all the tables but the one currently selected, select the **Exclude All But This** option. This is a useful during testing when you want to run only one table at a time. To undo any exclusions, select the **Include All Tables** option.

Debug Table Totals

At times, the totals in a table may differ from totals calculated using other methods. There are some easy ways to figure out the cause of such discrepancies.

Some values that exist in your data file may not be in the value set and thus these values will not be counted in the totals. For example, a value set might include the numbers 1 through 6. The data file might have a value of 7 or 8 for the variable or in some cases might contain blank (notappl) values for the variable. To see if this is the case, try including custom special values in your table. Make sure to check the boxes for the special values **notappl**, **undefined**, and **default**. Run the table again and check to see if any of the rows/columns for special values have counts in them.

Table 1. Place of birth by Sex

Place of birth	Sex					
	Total	Male	Female	Default	NotApplicable	Undefined
Total	24,271	11,787	12,484	-	-	-
Popstan	23,350	11,325	12,025	-	-	-
Endar	547	273	274	-	-	-
Victoria	122	60	62	-	-	-
Middle East	-	-	-	-	-	-
Asia	56	28	28	-	-	-
Europe	78	39	39	-	-	-
Default	-	-	-	-	-	-
NotApplicable	-	-	-	-	-	-
Undefined	118	62	56	-	-	-

If you have counts for the special value **undefined** then there are values in your data file that are not in the value set that you are tabulating. You may either edit the data file to eliminate such values or change your value set to include them. To find the actual values that are not in the value set, use the Tabulate Frequencies tool with the Each value found option.

If you have counts for the special value **notappl** then there are blanks in your data file for the variable in question. You will either need to impute the blanks or add an entry for notappl to your value set.

If you have counts for the special value **default** then there are values in your data file that cannot be read by CSPro. This can be because the properties of the variable in the dictionary do not match the data in the file. For example if the data type in the dictionary is numeric but the value in the data file contains non-numeric characters or if there is a decimal character in the data file but none specified for the item in the dictionary. It can also be because when the data file was written the value of the variable overflowed the length specified for the variable in the dictionary.

You should examine the data file and the dictionary file for discrepancies and also check any programs that have written to the data file to see if they could be writing **default** values.

There may be some values in your value set that are repeated and are being counted in more than one row or column of the table. For example, in the following value set, age 5 is in both the 0-5 and the 5-10 ranges and therefore each person with and age of 5 will be counted twice: once in the 0-5 category and once in the 5-10 category. The total will be the correct count of the number of persons, however, the sum of the age categories will not match the total.

Value Set Labe	Value Set Name	Value Label	From	To	Special
Age	P04_AGE_VS1				
		0 to 5 years	0	5	
		5 to 10 years	5	10	
		10 to 15 years	10	15	

Check your value set to make sure that no single value is repeated and that no ranges overlap. Of course, in some cases, you may want to have overlapping ranges in your value set, for example if you want to show subtotals.

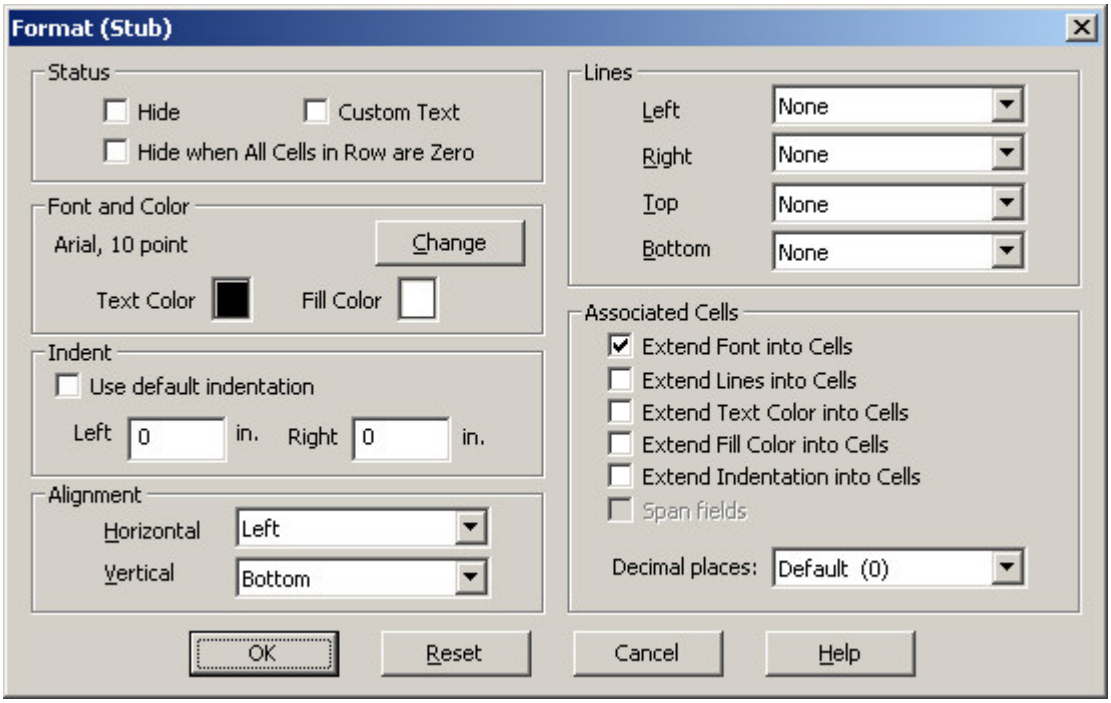
See Also: Implications of Data Dictionary Value Sets, Special Values

Formatting Tables

Formats for a Part of a Table

The Format (Table Element) Dialog Box allows you to change the appearance of individual elements of a table such as stubs, column heads, data cells, etc.

To bring up the Format (Table Element) Dialog Box, right-click on the element in the table that you wish to modify and choose Format (<element name>) where <element name> is the type of table element that you clicked on (title, stub, column head, etc…).



The Format (Table Element) Dialog Box has the following settings:

Hide: When checked, the selected element will not be displayed in the table. If a column head or row stub is selected, then the associated column or row will also be hidden. Note that stub heads, titles, and data cells may not be hidden.

Hide when All Cells in Row are Zero: Available only for stubs. When checked, if all cells in the row contain the value zero, then the row will be hidden.

Custom Text: When checked, the default text generated by CSPro for the selected element will not be displayed. You may add your own text for the selected element by double-clicking on the element and typing in the new text. Note that if you double-click on a table element and add your own text without first checking the custom text box, CSPro will automatically check it for you. You can uncheck this box to replace your new text with the default CSPro-generated text.

Font: Displays the current font and font size for the selected element. You may switch to a different font and/or font size by clicking on the change button.

Text Color: Shows the current color used to display the text of the selected element. To change the text color, click on the square displaying the current color.

Fill Color: Shows the current color used to display the cell background of the selected element. To change the fill color, click on the square displaying the current color.

Indent: Allows you to set the indentation of the selected element from the right and left edges of the cell. To set the

indentation for the selected element, first uncheck "use default indentation" and then modify the numbers for "Left" and "Right" indentation appropriately. Note that columns in the table will automatically be made wider to accommodate indentation.

Alignment: Allows you to set the horizontal and vertical justification of the selected element within its cell. Using the horizontal alignment control, elements can be right-justified, left-justified or centered. Using the vertical alignment control, elements can be top-justified, bottom-justified or centered (middle).

Lines: Allows you to set the borders on the top, left, bottom and right sides of the selected element. Each side of the element may have no border, a thick border or a thin border. Note that lines may not be modified for an individual data cell. Instead you should set the lines for a whole row or column by setting the lines for the associated stub or column head and checking the "extend lines into cells" checkbox (see below).

Extend Font into Cells: When this is checked for a column head or stub, the font and font size for the selected element are applied to all cells in the associated column or row. For a stub, all cells in the row of the stub will share the font of the stub. For a column head, all cells in the column underneath the head will share the font of the column head.

Extend Lines into Cells: When this is checked for a column head, spanner, stub or caption, the lines (borders) for the selected element are applied to all cells in the associated column(s) or row. For a stub or caption, all cells in the row of the stub or caption will share the top and bottom line settings of the caption. For a column head, all cells in the column will share the left and right line settings of the column head. For a spanner, the cells in the rightmost column underneath the spanner will share the right line setting of the spanner and the cells in the leftmost column underneath the spanner will share the left line setting of the spanner.

Table 1. Age in 10 year groups by Marital status

Age in 10 year groups	Marital status					
	Total	Married	Divorced	Separated	Widowed	Never Married
Total						
0 to 9 years						
10 to 19 years						
20 to 29 years						
30 to 39 years						
40 to 49 years						
50 to 59 years						
60 to 69 years						
70 to 79 years						
80 to 89 years						
90 years and over						

(Thick lines extended into cells from a stub ('30 to 39 years') and a column head ('Separated'))

Note that the lines between rows and columns are shared between multiple elements. For example, the vertical line between two columns is both the left edge of one column and the right edge of the other column. It may also be the edge of one or more spanners. At times, you may need to change the line setting in more than one of these elements in order to change a line. For example, in order to make the vertical line between two columns disappear, you may have to set the left line of one column to 'none' and the right line of the other column to 'none'. You may also have to set the left or right line of one or more spanners to 'none'.

Extend Text Color into Cells: When this is checked for a column head or stub, the text color for the selected element is applied to all cells in the associated column or row. For a stub, all cells in the row of the stub will share the text color of the stub. For a column head, all cells in the column underneath the head will share the text color of the column head.

Extend Fill Color into Cells: When this is checked for a column head or stub, the background color for the selected element is applied to all cells in the associated column or row. For a stub, all cells in the row of the stub will share the fill color of the stub. For a column head, all cells in the column underneath the head will share the fill color of the column head.

Extend Indentation into Cells: When this is checked for a column head or stub, the indentation setting of the selected element is applied to all cells in the associated column or row. For a stub, all cells in the row of the stub will share the indentation of the stub. For a column head, all cells in the column underneath the head will share the indentation of the column head.

Span Cells: This setting applies only to captions. When this is checked for a caption, the caption spans the entire row. By default, captions are placed in the first column of the table along with the stubs, and the cells containing them are the same size as the cells containing the stubs. This means that there are empty cells in all other columns of the caption row. When "span cells" is checked, the empty cells in the caption row are removed and the caption is placed in one

large cell that spans the entire row. This allows you, for example, to center or right-justify the caption in the entire row (by setting the alignment).

Decimal Places: Sets the number of decimal places used to display numeric values for the selected element. If applied to a data cell, this sets the number of decimal places used for that individual cell. If applied to a column head or stub it sets the number of decimal places for the entire row or column. This may only be applied to data cells, column heads and stubs.

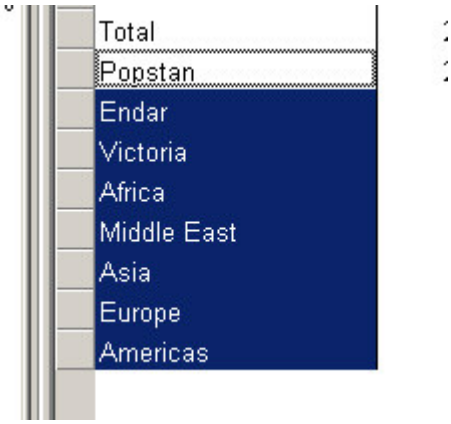
Formatting Row, Column, or Cell Data

Rows, columns and data cells are formatted in the same way as other parts of a table. Many formats, such as fonts, colors, indentation and lines, can be applied either to individual data cells or to entire rows or columns. To format an entire row, right-click on the corresponding stub and choose "Format (Stub)". To format an entire column, select the corresponding column head and choose "Format (Column Head)". Once you have changed the formats, make sure that "extend into cells" is checked in the Format (Table Element) dialog for the formats that you have modified. For example, if you wish to change the font for an entire row, make sure that "Extend font into cells" is checked for the stub, otherwise the font change will only apply to the stub itself and not to the rest of the row. See Formats for a Part of a Table for details on the Format (Table Element) dialog.

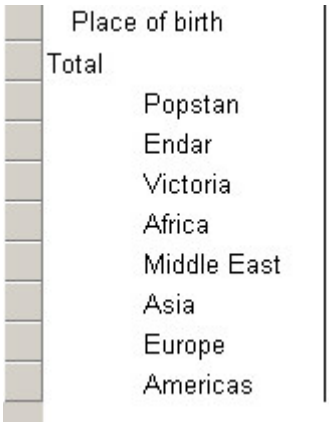
You can select multiple rows (stubs) or columns (column heads) to format. This is very useful to modify decimal places, indentation or specific attributes of specific groups of rows or columns; for example, to make 'total' rows bold.

Multiple selection is done in the usual Windows manner by left-clicking the mouse while using the Shift or Ctrl key. Using the Ctrl key adds each cell in the table that is clicked on to the selection. Using the shift key adds all the cells in between the two cells clicked on to the selection. Although you may select different types of cells at the same time (data cells, stubs, column heads), you can only format selected groups when all selected items are of the same type (for example, all selected items are stubs).

In this example multiple stubs have been selected but 'Total' is not.



Use the right-click to access the Format (Stub) menu. Change the indentation to .5 in. then select OK. The result is below.

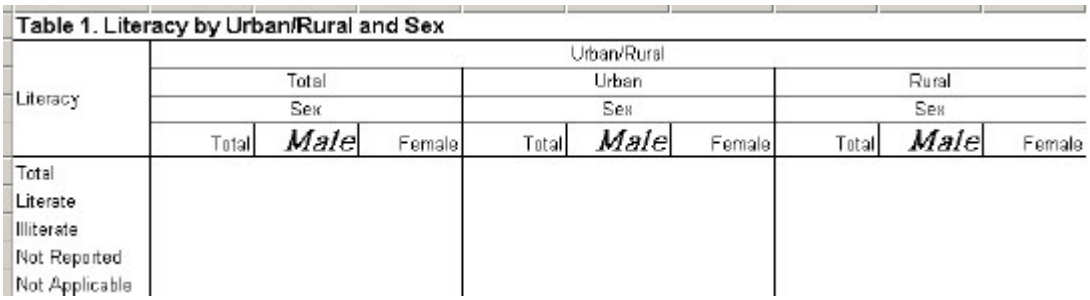


Similar actions can be performed on columns by selecting multiple column heads. You can also select multiple captions

or spanners.

Although the formatting option is available for each data cell, it should only be used in the rarest of cases.

When working with sub-groupings (one variable dropped on top of another), any formatting applied to one row or column in a sub-grouping will automatically be applied to the corresponding row or column in all of the other sub groupings. In the example below, changing the font for the column "Male" under the spanner "Total" also changes the font for the "Male" column under the "Urban" and "Rural" spanners.



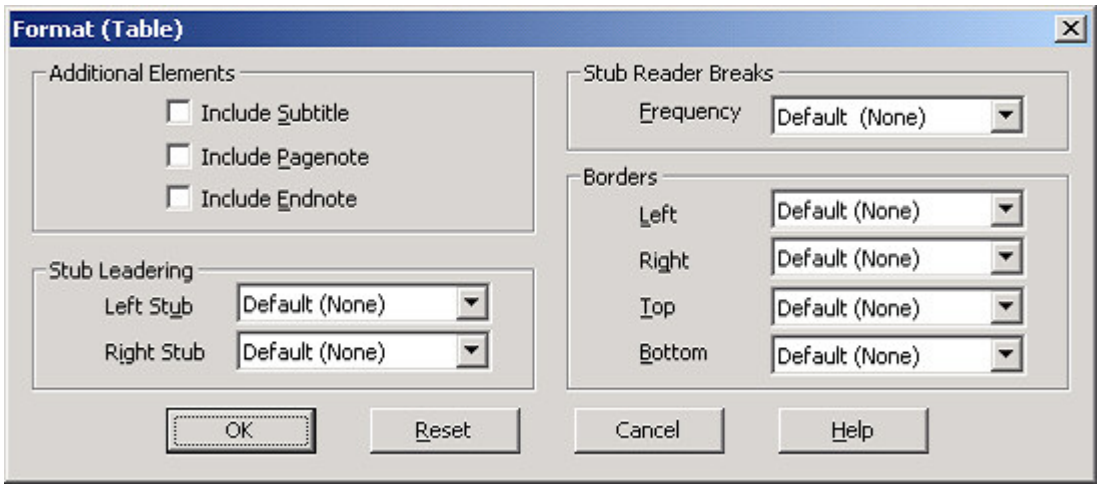
You cannot change the format for one row or column in only one sub-grouping but not the others. You can, however, achieve the same result using multiple subtables with universes.

See also: Change the Way Numbers are Displayed

Formats for a Table

The Format (Table) Dialog Box allows you to control the overall appearance of a table.

To bring up the Format (Table) Dialog Box, right-click anywhere on the table and choose "Format (Table)", or choose "Format (Table)" from the Edit menu. Any changes made will apply only to the current table.



The Format (Table) Dialog Box has the following settings:

Include Subtitle: When checked, an additional row below the title row is added to the table. You may enter a subtitle to the table by double-clicking on the new row and typing in the text for the subtitle.

Include Pagenote: When checked, an additional row is added at the end of the table. You may enter a pagenote for your table by double-clicking on the new row and typing in the text for the pagenote. Pagenotes are displayed at the bottom of each page of the table.

Include Endnote: When checked, an additional row is added at the end of the table. You may enter an endnote for your table by double-clicking on the new row and typing in the text for the endnote. Endnotes are displayed only once per table, at the bottom of the last page of the table.

Stub Reader Breaks: Sets the frequency of reader breaks in the rows of the table. Reader breaks are blank rows used

to break up the table. By default there are no reader breaks. The frequency of reader breaks may be set anywhere from 1 (every other row is blank) to 10 (every 10th row is blank). Note that you can hide individual reader breaks by right-clicking on them and selecting Format (Stub). This allows for finer control over placement of reader breaks.

Table 1. Age in 10 year groups by Sex			
Age in 10 year groups	Sex		
	Total	Male	Female
Total			
0 to 9 years			
10 to 19 years			
20 to 29 years			
30 to 39 years			
40 to 49 years			
50 to 59 years			
60 to 69 years			
70 to 79 years			
80 to 89 years			
90 years and over			

(Stub reader break frequency set to 3.)

Stub Leadering: Allows you to set a pattern such as '….' or '_ _ _ _' to appear in each row of the table between the end of the stub text and the start of the next column. You may choose different patterns for the stubs on the left and right sides of the table. Note that stub leadering is only visible in print preview and on the printed page.

Table 1. Age in 10 year groups by Sex

Age in 10 year groups	Sex			Age in 10 year groups
	Total	Male	Female	
Total.....	24,273	11,789	12,484	---------- Total
0 to 9 years.....	7,467	3,734	3,733	---------- 0 to 9 years
10 to 19 years.....	6,217	3,000	3,217	-------- 10 to 19 years
20 to 29 years.....	3,929	1,850	2,079	-------- 20 to 29 years
30 to 39 years.....	2,809	1,362	1,447	-------- 30 to 39 years
40 to 49 years.....	1,618	797	821	-------- 40 to 49 years
50 to 59 years.....	1,085	528	557	-------- 50 to 59 years
60 to 69 years.....	726	336	390	-------- 60 to 69 years
70 to 79 years.....	328	141	187	-------- 70 to 79 years
80 to 89 years.....	80	39	41	-------- 80 to 89 years
90 years and over.....	14	2	12	----- 90 years and over

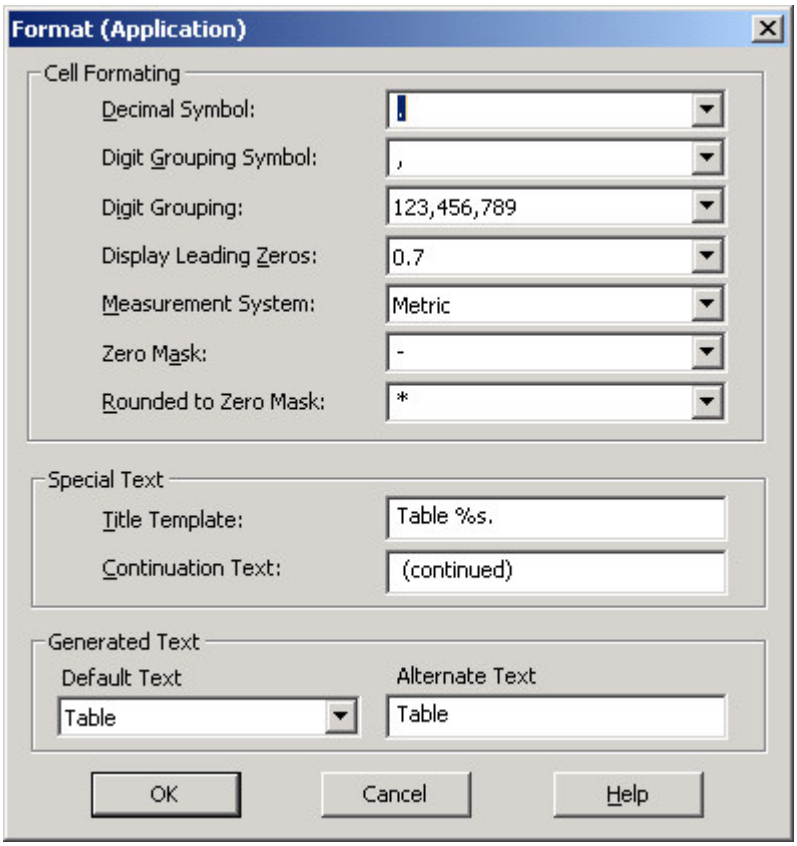
(Stub leadering on both left and right)

Borders: Allows you to set the borders on the top, left, bottom and right sides of the table. Each side may have no border, a thick border, or a thin border.

Formats for an Application

To bring up the Format (Application) Dialog Box, right-click on a table and select Format (Application), or choose "Format (Application)" from the Edit menu.

The Format (Application) Dialog Box controls the appearance of numbers and content of text labels in all tables in the file. It is particularly useful for customizing tables in languages other than English. Note that unlike Format (Table) and Format Print (Table), settings made in this dialog box apply to *all* tables in the file.



The following settings are available:

Decimal Symbol: Sets the symbol used to separate the whole and fractional parts of numbers in data cells. By default this is set to period; however, it may be set to comma or any other symbol. For example, changing the symbol from period to comma would change the display of a number from 3.14 to 3,14

Digit Grouping Symbol: Sets the symbol used to separate groups of digits in large numbers in data cells. By default this is set to comma so that numbers are displayed as 123,456,789 but it can be changed; for example, to a period so that numbers are displayed as 123.456.789

Digit Grouping: Controls how digits are grouped together to be separated by the digit grouping symbol. There are three options:

- 123,456,789 (default) – place the digit grouping symbol between each group of three digits.
- 123456789 – do not use the digit grouping symbol.
- 12,34,56,789 – place the digit grouping symbol between each group of two digits with one group of three digits at the right.

Display Leading Zeros: Controls whether or not zero should be displayed before the decimal point of a decimal number between zero and one. Leading zeros are displayed by default.

Measurement System: Allows you to switch between US (inches) and metric (centimeters) measurements for page margins. The measurement is used when specifying margins in the Format Print (Table) dialog as well as indentation in the Format (Table Element) dialog. The measurement system is set to US by default.

Zero Mask: Allows you to change the text that is displayed for data cells that contain a value of exactly zero. By default the zero mask is "-", so any data cell containing a value of zero is displayed as "-" rather than "0". If you want zero to be displayed, remove the zero mask text.

Rounded to Zero Mask: Allows you to change the text that is displayed for data cells that contain a value which is rounded off to zero because it cannot be displayed using the number of decimal places for that data cell. For example, if the value in a data cell is calculated to be 0.002 but the number of decimal places for that cell is set to 2, the rounded to zero mask will be displayed in that cell. Having the zero mask and rounded to zero masks be different makes it clear which values are exactly zero and which ones are rounded off. By default the rounded to zero mask is "*". If you want zero to be displayed, remove the rounded to zero mask text. Note that you can change the number of decimal places for

a row, column or data cell by using the Format (Table Element) Dialog Box.

Title Template: Sets the text used to automatically generate the first portion of default table titles. When you create a new table, CSPro automatically gives the new table a title. By default, this title begins with "Table %s." where the table number replaces %s. You can change this by entering your own text in this edit box. If you put "%s" in your new title template, it will automatically be replaced with the table number when the table title is generated. As an example, if you put "Tableau %1" in the edit box, your table titles will begin with "Tableau 1", "Tableau 2", …

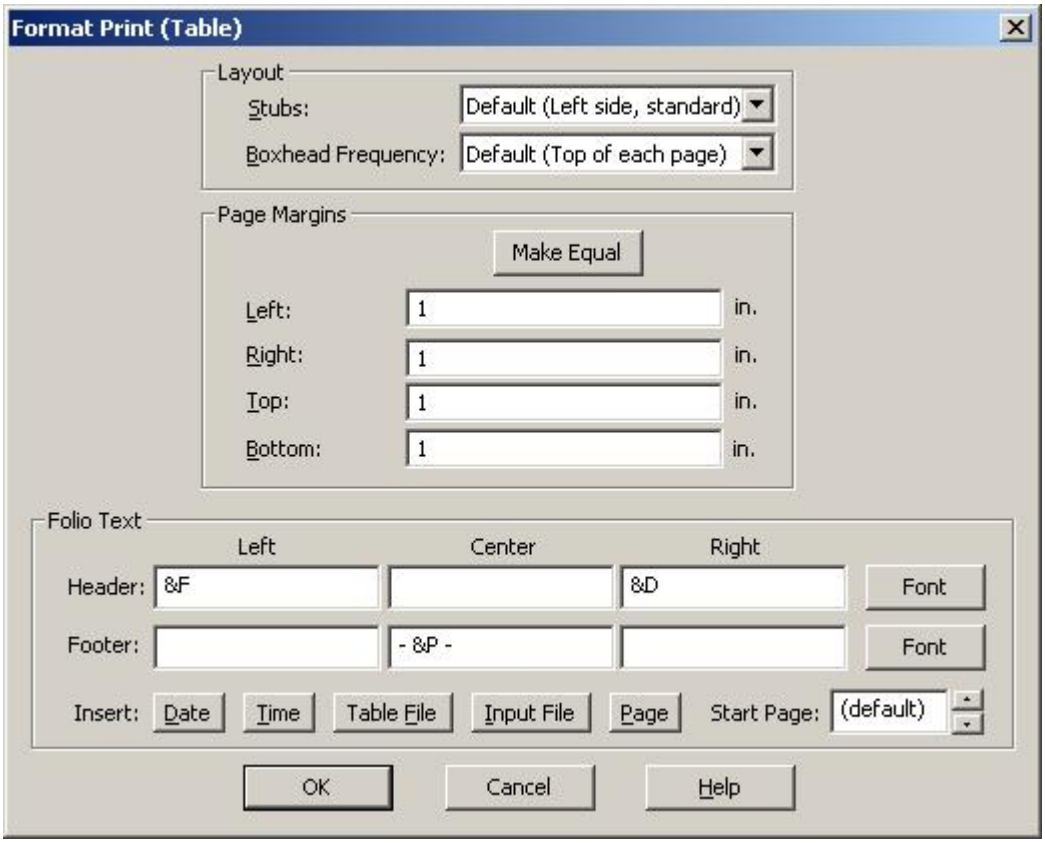
Continuation Text: Sets the text added to the end of the table title on the printed page and in print preview to indicate that the current page is not the first page of the table. By default this text is "(continued)", so that on all but the first printed page of a table, the title would be, for example, "Table 1: Age by Sex (continued)".

Generated Text: Sets other text (besides title template and continuation text) used by CSPro to generate default text in table titles, column heads, and stubs. To change a particular text string, first select the default string under "Default Text" and then enter the new string under "Alternate Text". For example, to replace the string "Percent" used in stubs and column heads with "%", first select "Percent" from "Default Text" and then type "%" into "Alternate Text". Now any rows or columns with percents in them will have "%" in the stub or column head.

Formats for Printing

To bring up the Format Print (Table) Dialog Box, either choose Page Setup from the File menu or right-click anywhere on the table and choose Format Print (Table) or choose "Format Print (Table)" from the Edit menu.

The Format Print (Table) Dialog Box allows you to change the appearance of the selected table on the printed page. Note that changes made in this dialog will only be visible when the file is printed or viewed in print preview. Changes made using Format Print (Table) only apply to the current table. To change the printed appearance of all tables in the file, use the Preferences Dialog Box available from the Edit menu.

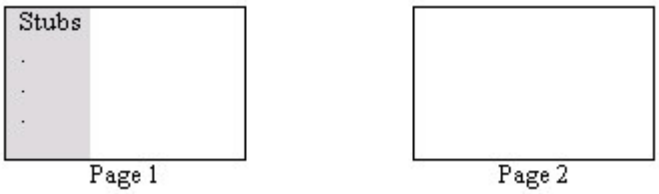


The Format Print (Table) Dialog Box has the following settings:

Stubs: Controls where stubs will be placed on the printed page. There are four options:

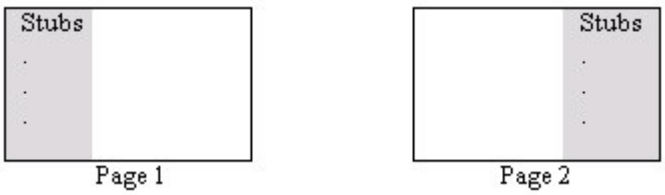
- Left side, standard (default): place stubs along the left side of every page. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left sides of both pages.

- Left side, facing pages: place stubs along the left side of the page but only for the first page of facing pages. If the table is wide enough so that some of the columns go onto a second page, no stubs will be placed on the second page. If the table columns all fit on one page then this is the same as Left side, standard.



- Left and right sides, standard: place stubs on both the left and right sides of the page for every page. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left and right sides of both pages.

- Left and right sides, facing pages: place stubs along the left side the first page of facing pages and on the right side of the second page of facing pages. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left side on the first page and the right side of the second page. If the table columns all fit on one page then this is the same as Left side, standard.



Boxhead Frequency: Controls where the boxheads will be placed on the printed pages. The box head is the area at the top of the table that contains the stub heads, column heads and spanners. You can set the boxhead frequency to any of the following:

- Top of each page (default): Places a boxhead at the top of every page of the table.
- No boxheads: No boxheads will be placed at all.
- Top of table only: Only place the boxhead on the first page of the table and not any of the other pages of the table.

Page Margins: Allows you to set the top, bottom, left and right margins of the printed pages for the selected table. By default, margins are measured in inches although you can change it to centimeters by choosing "metric measurement" in the Format (Application) Dialog Box. Use the "Make Equal" button to set all four margins to the currently selected number.

Folio Text: Allows you to set header and footer text for each page of the selected table. The header and footer each contain left, center and right portions that can be set separately by entering the desired text in the corresponding edit box.

You can modify the fonts used for the header and footer by clicking the Font buttons. The top Font button applies to the header and the bottom button applies to the footer.

You can place the current date, current time, page number and file name(s) in any of the header or footer sections by clicking on the appropriate buttons below the header and footer edit boxes or by adding the following text into the appropriate header/footer edit box:

Text in header/footer edit box	Replaced with on printed page
&D	Current date
&T	Current time
&P	Page number
&F	Name of table file
&I	Name of data input file(s)

By default, CSPro places the filename in the left header, the date in the right header and the page number in the center footer.

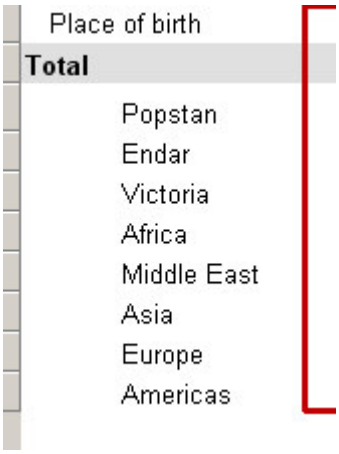
Start Page: Sets the page number to use on the first page for page numbering. If this is set to 1, then the first page of the table will be page 1; the second page will be page 2, etc… If it is set to default then the first page of the selected table will be one greater than the last page of the previous table in the file so that the numbers continue from one table to the next with no gaps. For the first table in the file, if the start page is set to default, then the page numbering for that table will start at page 1.

Views of Tables

Beside Print Preview, there are two views of every table. However, unless the table has hidden rows or columns, or multiple subtables, the two views are basically the same. One view, called the "Design view" shows the hidden rows or columns, and the other view, called the "Display" or "Data" view, does not. Users can switch between the two views by using Ctrl+D or checking/unchecking "Hidden parts" under the View menu.

Most often, users will see the Design view of the table. This is the one that shows hidden stubs or columns. Such stubs or columns are shaded in gray.

In example the "Total" stub is hidden.



This is the Display view of the same stub group.

When a table is first created, you see the design view of the table. When a table is run, the display view is automatically shown and hidden parts of the table are no longer visible. In order to switch back the design view (for example, so that you can select and unhide a part of the table that is hidden), use Ctrl+D or check "Hidden Parts" on the view menu.

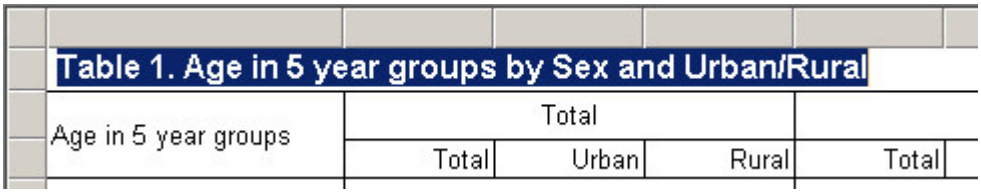
In addition to showing hidden parts, the design view also marks subtables with a colored outline while the display view does not. It is often useful to switch to the design view in order to be able to more easily set the tally attributes for a subtable. When the colored outline of subtables is displayed, you may right-click inside the outline and choose Tally Attributes (Subtable).

See Also: Using Print Preview, Hide or Show a Row or Column

How To ...

Customize Table Text

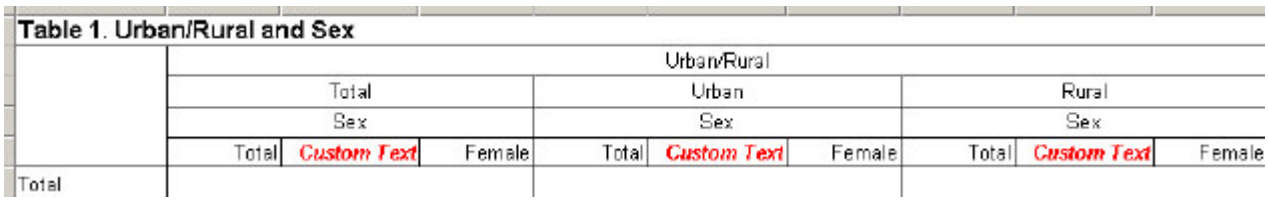
To customize the text in any table element (title, spanner, column head, stub, etc.) double click on the text you want to change. In the example below the table title has been selected.



You can now change the text of the table title. If you want a second line of text, press **Ctlr+Enter** to create a new line of text.

To go back to the original default text, right click on the text you want to change, select **Format (*table part*)**, uncheck the **Custom Text** check box, then press OK.

Note that if you customize the text of a stub or column header in a sub grouping, the text will be changed for all corresponding rows or columns in the sub grouping. In the example below, the text for the column header Male has been changed in one column and the change is reflected in the other sub groupings.

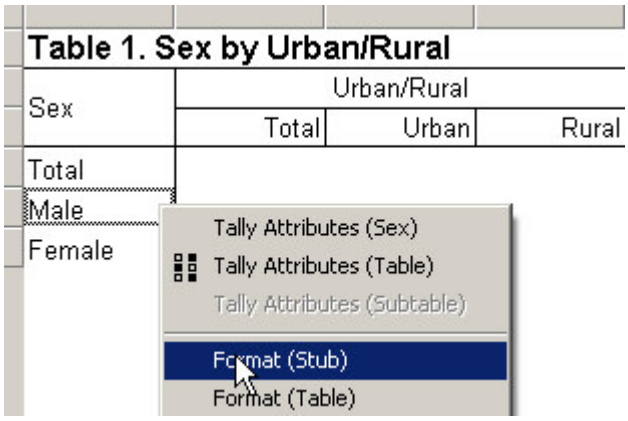


If you wish to change the text in only one sub grouping, you will have to use to multiple subtables with universes.

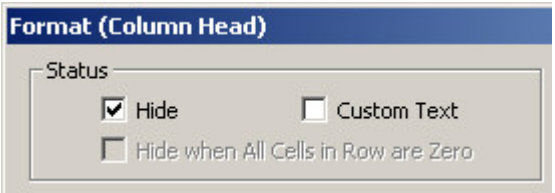
See Also: Change the Automatically Generated Text

Hide or Show a Row or Column

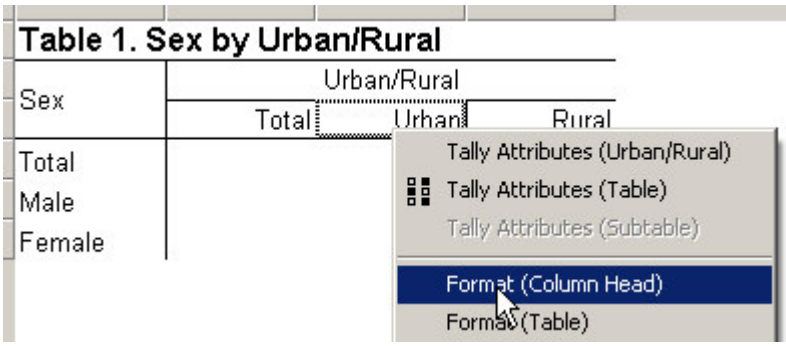
To hide a row, right-click on the stub for the row you wish to hide, select "Format (Stub)"



and check the "Hide" box.



Similarly, to hide a column, right-click on the column head for the column you wish to hide, select "Format (Column Head)"



and check the "Hide" box.

You can hide multiple rows or multiple columns at the same time by first selecting the rows or columns to hide and then right-clicking and choosing "Format (Stub)" or "Format (Column Head)" as above. Multiple selection is done in the usual Windows manner by left-clicking the mouse while using the Shift or Ctrl key. Using the Ctrl key adds each cell in the table that is clicked on to the selection. Using the shift key adds all the cells in between the two cells clicked on to the selection.

Hidden rows and columns will be shown with a grey background in the design view and will not be shown at all in the display or print views or on the printed page. To show a hidden row or column, right-click on the stub or column head, choose "Format (Stub)" or "Format (Column Head)" and uncheck the "Hide" box. If you are not in the design view, you may need to first switch to the design view in order to see the row or column that you wish to unhide. To switch to the design view, use Ctrl-D or select "Hidden Parts" from the View menu.

When working with sub-groupings (one variable dropped on top of another), hiding or showing a row or column in one sub-grouping will automatically hide or show the corresponding row or column in all of the sub groupings. In the example below, hiding the column "Male" under the spanner "Total", also hides the "Male" column under the "Urban" and "Rural" spanners.

Table 1. Literacy by Urban/Rural and Sex									
Literacy	Urban/Rural								
	Total			Urban			Rural		
	Sex			Sex			Sex		
	Total	Male	Female	Total	Male	Female	Total	Male	Female
Total									
Literate									
Illiterate									
Not Reported									
Not Applicable									

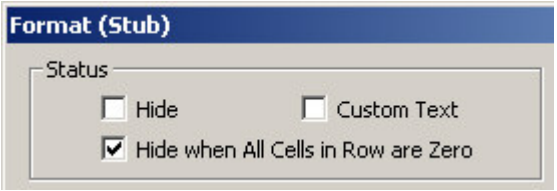
You cannot hide a row or column in only one sub-grouping but not the others. You can, however, achieve the same result using multiple subtables with universes.

To automatically hide rows when all data in the row is zero see Hide Rows Containing All Zeros.

See Also: Formats for a Part of a Table, Formatting Row, Column, or Cell Data, Hide or Change the Position of the Total

Hide Rows Containing All Zeros

It is possible to have CSPro automatically hide rows when all the cells in the row contain the value zero. This is useful when many rows in a table will have only zeros and you wish to view only the rows that have non-zero data.



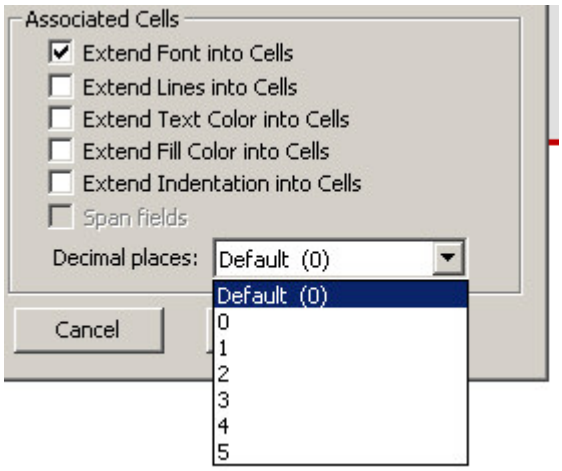
To have CSPro automatically hide a row if all values in the row are zero, right-click on the stub for the row and choose "Format (stub)". In the "Format (stub)" dialog, check the box "Hide when All Cells in Row are Zero". When you run the table, the row will be hidden if all cells are zero. To view the row again, in order to able to select it, switch to the design view. To switch to the design view, use Ctrl-D or select "Hidden Parts" from the View menu.

You can hide multiple rows if they contain all zeros at the same time by first selecting the rows and then right-clicking and choosing "Format (Stub)" as above. Multiple selection is done in the usual Windows manner by left-clicking the mouse while using the Shift or Ctrl key. Using the Ctrl key adds each cell in the table that is clicked on to the selection. Using the shift key adds all the cells in between the two cells clicked on to the selection.

Change the Number of Decimal Places Displayed

You can change the number of decimal places displayed in an individual data cell or for all cells in a row or column. To change the number of decimal places in a data cell, right click on the cell and choose Format (Data Cell). For rows, right click on the Stub and choose Format (Stub). For columns, right click on the column head and choose Format (Column Head).

In all cases, this brings up the Format (table part) dialog. Simply select the number of decimal places for the associated cells, i.e., the row, the column, or the data cell and click **OK**.



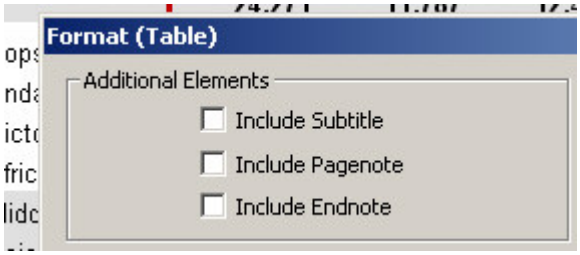
Rows or columns containing percentages are automatically designated as having one decimal place.

If you change the number of decimal places in a data cell to a number other than the default setting (indicated by the word "Default" in the pulldown), then the setting for the data cell will override any settings made for rows or columns containing that data cell. For example, if the data cell is set to one decimal place, and you change the row to have two decimal places, the data cell will still have one decimal place although other cells in the row will change. In order to make the number of decimal places in the data cell follow the number of decimal places set in the row or column, change the setting in the data cell to the default.

You can change the default number of decimal places for data cells in the preferences.

Add a Footnote (Pagenote or Footnote)

To add footnote to a table, right-click anywhere on the table and choose Format (Table) or choose Format (Table) from the Edit menu to launch the Format (Table) dialog.



Click to add a check mark to either the pagenote or endnote and then click **OK**. This will add a new row at the bottom of the table. Double click on this new row and type in the text for your footnote.

A pagenote will be displayed at the bottom of every page when the table is printed while an endnote will only be displayed on the last page of the printed table.

To remove a pagenote or endnote, uncheck the corresponding box in Format (Table).

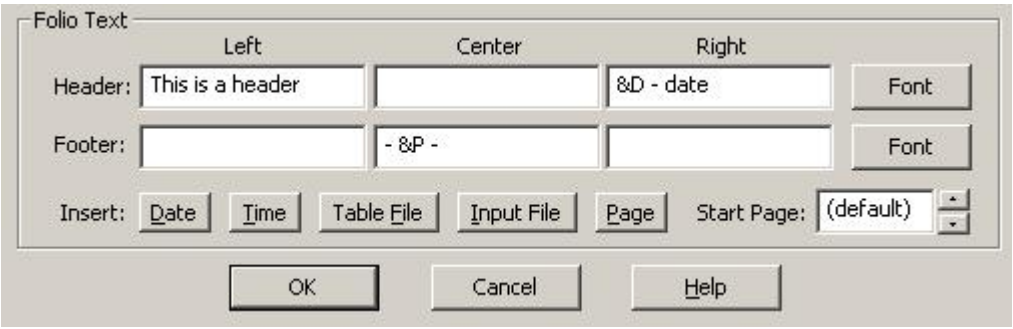
Format (Table) allows you add a pagenote or endnote to an individual table. To add an endnote or pagenote for all tables in a file, use the preferences.

See also: Add Header/Footer Text to a Table

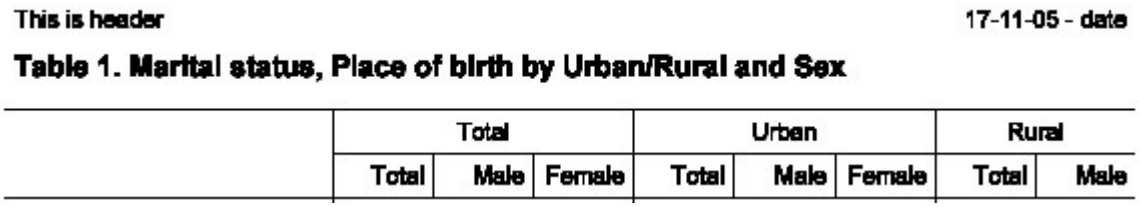
Add Header/Footer Text to a Table

To add header or footer text to a table, right click anywhere on the table and choose Format Print (Table) or choose Format (Table) from the Edit menu to launch the Format Print (Table) dialog.

This dialog has text boxes to enter left, right and center text for both the header and footer. "Header" refers to common text at the top of every printed page. "Footer" refers to common text at the bottom of every printed page. Left, Center, and Right refer to areas of the printed page.



The above options produce the following header:



The six text boxes can be filled with any text that you type. In addition, you can choose to display any combination of date, time, page number and file name in the header and/or footer by adding one of the following predefined text strings:

Text in header/footer edit box	Replaced with on printed page
&D	Date on which the tables were run
&T	Time at which the tables were run
&P	Page number relative to start page
&F	File name of the table file
&I	File name of the data input file(s)

To place predefined text in a text box, you can either type it directly or click in the text box where it should go and then click the button corresponding to the desired item. Predefined text strings may be mixed with other text, as in the right header of the example above. To delete text in the header or footer, delete it from the corresponding edit box.

Separate fonts are available for headers and footers. You can set these fonts using the font button to right of the text boxes.

If page numbering is used you may set the starting page number, the number to use for the first page of the table, in the "Start Page" text box. By default, the starting page number will be 1 for the first table in the file and for all subsequent tables in the file it will be one greater than the last page of the preceding table.

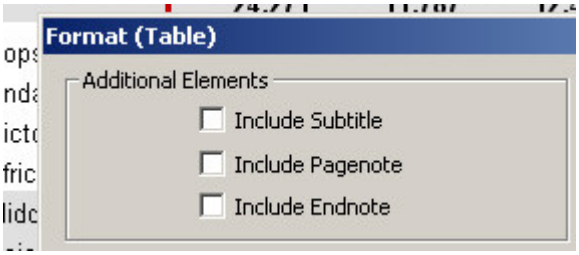
Using the Format Print (Table) dialog will set the header and footer text for the current table. You may set the header and footer text for all tables in the file using the preferences.

Note: Headers and Footers are visible only in Print Preview and on the printed page.

See also: Add a Footnote (Pagenote or Endnote)

Add a Subtitle

To add a subtitle to a table, right-click anywhere on the table and choose Format (Table) or choose Format (Table) from the Edit menu to launch the Format (Table) dialog.



Click to add a check mark next to "Subtitle" then click OK. This will add a new row in the table directly underneath the title. Double click on this new row and type in the text for your subtitle. You can then change the format (font, color, alignment, etc…) of your subtitle by right-clicking on it and choosing Format (Subtitle).

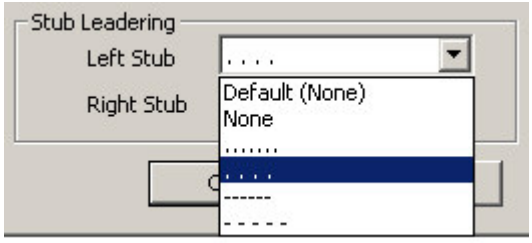
To remove the subtitle row, uncheck the corresponding box in Format (Table).

Add Stub Leadering

Stub leadering refers to text patterns such as "….." or "——" that are placed between the text of a stub and the first column of the table.

Marital status	
Total.....	24,2
Married(code 1).....	6,9
Divorced(code 2)	7
Widowed(code 3).....	7
Never Married(code 4).....	15,6
Place of birth	
Popstan.....	23,3
Endar.....	5
Victoria.....	1
Africa	

By default, no stub leadering is displayed. To add stub leadering to a table, right-click anywhere on the table and choose Format (Table) or choose Format (Table) from the Edit menu to launch the Format (Table) dialog.



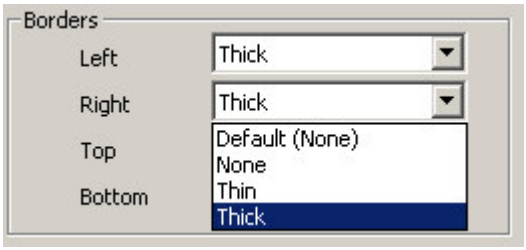
Choose the desired pattern from the Stub leadering pulldown menu. If you have chosen to display both left and right stubs, you can set different leadering patterns for each one.

Note: Stub leadering is only visible in Print Preview and on the printed page.

Using the Format Print (Table) dialog will stub leadering for the current table. You may set stub leadering for all tables in the file using the preferences.

Add Borders

To add borders to a table, right-click anywhere on the table and choose Format (Table) or choose Format (Table) from the Edit menu. This will launch the Format (Table) dialog.



Borders outline the table on the printed page. Different borders are available for each side of the table.

The options are:
None – no border on this side.
Thin – a normal line border on this side.
Thick – a "double" line border on this side.

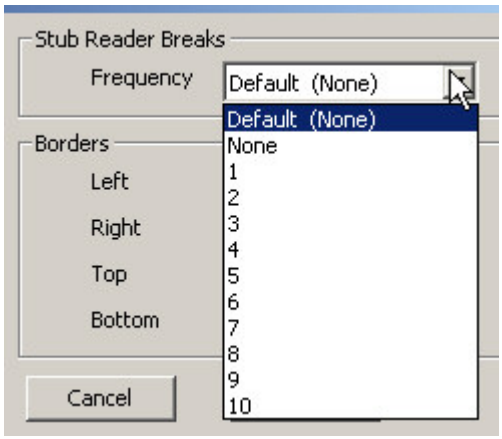
Using Format (Table) will set the borders for the current table. To set the borders for all tables in the file, use preferences.

See Also: Add Borders to Cells

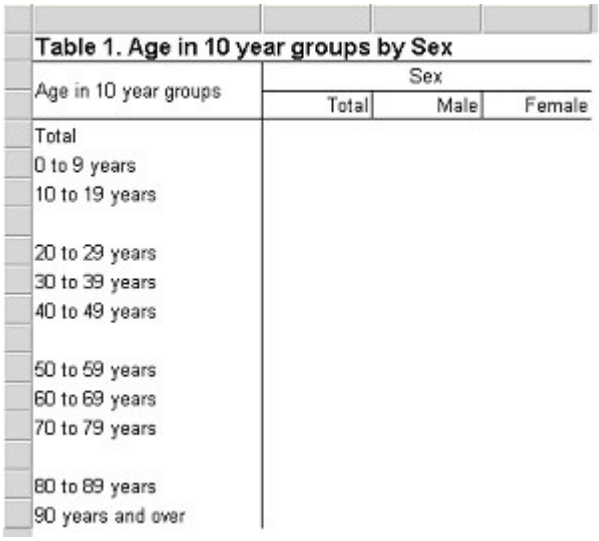
Add Reader Breaks

Reader breaks separate rows of data into readable groupings. This is done by inserting blank rows in the table at the specified intervals.

To add reader breaks to a table, right-click anywhere on the table and choose Format (Table). This will launch the Format (Table) dialog.



Select the number of lines (intervals) for each group.



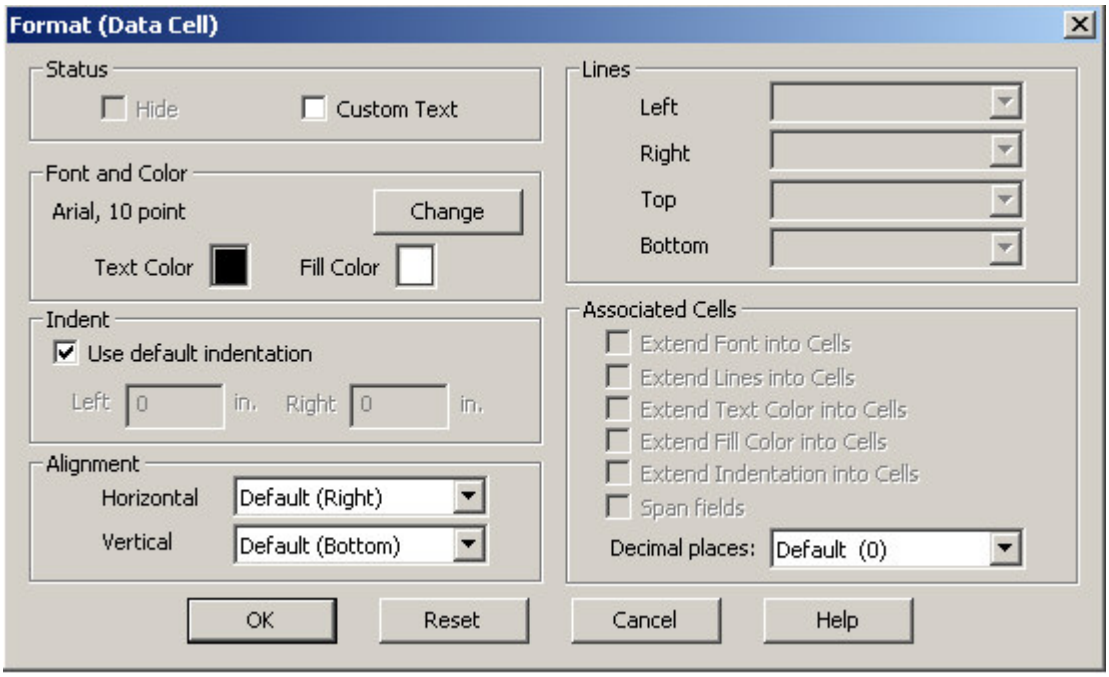
(Stub reader break frequency set to 3.)

The frequency of reader breaks may be set anywhere from 1 (every other row is blank) to 10 (every 10th row is blank). Note that you can hide individual reader breaks by right clicking on them and selecting Format (Stub). This allows for finer control over placement of reader breaks.

Note: Reader breaks apply to the entire table. Users may need to do additional spacing in the table rows to gain the desired result.

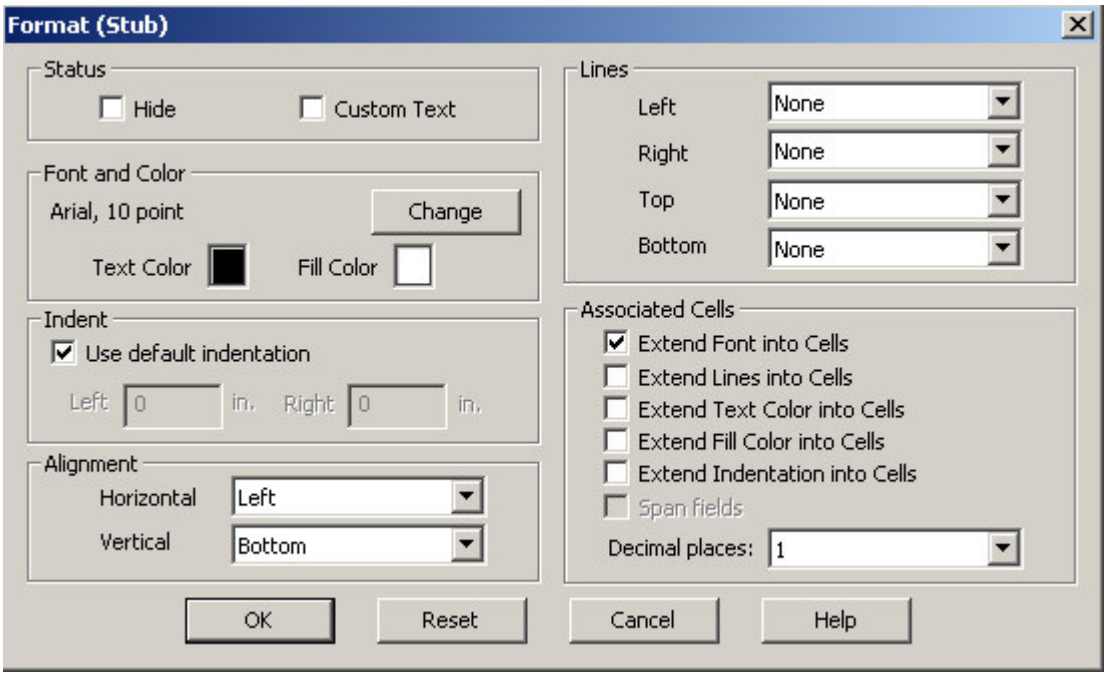
Change the Way Numbers are Displayed

If you want to modify the display of a single data cell or a group of unassociated data cells this is done by selecting the individual cells, right clicking, and using Format (Data Cell).



Individual Data cells cannot be hidden but you can replace their contents with Custom Text. Of course, the number of decimal places, fonts, alignment, and indentation can be changed using the options in the Format (Data Cell) dialog above.

You can hide or otherwise modify entire rows or columns of numbers. Select the rows or columns to be modified (multiple selection is done using Shift and/or Ctrl keys), right click and use **Format (Stub)** for rows or **Format (Column Head)** for columns.



The settings that have an effect on the numbers in the rows or columns associated with the stub or column head are in the "Associated Cells" section. These check boxes allow you to extend the format of the stub/column head into all the cells in the corresponding row/column. Of course, the number of decimal places affects only the data cells themselves and not the column head or stub.

See Also: Formatting Row, Column, or Cell Data, Hide or Show a Row or Column

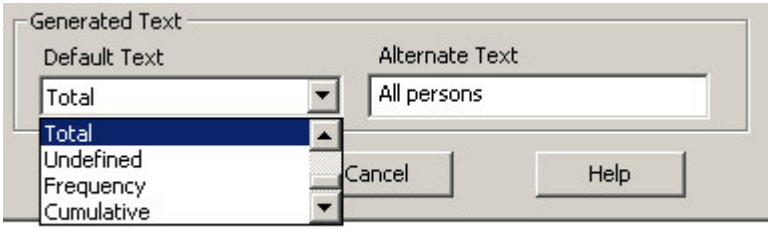
Change the Automatically Generated Text

Most text in tables comes from value labels in the value sets for the variables that you drop onto the table. Some

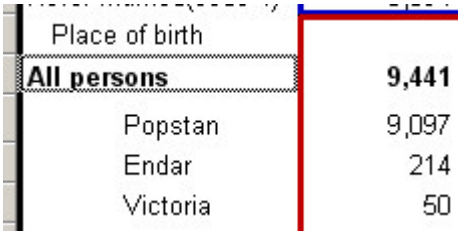
additional text, such as "Total", "Percent" and "by", is generated by the software. You can customize the text that the software will use for all of these terms.

This option is controlled through the Format (Application) dialog. Choose Format (Application) from the edit menu or right-click anywhere on the table and choose Format (Application).

Find a list of the words used in the generated text at the bottom of the menu. Select the "default text" which is to be replaced and enter the "alternate text" or replacement text.



Every occurrence of the default text will be replaced by the alternate text. In the example, every "total" text in every table will be replaced by "All persons".



If the text is not appropriate in certain places it can be replaced by "custom text ".

Note: Any text can always be replaced by custom text.

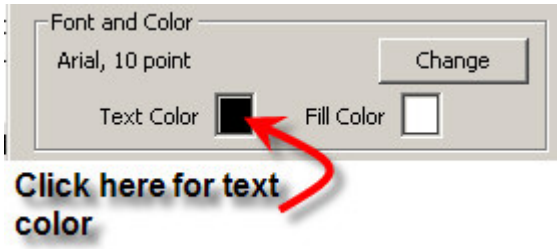
Changing the automatically generated text is particularly useful when generating tables in languages other than English. You can translate all of the automatically generated text into your language in the Format (Application) dialog and will be updated in all tables in your file.

Change Fonts or Colors

There is no "global" way to change fonts or colors in a table. The formats of the individual elements, e.g., Title, Spanner, Caption, etc., must be individually modified to replace the 'default' fonts and colors. To do so, right-click on the element and choose Format (<name of table part>) from the menu to bring up the Format (Table Part) dialog.

There is also the option to change the default formats for each of these elements. The modifications will take effect immediately in ALL the tables. See Preferences and Default Formats for more information.

In both circumstances, specific or default formats, the font and color are changed in the 'Font and Color' section of the format (Table Element) dialog.



To change the font, click on the 'Change' button and select the font characteristics from the available list.

To change the text or fill (background) click in the color box and select from the choices given.

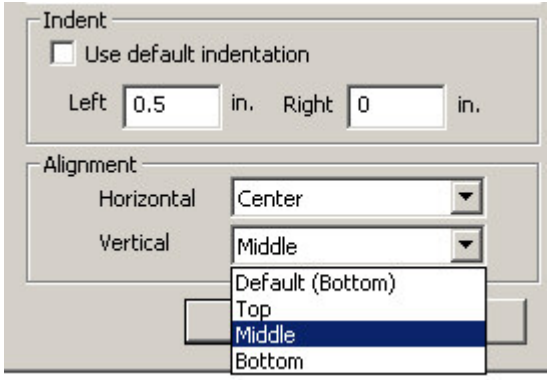
See Also: Preferences and Default Formats

Change Indentation or Alignment

There is no global way to change indentation or alignment of text. The formats of the individual elements, e.g., Title, Spanner, Caption, etc., must be individually modified to replace the 'default' alignment and indentation. To do so, right-click on the element and choose Format (<name of table part>) from the menu to bring up the Format (Table Part) dialog.

There is also the option to change the default formats for each of these elements. The modifications will take effect immediately in ALL the tables. See Preferences and Default Formats for more information.

In both circumstances, specific or default formats, alignment and indentation are changed in the 'Indent' and 'Alignment' sections of the format menu.



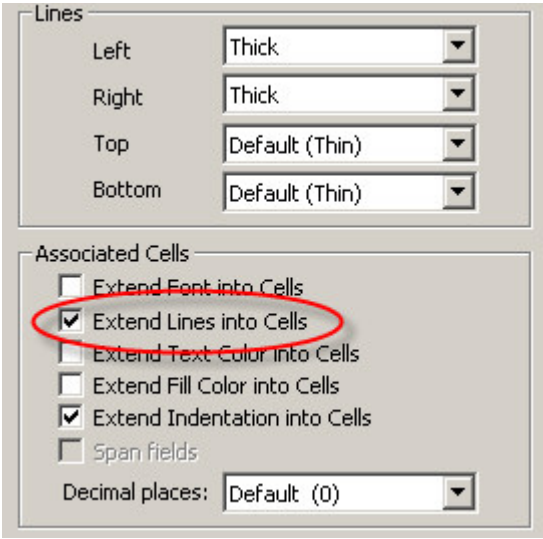
Indentation is given in inches (in.) or centimeters (cm.) depending upon the choice of Measurement System in the Format (Application) dialog. Indentation affects where in the text box, the text starts (Left) and/or ends (Right).

Alignment is the placement of text in the text box. The horizontal options are left(-justified), center, or right(-justified). Vertical choices (shown) are top, middle, and bottom.

See Also: Preferences and Default Formats

Add Borders to Cells

Individual cells have very few attributes that can be changed. Font, color, indentation, alignment, custom text, and number of decimal places are the only ones available.



Borders can only be created for cells by setting the borders on the rows and columns that contain the cell. To do this, right-click on the associated stub, caption or column head and choose Format (Stub), Format (Caption) or Format (Column Head) to bring up the Format (Table Part) dialog. For columns, set the left and right lines and for rows set the top and bottom lines. To ensure that line settings are applied to the entire row or column, make sure that **Extend Lines into Cells** is checked.

Table 2. Age in 5 year groups by Mother living

Age in 5 year groups	Mother living			
	Total	Yes	No	Don't know
Total	24,271	19,239	4,655	377
0 to 4 years	4,130	4,025	53	52
5 to 9 years	3,337	3,150	138	49
10 to 14 years	3,250	2,983	220	47
15 to 19 years	2,967	2,681	239	47
20 to 24 years	2,243	1,912	283	48
25 to 29 years	1,686	1,351	296	39
30 to 34 years	1,508	1,102	382	24

The column header "No" has thick lines set for left and right
and extend into cells is checked

You can also set the left and right borders of a spanner and have them extend into the leftmost and rightmost columns under the spanner. This is done the same way it is done for individual columns. Right-click on the spanner, choose Format (Spanner), set the left and right lines and ensure that **Extend Lines into Cells** is checked.

Table 2. Age in 5 year groups by Sex, Urban/Rural

Age in 5 year groups	Sex			Urban/Rural		
	Total	Male	Female	Total	Urban	Rural
Total	24,271	11,787	12,484	24,271	13,920	10,351
0 to 4 years	4,130	2,088	2,042	4,130	2,083	2,047
5 to 9 years	3,337	1,646	1,691	3,337	1,869	1,468
10 to 14 years	3,250	1,606	1,644	3,250	1,943	1,307
15 to 19 years	2,967	1,394	1,573	2,967	1,865	1,102
20 to 24 years	2,243	1,061	1,182	2,243	1,334	909
25 to 29 years	1,686	789	897	1,686	964	722
30 to 34 years	1,508	761	747	1,508	829	679
35 to 39 years	1,300	600	700	1,300	757	543
40 to 44 years	957	474	483	957	613	344

Left border for "Sex" spanner

Left border for "Urban/Rural" spanner

By default, stubs and columns have no borders, but spanners have thin lines on the left. This means that cells in the leftmost columns under a spanner will have left borders. You can change these default settings by using the Preferences.

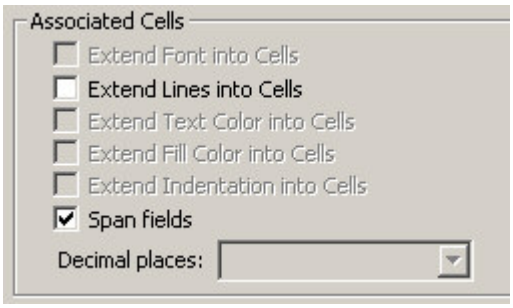
See Also: Formatting Row, Column, or Cell Data, Add Borders to a Table

Make Captions Span Data Cells

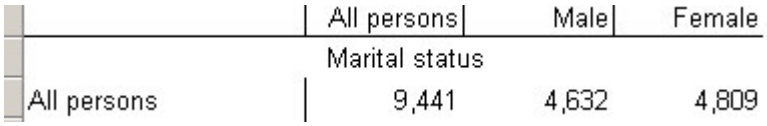
This option is controlled through the Format (Caption) menu.

Captions are rows without data. Normally the value set label is used as a caption before the stub group and the caption text box is restricted to the same size as the other stub text. However the caption text box can be widened to the width of the table by using its **Span fields** option. Right-click on the caption, choose Format (Caption) to bring up the Format (Table Part) dialog and check "Span fields".
An example:

Select the "Span fields" option from the Format (Caption) menu.



The result:

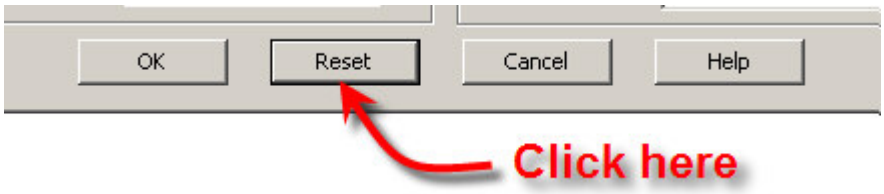


Note: The horizontal alignment of the caption was also changed to "center" to demonstrate the size of the associated text box.

By default, captions do not span fields. You can make all captions in all tables in the file span fields using Preferences and Default Formats.

Reset Format of Table Item to Default

All the format attributes of every element in a table have a "default" setting. These are established using the Preferences and Default Formats. Each attribute can be changed through certain entries in the Format (Table Part) dialog. To return any element to its "default" state, i.e., return to all the default settings in the dialog, just click on the **Reset** button at the bottom of the dialog.

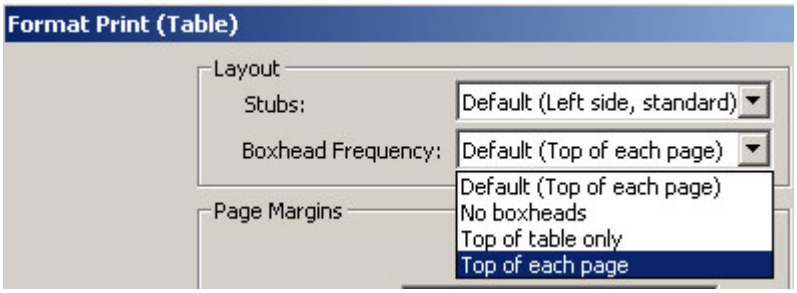


Note: There is **NO** "undo" for this operation. All custom entries are lost.

Change the Repeating of Boxheads

The boxhead refers to the area of the table containing the stub heads, column headers and spanners. You can control how the boxhead is displayed when the table is printed using the Format Print (Table) dialog. To bring up this dialog, choose Format Print (Table) from the Edit menu or right-click anywhere on the table and choose Format Print (Table).

The boxhead setting is found in the "Layout" section of the dialog. It only has meaning for a table that spans many pages and it only has an effect in the print preview and, of course, the printed table.



The options are:

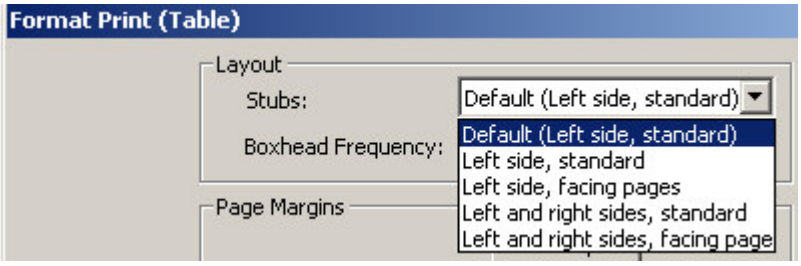
No boxheads – Only the table title is printed on each page.
Top of table only - Table title and box head only appear on the first page of table.
Top of each page – Table title and box head appear on each page of the table.

Use the Format Print (Table) dialog to set the boxhead frequency for a single table. Use Preferences and Default Formats to set the default boxhead frequency for all tables in the file.

Change Stub Column Position

Stubs are the text associated with each row of the table. By default the stubs appear to the left of the first data column. You can place the stubs for a table on the right, or on both the left and right as well as placing them on the left and right across a two-page spread when the table is printed. To bring up this dialog, choose Format Print (Table) from the Edit menu or right-click anywhere on the table and choose Format Print (Table).

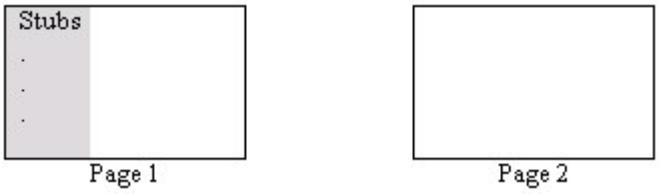
Stub position is found in the "Layout" section of the dialog. It only has an effect on the print preview and, of course, the printed table.



There are four options:

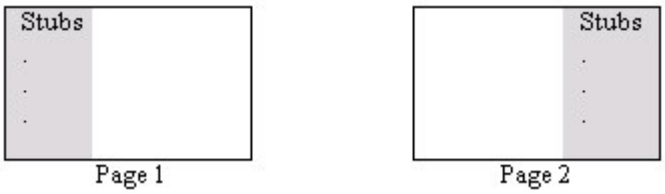
- Left side, standard (default): place stubs along the left side of every page. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left sides of both pages.

- Left side, facing pages: place stubs along the left side of the page but only for the first page of facing pages. If the table is wide enough so that some of the columns go onto a second page, no stubs will be placed on the second page. If the table columns all fit on one page then this is the same as Left side, standard.



- Left and right sides, standard: place stubs on both the left and right sides of the page for every page. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left and right sides of both pages.

- Left and right sides, facing pages: place stubs along the left side the first page of facing pages and on the right side of the second page of facing pages. If the table is wide enough so that some of the columns go onto a second page, stubs will be placed on the left side on the first page and the right side of the second page. If the table columns all fit on one page then this is the same as Left side, standard.



Use the Format Print (Table) dialog to set the boxhead frequency for a single table. Use Preferences and Default Formats to set the default boxhead frequency for all tables in the file.

See Also: Viewing Multiple and Facing Pages

Creating Tables by Geographic Area

Area Processing

Area Processing is the generation of tables by a defined hierarchical structure of area codes. If the geographic hierarchy of a hypothetical country is region, province, locality (major-to-minor order), it may be necessary to produce tables for each of these sub-divisions.

To use area processing you must have done the following:

• Created an area names file (.anm)

• Selected one or more Area IDs from the area IDs dialog box

For example, suppose we wish to perform geographic area processing for our top-most units of geography, which are Province and District (choose them in the Area IDs dialog box). Each tabulation will be repeated for each Province and District in the data file. In addition, a summary entry [Total] will be shown for the entire data file. The tabulation will be displayed in the following order:

```
Total
   Province 1
      District 1 (of Province 1)
      District 2 (of Province 1)
      District 3 (of Province 1)
   Province 2
      District 1 (of Province 2)
      District 2 (of Province 2)
      District 3 (of Province 2)
   :
   :
```

Area Processing will apply to **all** tables in your application.

No File Processing: It is also possible to produce tables without creating an area names file. Select the levels of geography desired but after clicking on the Traffic Light button in CSPro to run your tables, leave the <Area Names> field blank. CSPro will use codes from the data file instead of names for each level of geography. For example, if province Gluten's code is 20, the tables will show code 20 instead of the name Gluten. This can be useful for generating a list of every geographic code found in a data file.

See also: Area Names File,

Create an Area Names File

The Area Names file (.anm) is a text file that you can create using any text editor. Be sure you save this file with the extension .anm. This file defines the levels of geography and assigns text names to the numeric codes for each geographic unit.

File Format

The following is an excerpt from the area names file for Popstan, which you can find in the Examples folder. The first section identifier, `[Area Names]`, indicates the type of file, followed by the CSPro version number.

Next, the `[Levels]` section provides the names of the geographic hierarchy (levels) in major-to-minor order.

Finally, the `[Areas]` section gives the correspondence of the geographic codes (major-to-minor order) found in the data with a name for the geographic area. More detail follows the listing of the area name file.

Area code values can be positive, negative or zero.

If the data file contains area codes that do not have corresponding names in the area names file then the unmatched area codes will be displayed as a text string in place of the missing name.

```
[Area Names]
Version=CSPro 7.7

[Levels]
Name=Province
Name=District

[Areas]
X X = Popstan

1 X = Artesia
1 1 = Dongo
1 2 = Idfu
⋮

2 X = Copal
2 1 = Baja
2 2 = Bassac
⋮

3 X = Dari
3 1 = Argentina
3 2 = Benlata
3 3 = Bristol
⋮
```

The line following the `[Areas]` section contains the codes and name of the "total" area ("country" in our case). It is considered the "Grand Total" level and denoted by 'X' values for each level of the area hierarchy. In the example, the first 'X' represents the Province code and the second 'X' represents the District code. Basically, an 'X' value is similar to a "wildcard" match, so any value in this field is acceptable (and thus part of the area).

Following the total area name is the set of codes and names for the lowest valued major level, "province" in our example. The lowest code for a province is the '1' associated with Artesia. Again, an 'X' value is given for the District code since any code here is acceptable. Next, we must give codes and name for all districts in Artesia starting with the lowest code value.

Note that each line for this province begins with '1' since the province code must be combined with the district code to uniquely distinguish this district. Dongo is district '1' of Artesia (province '1'). Data for a questionnaire with Province code = 1 and District code = 1 will be tallied for Dongo District.

When districts for Artesia are all listed (codes and names), start with the next lowest province code followed by its districts. The process is repeated for each province.

Area codes must be listed in ascending sort order from major to minor.

If the area name file has only one level, e.g., province, then only one code would be given. If three levels were needed, e.g., province, district, village, then three codes would be required. As always, 'X' represents the wildcard match.

The indentation associated with the names above is for illustrative purposes only. Area names shown in tables will not be indented, but indenting the file can make it easier to read.

Also, codes within the `[Areas]` section can be separated by commas, spaces, or a combination of both. Any of the

following are acceptable to define an item at the district level:

```
3 15 = Sharif
3, 15 = Sharif
3,15 = Sharif
```

Hiding Tables

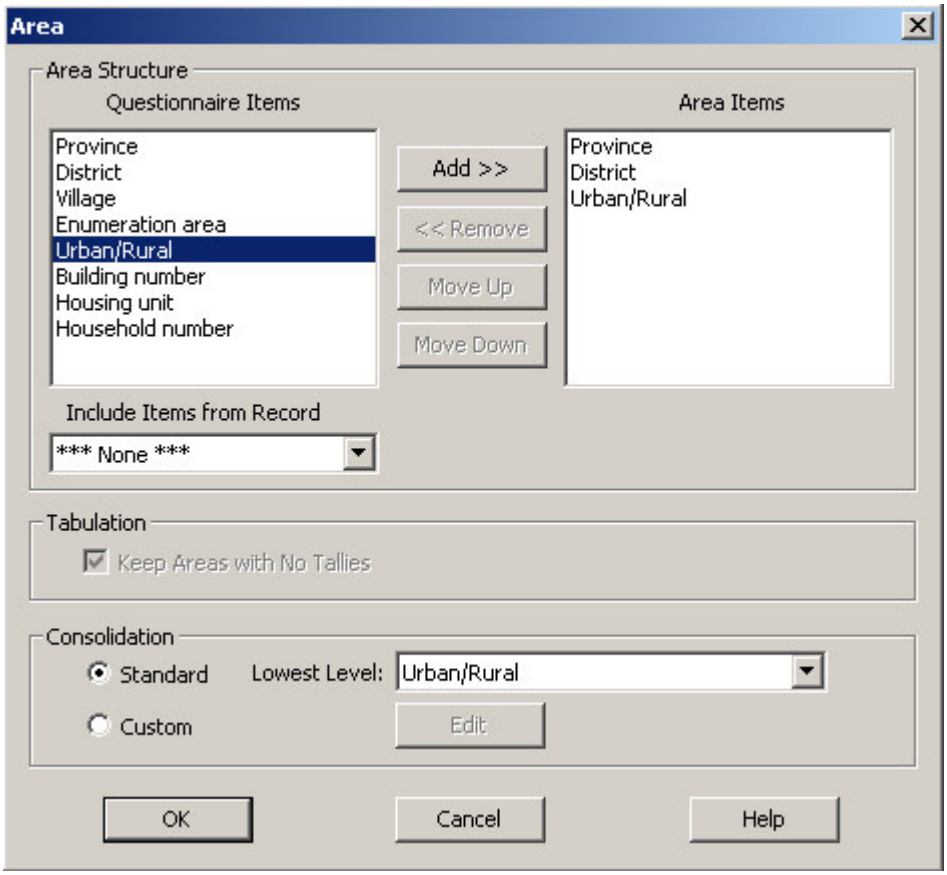
To hide tables for certain level of geographies, preface the area name label with a tilde (~). For areas with this label, a table will still be created (and stored in the .tbw table file), but it will not be displayed and will thus be hidden. For example, if a table shows states (provinces) and counties (districts) but some states only contain one county, it may be desirable to suppress the repetition of the data at both the state and county level. For example:

```
X X = United States

1 X = Maryland
1 1 = Montgomery Country
1 2 = Prince George's County

2 X = District of Columbia
2 1 = ~ District of Columbia

3 X = Virginia
3 1 = Arlington County
3 2 = Fairfax County
```

Without the tilde, a table would be displayed for District of Columbia as both a state and a county. Adding the tilde leads to the desirable behavior of only displaying one table for that level of geography.

Area Dialog Box

This dialog box allows you to select the Questionnaire ID items to be used for area processing.



Area Structure Section

Codes for these items are used to 'tally' in the specific areas. Area items can come from the questionnaire ID items or items on singly occurring records. Available single record names will be available in the 'Include Items from Record' menu. If selected those items will be added to the list under 'Questionnaire Items'.

Note: The number of the Area IDs items determine the number of [Levels] needed in the Area Name File.

- Select one or more Questionnaire IDs from the left box (to select multiple items, hold down the Ctrl key when you make your selections).
- Click the 'Add' button to copy your choices to the Area IDs list.
- To remove an item from the Area IDs list, select it and click the 'Remove' button.
- To re-order items in the list, select one and click the 'Move Up'/'Move Down' button as appropriate.

Tabulation Section

In future versions, users will be able to decide if areas without any counts [no tallies] are displayed in the table or not.

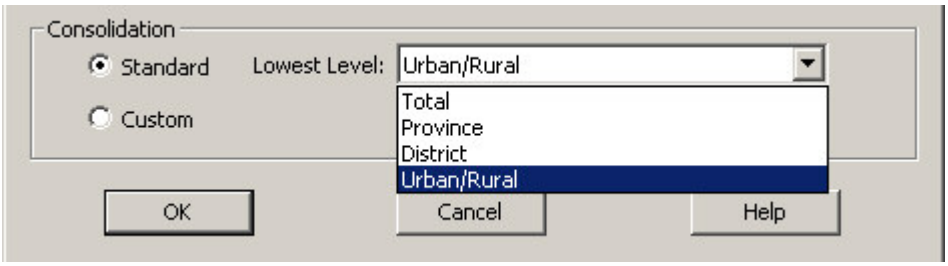
- Show area in table even if no counts are tallied [always in version 5.0].

Consolidation Section

Generally, only tables at the lowest level are created when the data file is processed. After these are completed these lowest levels are combined and recombined to create the higher level tables.

The two options here are Standard or Custom.

'Standard' follows the usual fashion of aggregation, i.e., adding to get total. Each lower level is combined to create the next highest level.



As an example from above, the original tables [created from the data file] would be an urban table for each district in each province and a rural table for each district in each province.
Standard Consolidation would:

- Add the appropriate urban and rural tables to create the province-district table.
- Add the appropriate district tables to create the province table.
- Add all province tables to create the 'country' table.

This procedure produces four levels of tables: Urban/Rural, District, Province, and Country (given in minor to major order).

The **Lowest Level** allows users to select the most minor level to be produced. Counts at this level and higher will appear in the tabulation but any lower level counts will be discarded.

Custom option allows some freedom in defining the aggregation or consolidation scheme. See Custom Consolidation for more information.

See also: Area Processing, Create an Area Names File

Area Captions

Area Captions are, as the name indicates, row captions in which the corresponding area name is placed for each table.

Area captions are placed just above the total line of the first stub group in the table. An area caption is denoted by the "%AreaName%" text.

Table 1. Urban/Rural by Sex

Urban/Rural	Sex		
	Total	Male	Female
%AreaName%			
Total			
Urban			
Rural			

Area Caption

When the table is run, the %AreaName% is replaced by the names from the area names file.

Table 1. Urban/Rural by Sex

Urban/Rural	Sex		
	Total	Male	Female
Popstan			
Total	24,271	11,787	12,484
Urban	13,920	6,705	7,215
Rural	10,351	5,082	5,269
Artesia			
Total	1,500	758	742
Urban	919	462	457
Rural	581	296	285
Copal			
Total	2,630	1,310	1,320
Urban	1,553	746	807
Rural	1,077	564	513

Area Caption

If the table is a 'one-row' table, e.g., it has only column items but no stub groups, the area name caption replaces the 'Total' stub that would otherwise be present and the area caption is 'hidden'.

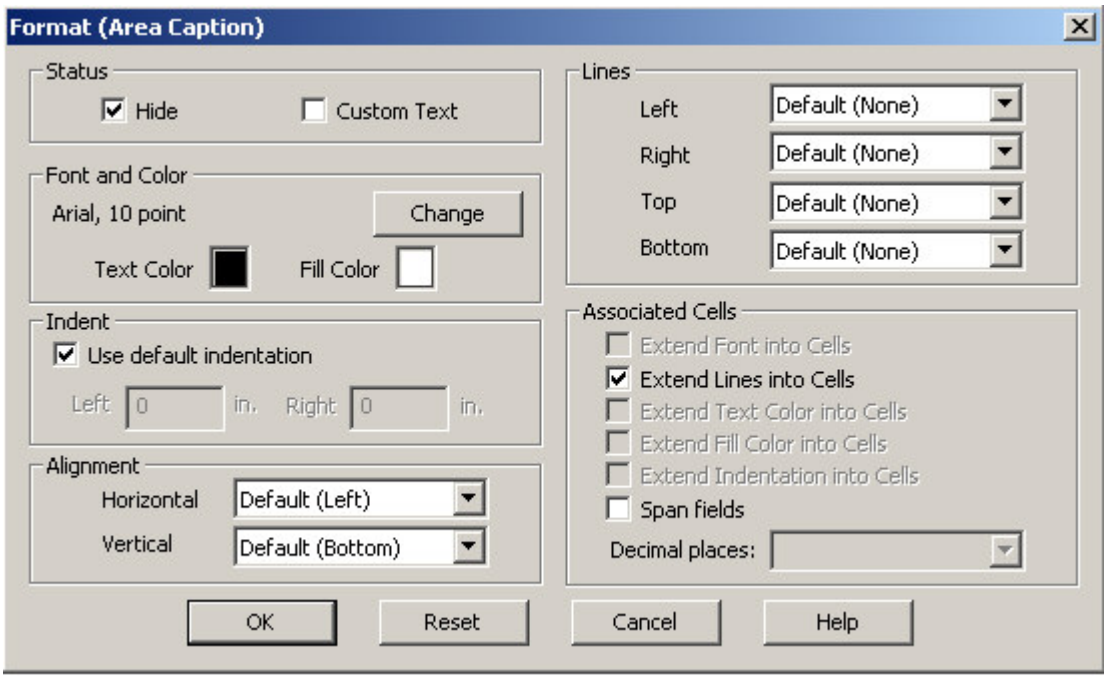
Table 3. Sex

	Sex		
	Total	Male	Female
%AreaName%			
%AreaName%			

Area Caption

Stub

The area caption has the same formatting options as other captions, available by right clicking on the caption and choosing Format (Area caption). You can hide area captions, as in the example above, you can change the fonts, make them span rows, etc…

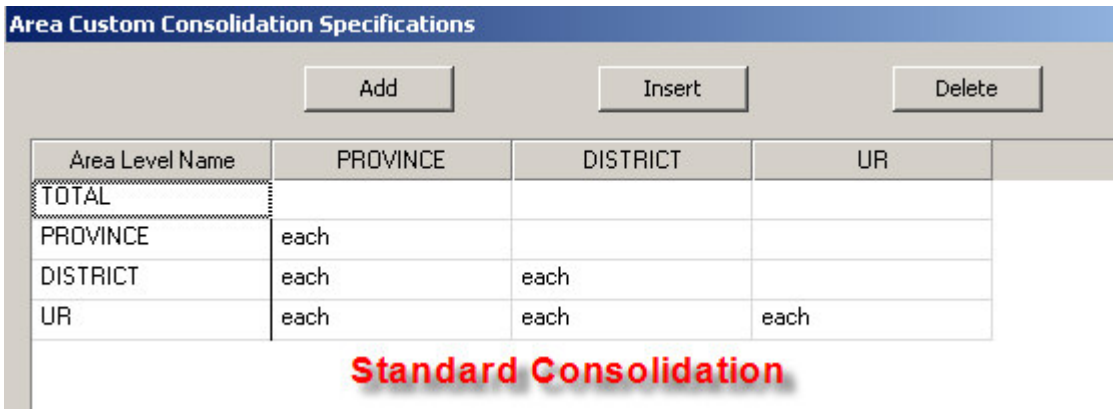


You can also set the default formats for all area captions in all tables in your file using Preferences and Default Formats.

See Also: Formats for a Part of a Table

Custom Consolidation

Custom Consolidation allows the aggregation of lowest level tables to create almost any reasonable higher level. Users define the consolidation scheme starting from the **Standard** consolidation, shown below.



Standard Consolidation would:

- Retain each Urban and Rural (UR) table for each district in each province.
- Add each group of UR tables to create a table for each district in each province.
- Add each group of district tables to create a table for each province.
- Add all province tables to create a single 'country' table.

The result of the consolidation is:

- One Country table
- A Province table for each province
- A District table for each district in each province
- One urban table and one rural table for each district in each province.

Each higher level is the aggregation of the levels below with the corresponding area code.

Using custom consolidation, you can produce tables at only the levels you wish. For example, if you did not want to produce tables at the district level but wanted to continue producing them for province, urban/rural and entire country; you would simply remove the district row:

Area Level Name	PROVINCE	DISTRICT	UR
TOTAL			
PROVINCE	each		
UR	each	each	each

Using Custom Consolidation you can also create user-defined schemes for combining the lowest level tables by adding new rows to the consolidation scheme:

Area Level Name	PROVINCE	DISTRICT	UR
TOTAL			
TOTAL_UR			each
PROVINCE	each		
PROVINCE_UR	each		each
DISTRICT	each	each	
UR	each	each	each

Each row of the scheme must have a CSPro 'name' for the set of tables to be created, listed under **Area Level Name** above.

In the example above, the following tables would be created:

- One Country table
- ***One urban table and one rural table for the entire country***
- A Province table for each province
- ***One urban table and one rural table for each province***
- A District table for each district in each province
- One urban table and one rural table for each district in each province.

In addition, you may place conditions in the cells of the consolidation scheme grid. These conditions are used to modify how the tables replaced for a given level (the column) are consolidated for a particular consolidation scheme (row).

The options are:

- Blank – Any code in this position is included.
- Each – Each different code in this position creates a separate table.
- Single value – One table created for this value of the area level
- Replacement formula – One 'summary' is table created for all areas that the meet the condition and the replacement code is substituted for the area code for the summary table. [start:end = replacement]

The last two options are illustrated by the following example:

Area Level Name	PROVINCE	DISTRICT	UR
TOTAL			
TOT_URB			1
PROV_1_4	1:4=99		

The TOT_URB table will include all tallies that had a value of '1' for the UR area level. The area codes on this table will be Province=X, District=X, UR= 1.

The PROV_1_4 table will include all tallies that had Province codes of 1 through 4. The area codes for this table will be Province=99, District=X, UR= X. Since other province codes are 1 to 15, the table associated with '99' would be displayed after any for those coded. (In the example above NO province tables were created hence they are not displayed)

In order to use these codes – a corresponding name should be in the area name file.

Table 1. Sex

	Sex		
	Total	Male	Female
Popstan	24,271	11,787	12,484
Total Urban	13,920	6,705	7,215
99	10,223	5,015	5,208

No area name for '99

Note: When custom consolidation is used it is the user's responsibility to make sure the consolidation scheme is reasonable and that it works in the desired manner. As usual, every aspect of a data processing system should be tested for correctness.

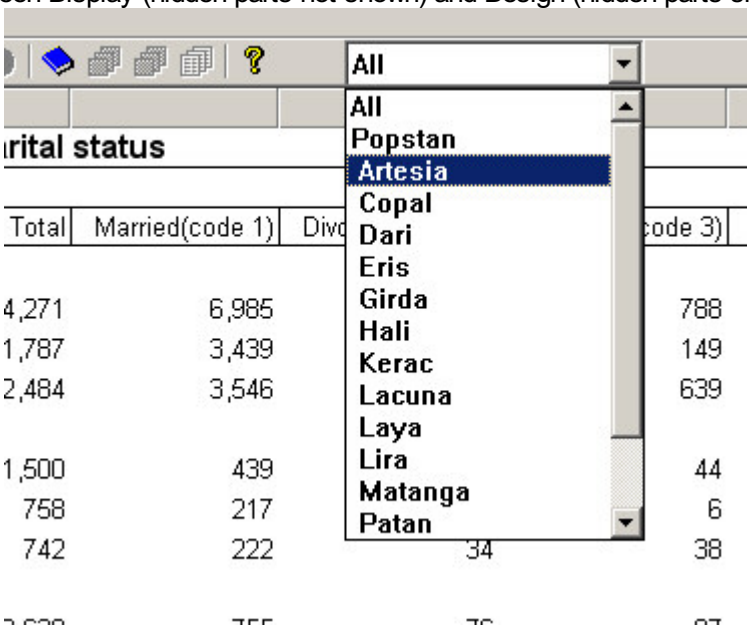
How To ...

Display Results for One Geographic Area

When a Tabulation Application uses the Area Processing option, the table usually displays the counts for each 'area' concatenated one after the other sorted by increasing area code values.

If you want to display the counts for a certain geographic then select it from the drop down menu available in Display view.

Note: Ctrl+D switches between Display (hidden parts not shown) and Design (hidden parts shown) views.



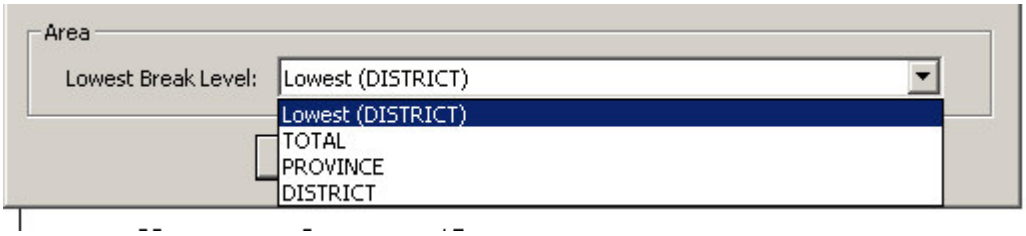
The 'All' selection represents the usual display. If area codes are found in the data file that are not defined in the Area Name File then those codes will be listed. The areas are listed in sort order, i.e., the same order that they are defined in the area names file.

Note that this only changes which area is currently displayed. Tables are still produced for all areas originally specified and all of these areas will still appear when the tables are printed or saved. To only produce tables for certain geographies either set the lowest break level for a table, or use custom consolidation.

Tabulate only Certain Levels of Geography

When the area structure is defined in the Area Dialog Box it defines the levels at which all the tables in the Tabulation application will be tallied. This can be modified for individual tables using the Tally Attributes for a Table.

The last section in the menu allows the selection of a lowest area break level for this table only. By default it will be set to the lowest area level available.



In the example above, if a certain table is to be produced only at the province level then that item would be selected as the **Lowest Break Level** instead of district.

Printing Tables

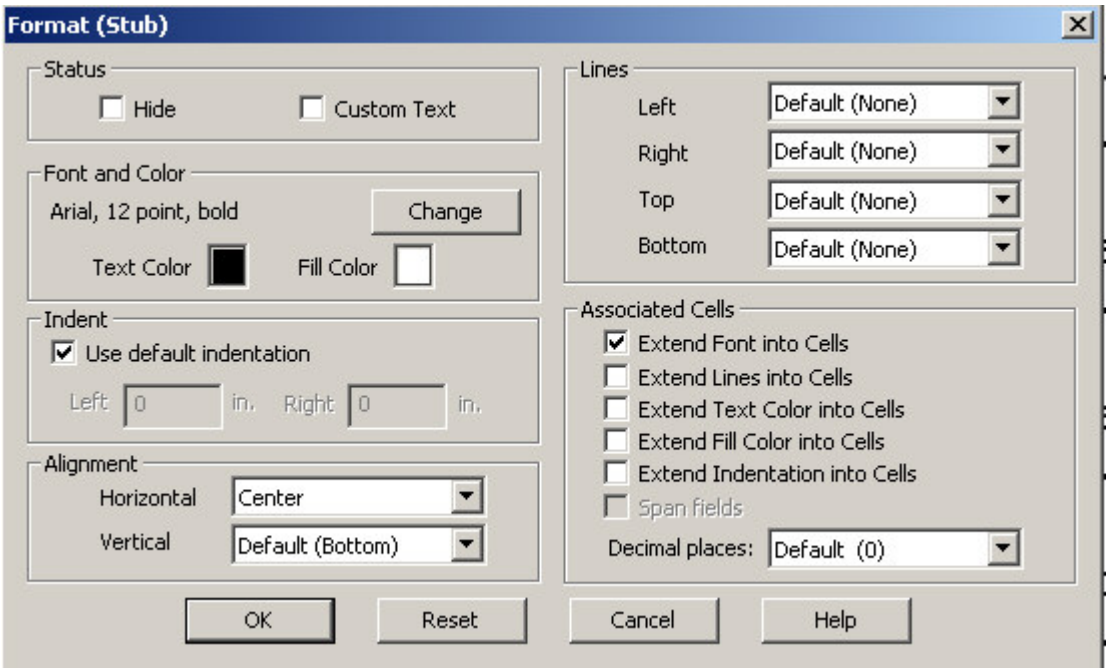
Using Print Preview

Print Preview is a view of the tables as they are display on the printed page, i.e., what they will look like when they are printed on the default printer. To view Print Preview, press the button on the toolbar or under **File** menu, select **Print Preview**.

Note: No Print Preview is available if there is no printer installed!

This is just one of three views available for a Tabulation Application. To return to Display or Design View use the "Close" button or just click the Print View icon [the icon is depressed when active and "undepressed" when not active], both are in the tool bar.

All the formatting done through the various format menus of a table are carried over to print preview. For example all the settings used in the following would be applied in this view.

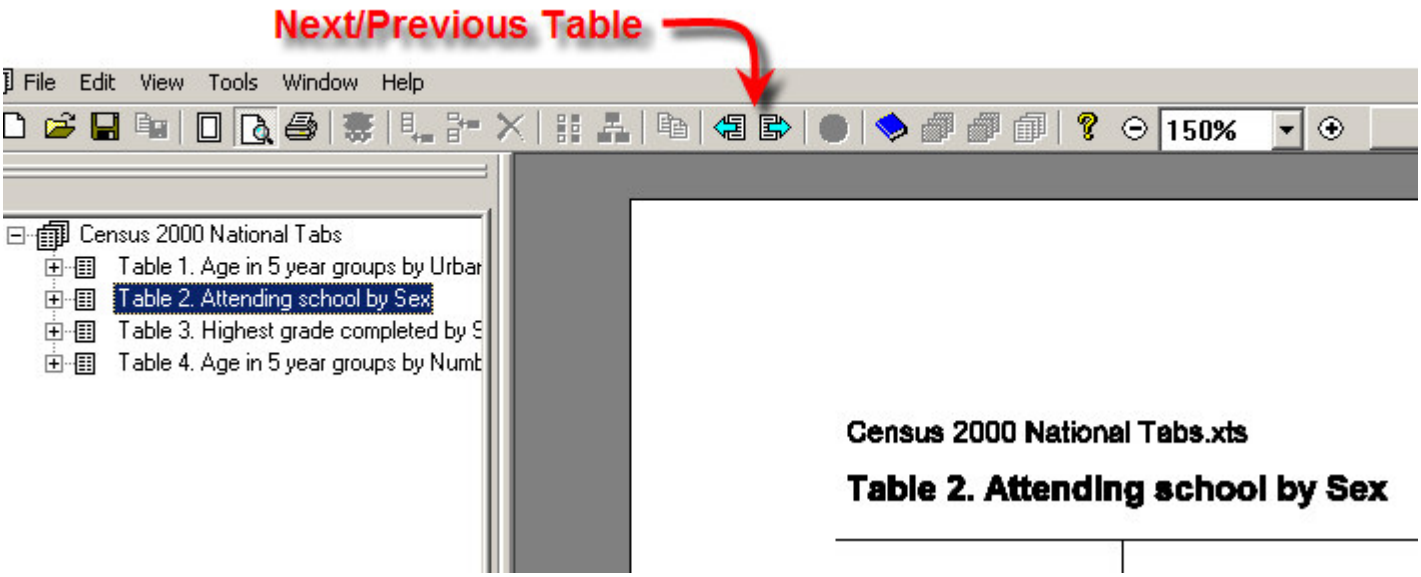


The spacing or sizing of rows or columns in Display view is NOT carried over to Print Preview.

Navigating Between Pages, Tables, and Areas

The easiest ways to move between tables are

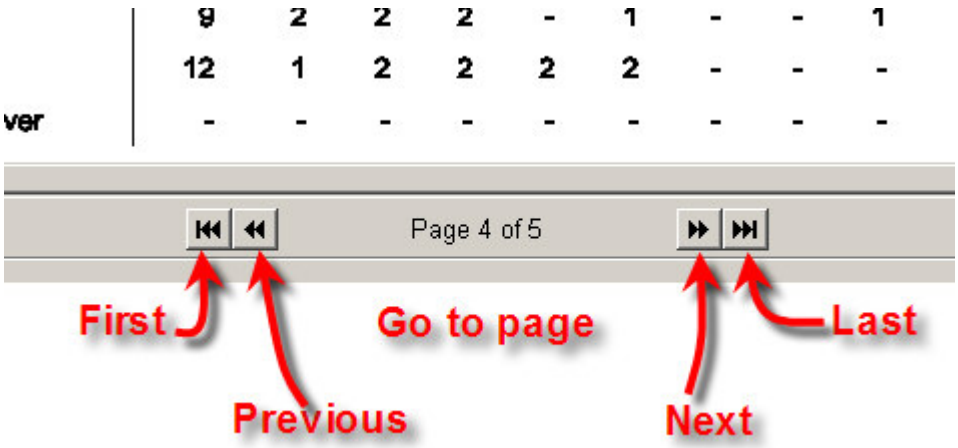
- Use the "Next Table" or "Previous Table" icons on the toolbar [see below].
- Select the table from the table tree on the left part of the screen [see below].



The easiest ways to move between pages of a table are:

- Use Home, End, Page Up or Page Down keys.
- Use navigation buttons at bottom of preview window (see below).
- Use the mouse "wheel" (if available and activated). One "click" (of the wheel) toward the user goes to the next page, one "click" away from the user goes to the previous page.

(Note: this behavior is only available in the page view (100%))



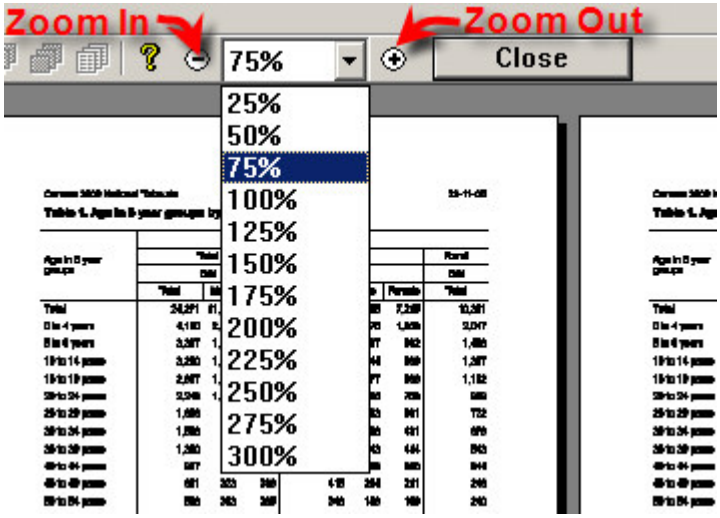
See the View menu for other combinations of keys that will allow you to move between tables.

There is also a "Goto ..." option available from the Edit menu and from the right-click menu which allows you to go directly to a particular page, table or area within a table.

See Also: Print Preview Options

Viewing Multiple and Facing Pages

The view presented when Print Preview is activated is that of the first page of the first table. Users can move between pages of the tables (see Navigating Between Pages, Tables, and Areas. Users can change magnification level [zoom in or zoom out] by using the drop down menu available at the top of the window or by clicking on the zoom in and zoom out buttons on the toolbar.



The "100%" setting is the full page and default setting. A value less than 100% shows multiple pages in the window [zoom out]. A value more than 100% shows less than one page in the window [zoom in].

If the current view is 100% or less -
A single wheel "click" toward the user will display the next page. A "click" away from the user will display to the previous page. This is true even if more than one page is displayed in the window. Of course, when an additional page is displayed one of the other pages is no longer in the window.

If the current view is greater than 100% -
The wheel activates the vertical scroll bar of the table.

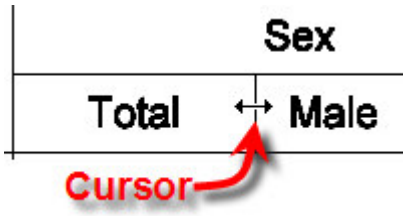
If the Ctrl key is held down and the wheel used –
A single wheel "click" toward the user will display the next "zoom in" option. A "click" away from the user will display the next "zoom out" option. Note that these wheel "click"s are executed one at a time so in the case of multiple clicks it may take time for the display to adjust.

You can activate **facing pages** mode by choosing it from the View menu. This mode simulates what the printed tables would look like if they were bound together in a book. The two pages that would face each other in the book are shown together. When in facing pages, the next page and previous page buttons will move to the next two pages that face each other rather than moving one page at a time. If you change the zoom level while in facing pages mode, facing pages is turned off.

Modifying Row and Column Spacing for Printing

The width of columns and the height of rows can be modified in Print Preview. Sizing is also available in Design and Display views but it does NOT carry over into Print Preview.

To modify the width of a column, place the cursor on the "vertical bar" in the box head that you want to move. The cursor will change to a horizontal double-sided arrow, left click the mouse and drag the vertical bar to the new position.



To modify the height of a row place the cursor below the row text until it changes to a vertical double-sided arrow, left mouse click and drag the (usually invisible) horizontal bar to the new position.

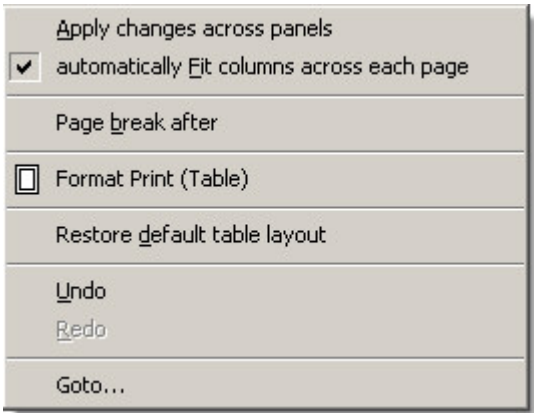
Modifications to row height and column width work in conjunction with the "Apply changes across panels" and "automatically Fit columns across each page" options. See Print Preview Options for more information.

Any changes made can be undone using the **undo** command. To reset the column and rows spacing back to its original settings, use **Restore Default Table Layout**. Both of these commands are available by right-clicking on the table while in the Print Preview.

See Also: Print Preview Options

Print Preview Options

You can make some modifications to the table in Print Preview. Most notably the rows and columns can be "sized" similar to the way it is done in spreadsheet software. There is also a "right-click" menu with additional options.

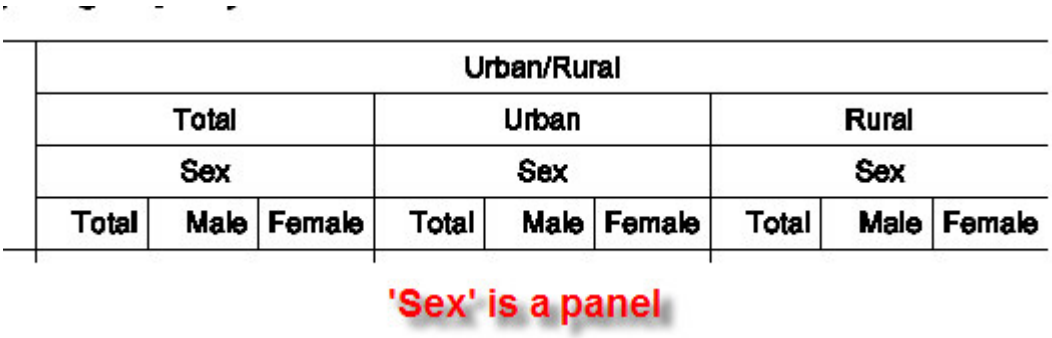


Briefly these options are:

Apply changes across panels

If unchecked this means that changes to row height or column width will be applied to only the selected row or column. If checked any sizing changes made to a row or column in a panel will be reflected in the corresponding row or column of each panel.

For example below are three panels for columns:



If this option is checked and a change is made to the width of the "Male" column in any of the three panels then the width of all three will be modified. If the option is unchecked then only the specific column selected will have its width modified.

Automatically Fit columns across each page

If checked this option will "spread" the table columns for all tables across the entire printed page. If unchecked, the columns for all tables will be displayed in the minimal width possible given text, font, etc.

An example of unchecked:

Table 1. Sex

	Sex		
	Total	Male	Female
Popstan	24,271	11,787	12,484
Artesia	1,500	758	742

Columns NOT 'fit'

An example of checked:

Table 1. Sex

	Sex		
	Total	Male	Female
Popstan	24,271	11,787	12,484
Artesia	1,500	758	742

Columns Fit Across Page

Page Break After

Insert a page break [go to next page] after the row selected. This usually done when a stub group is "split" across two pages.

An Example before Page Break applied. The user right-clicks on the "10-14 years" row and selects page break after to place a break between the "10-14 years row" and the "15-19 years row".

Table 2. Age in 5 year groups by Sex

Age in 5 year groups	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
0 to 4 years	4,130	2,088	2,042
5 to 9 years	3,337	1,646	1,691
10 to 14 years	3,250	1,606	1,644
15 to 19 years	2,967	1,394	1,573
20 to 24 years	2,243	1,061	1,182
25 to 29 years	1,686	789	897
30 to 34 years	1,508	761	747

Cursor in "10 to 14 years" row

Next page after Page Break applied. This page starts with the "15-19 years row"

Table 2. Age in 5 year groups by Sex (continued)

Age in 5 year groups	Sex		
	Total	Male	Female
15 to 19 years	2,967	1,394	1,573
20 to 24 years	2,243	1,061	1,182
25 to 29 years	1,686	789	897
30 to 34 years	1,508	761	747

Format Print (Table)
See Formats for Printing

Restore default table layout
The default layout of a table is the layout produced for it when the table is first created. Any changes made to the print view of the table will be "undone". There is also an "undo" of a single previous action in Print Preview but this is an "undo" of ALL previous actions for this table.

Undo
Revert to view of table before the last action was taken. Ctrl+Z will also "Undo".

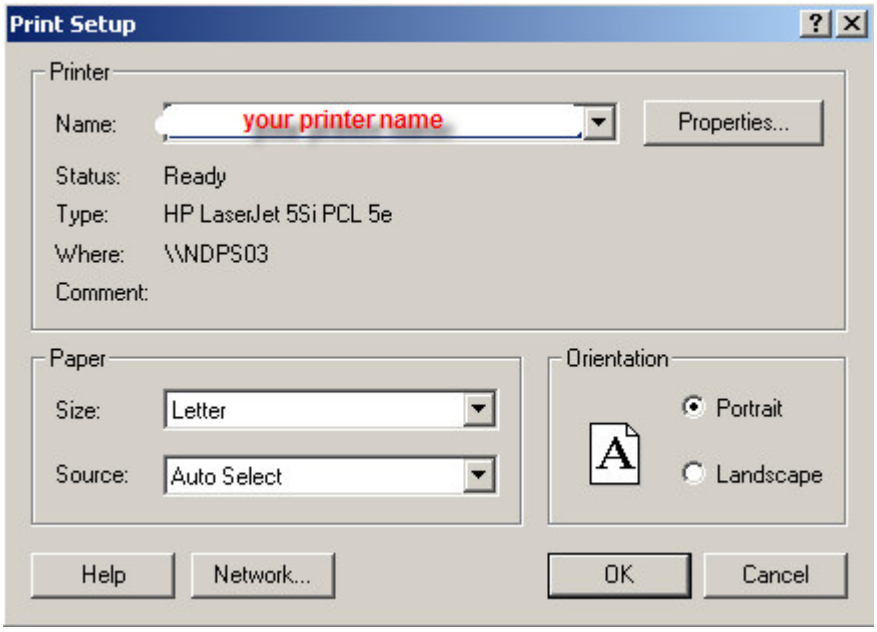
Redo
If an "Undo" was the last action then "Redo" is available.

Goto…
Allows you to move to a particular page, table or area within a table. This option activates the following menu:

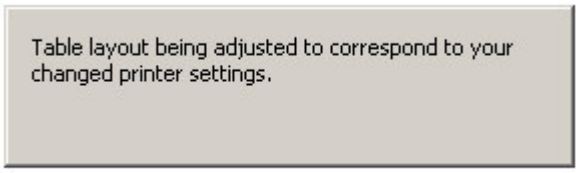
Page option – enter the page number to display.
Table option – choose the table (by name) to display.
Area option – choose the table and the area within that table to display. This option is only available if area processing is used.

Print Setup

This is the usual dialog box for Print Setup. It should be something like the following:



What is important is that this information is used to create the Print Preview of your tables. Changes made here will affect the "preview" of your tables, particulary changes to the paper size and layout. If changes impact the print preview the following message will appear:



Sending Tables to the Printer

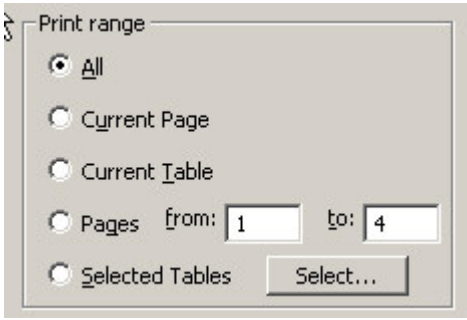
• **To preview the printing of tables**

Click on the tool bar; or from the **File** menu, select **Print Preview**.

• **To print entire tables**

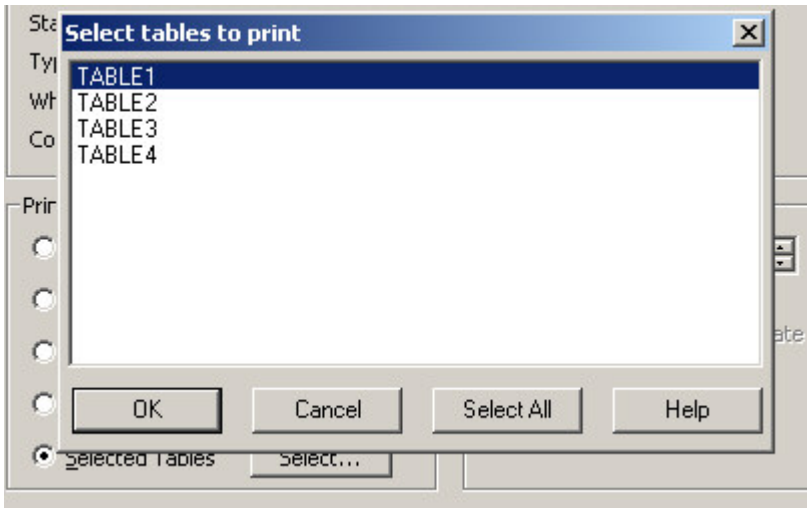
Click on the toolbar; or from the **File** menu, select **Print**; or press **Ctrl+P**.

When the printer menu is displayed the important part is the Print Range section that determines what is to be printed.



The selection here affects what is printed. The options:

- **All** – All pages of all tables.
- **Current Page** – Page currently displayed in window.
- **Current Table** – Table containing the currently displayed page.
- **Pages** – Range of page numbers for printing.
- **Selected Tables** – for this option click the "Select …" button to its right. A secondary menu is presented for table selection.



Other options will be included in the Print menu such as the number of copies to be printed. These options affect how the tables will be printed but not what will be printed.

How To ...

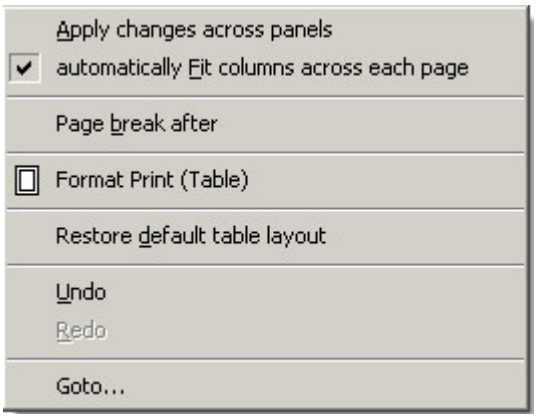
Undo or Reset Changes in Print Preview

Most formatting of tables is done using the various format menus available in the Display or Design View but not available in Print Preview.

The major formatting tools in Print Preview are

- Sizing of Columns or Rows [changing width or height]
- Add Page Breaks

If an action is taken and you want to "undo" it then right click in the Print Preview window. This gives the following menu:



Select "Undo" to reverse a single action.

If you just want to restore the table to its system default format then select "Restore default table layout".

See Also: Print Preview Options

Print Only Selected Tables or Pages

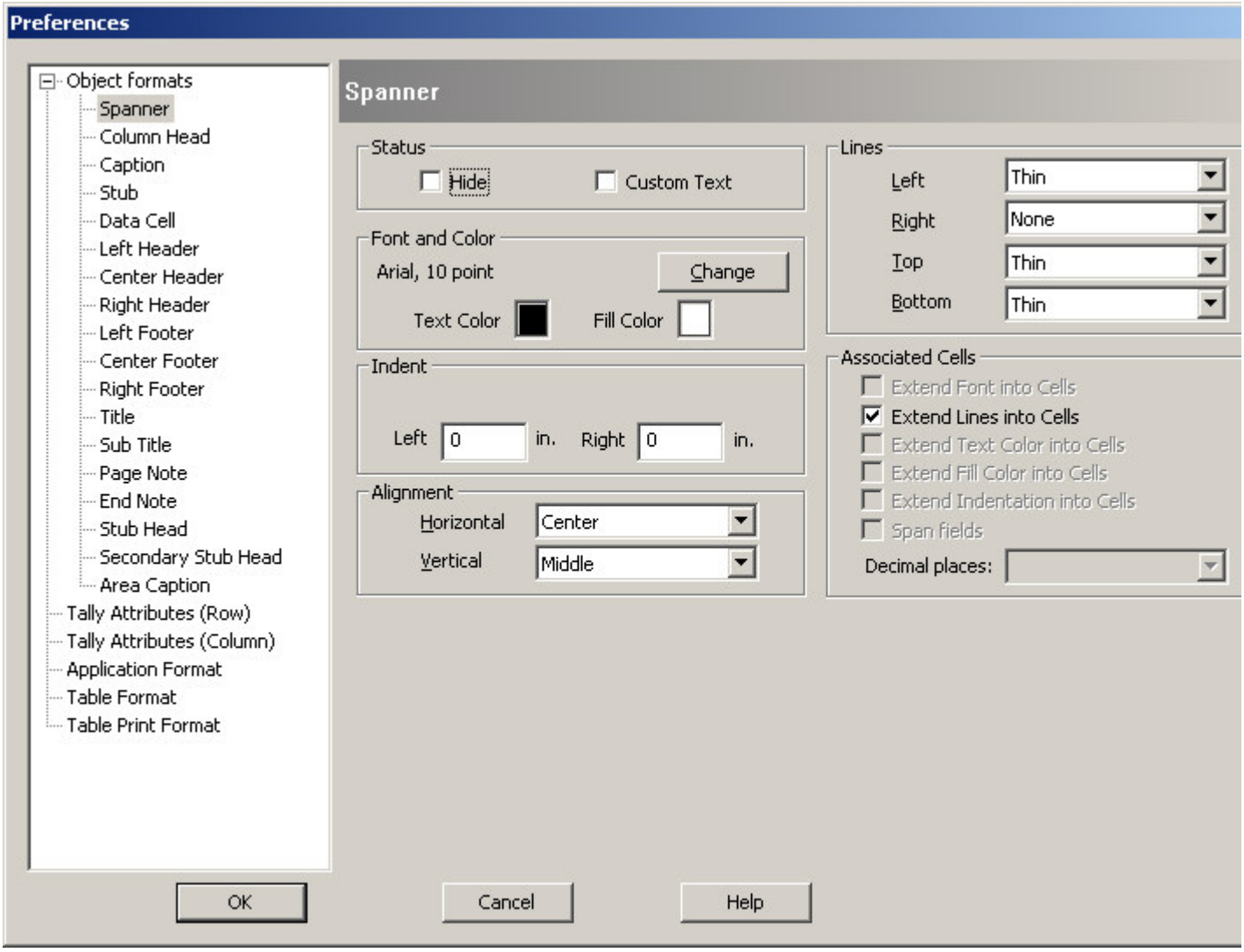
See Sending Tables to the Printer

Tabulation Preferences

Preferences and Default Formats

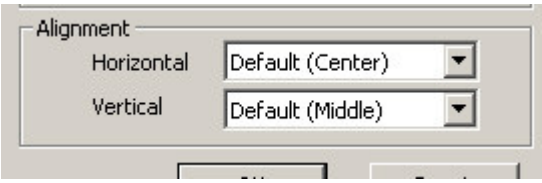
The "Default" settings used in all of the format menus (table element, table, table print, and application) can be set for the entire application. These "Default" settings are then accepted or changed for individual tables. Changing the default setting is the easiest way to change the formatting of all of your tables at once. It allows you to customize the look of all of your all tables.

The "Default" Options menu is available under the Edit drop down menu and listed as "Preferences".



Notice the alignment section in the above menu for Spanner.

Here is the alignment section in the Format (Spanner) menu for a specific table:

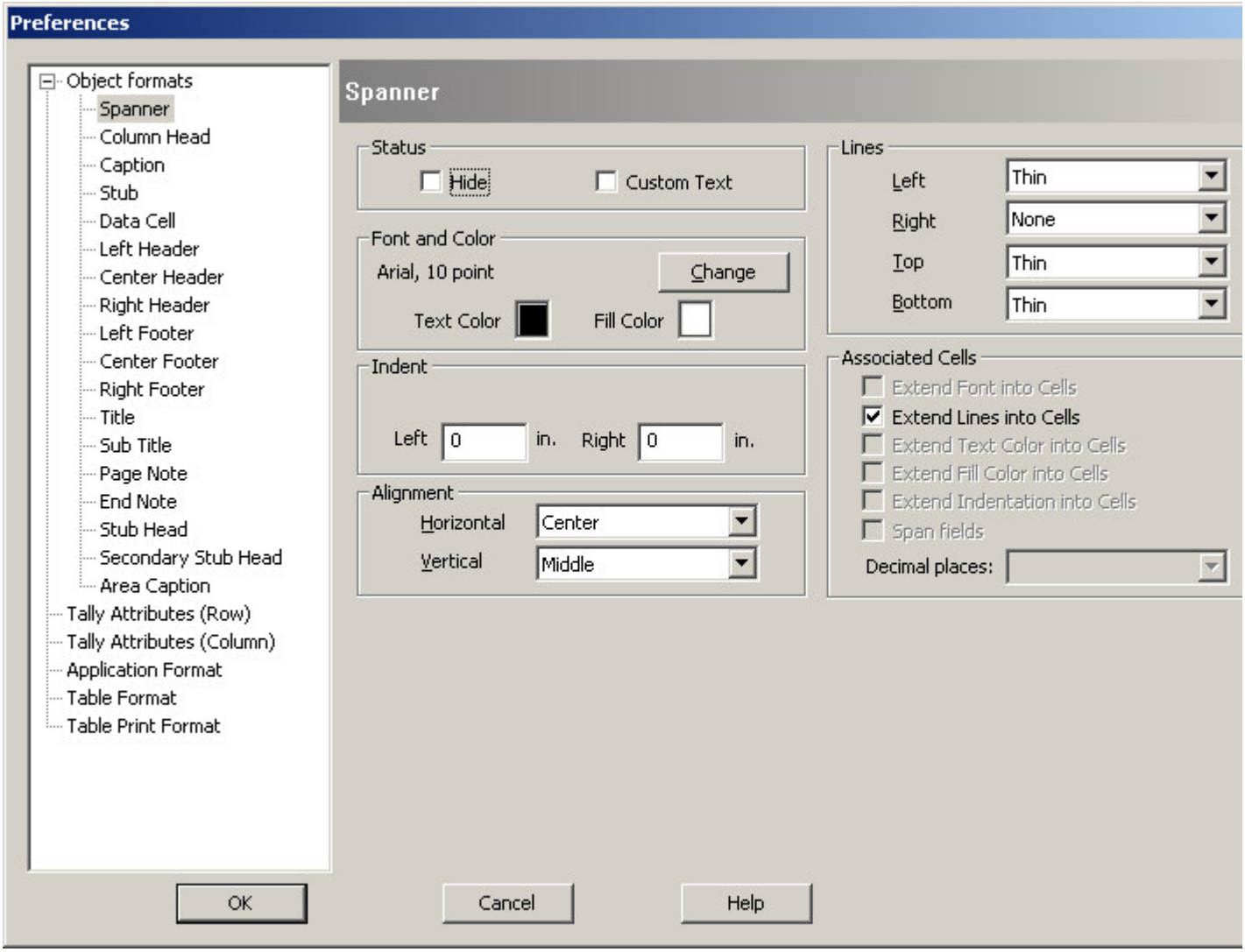


See Also: Modifying Preferences

Modifying Preferences

The option is available under the Edit drop down menu and listed as 'Preferences'.

This brings up the Preferences Window:



Each "Object Format" on the left has a corresponding menu. Select the object in the tree then you can set the 'default' settings for each of the available options. The only difference is that 'Use Default ' check box may be missing since this is the menu that sets those defaults. It is important to note that if a setting is changed here it will be applied to all tables that use the default settings even the tables that have already been defined.

For example: If you select blue as "Text Color" for Spanner then all spanners that exist in tables already defined (and use the default setting) will now be displayed in blue as well as spanners in tables yet to be created.

For the meanings of the various options in each menu see the related topic.

See Also: Tally Attributes for a Variable, Formats for a Part of a Table, Formats for a Table, Formatting Row, Column, or Cell Data , Formats for an Application , Formats for Printing

Loading and Saving Preferences

Every tabulation application starts with the "default preferences" set by CSPro. Users can easily modify these preferences to reflect their own choices of defaults. If you have a standard set of preferences that you want to use for your Tabulation applications, you do not need to make these same changes for each new application. Your settings can be saved after they are modified and then loaded into other applications.

Under the File menu are the two options: **Load Preferences** and **Save Preferences**. Use the **Save** option to create a file that contains your selections. For this option, the user will supply a name for the CSPro Table Format (.tft) file to be created. This file will contain all the "default" settings currently active for this tabulation application.

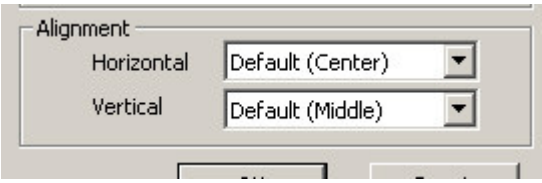
The **Load** option is used to change preferences to those in your CSPro Table Format (.tft) file. For this option, enter the name of the .tft file or browse to find it. Once the file is loaded the revised preferences will be in force.

See also: Modifying Preferences

How To ...

Share the Same Format on Multiple Computers

In most applications the various options in the Format menus use the "Default" settings.



These "Default" settings are contained in Preferences and Default Formats. In any Tabulation application the initial set of Preferences is established by CSPro. They can be reset to other settings through "Load Preferences…" under the File menu.

Sets of user-defined preferences are contained in CSPro Table Format (.tft) files. Once the file has been created by the "Save Preferences…" option under the File menu, it can made available to other users in the same manner that any other file is shared.

See Also: Loading and Saving Preferences

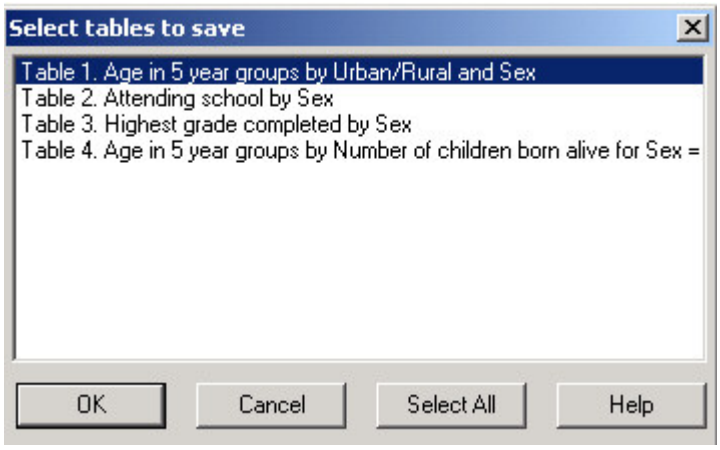
Saving and Copying Table Data

Save Tables for the Table Viewer

Click on the toolbar; or from the **File** menu select **Save Tables**.

If there is only one table in the Tabulation application, you will go directly to the Save As dialog.

If your Tabulation application has multiple tables defined within it, a **Select tables to save** dialog box listing the tables is displayed.



Use the **Select All** button if all tables are needed. Otherwise, select the individual table(s) that you would like to save in a single Table Viewer file. (Multiple tables are selected in the usual manner with the Shift and/or Ctrl keys.)

After selection press **OK**.

In the Save As dialog box enter the name of the Table Viewer file to be created or browse to select the file to be replaced. Table Viewer files must have the .tbw extension.

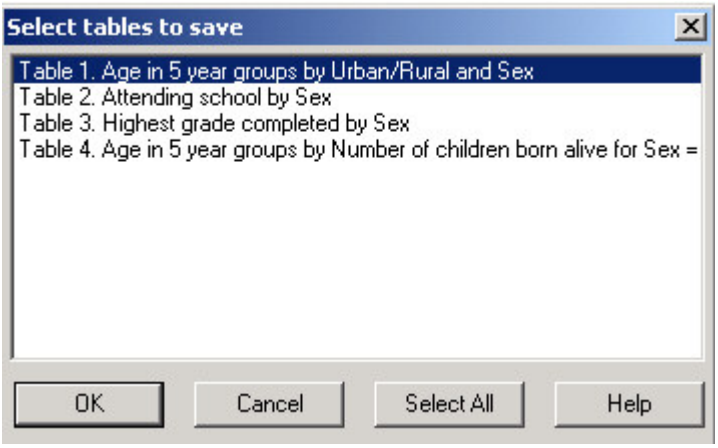
See Also: Saving Tables as Text, HTML and Rich Text Format

Saving Tables as Text, HTML or Rich Text

Click on the toolbar; or from the **File** menu select **Save Tables**.

If there is only one table in the Tabulation application, you will go directly to the Save As dialog.

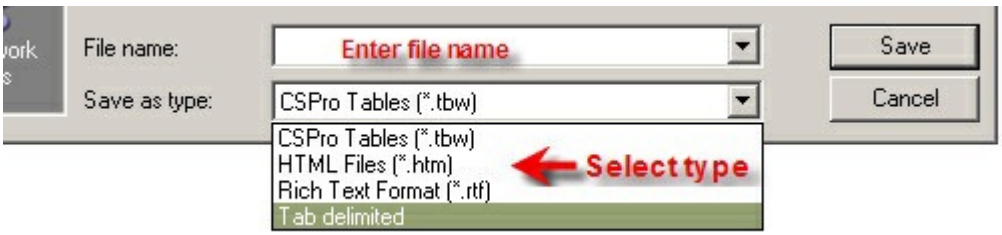
If your Tabulation application has multiple tables defined within it, a **Select tables to save** dialog box listing the tables is displayed.



Use the **Select All** button if all tables are needed. Otherwise, select the individual table(s) that you would like to save in a single Table Viewer file. (Multiple tables are selected in the usual manner with the Shift and/or Ctrl keys.)

After selection press **OK**.

In the "Save As" dialog box use the drop down "Save as type" menu to select the type of file other than .tbw.



Enter the name of the file to be created or browse to select the name of the file to be replaced.

Note: ONLY ONE table at a time can be in Rich Text (.rtf) or HTML (.htm) format. Tab delimited format supports saving multiple tables in a single text file.

Select and Copy Table Data to Other Applications

If you want to copy a table or selected cells (with or without associated text) of a table it must be done in the Display or Design View. There is no selection possible in the **Print Preview**.

• **To select table cells**

- Move the mouse pointer to the upper left-hand corner of the cells you wish to select.

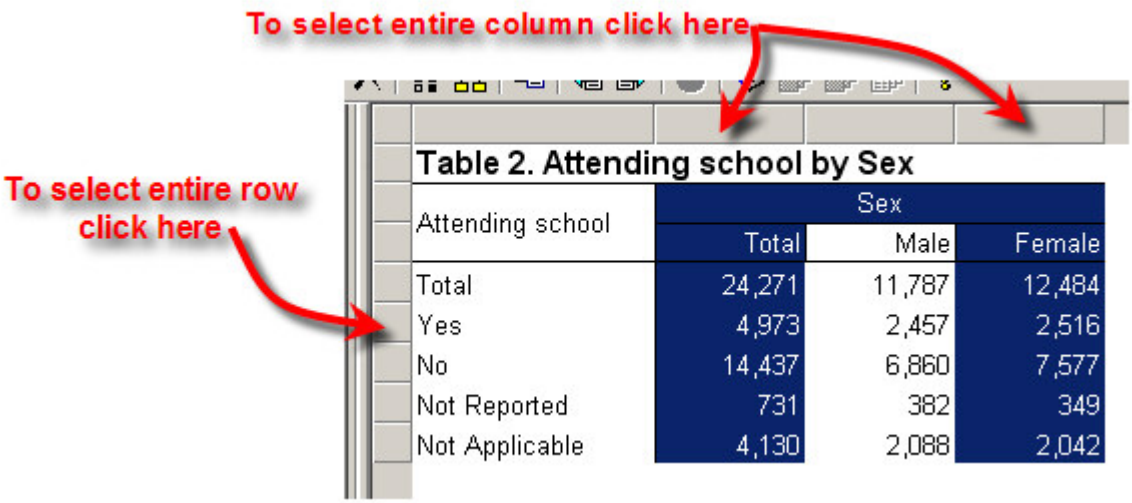
- Press the left mouse button and hold it down while you drag the mouse across the cells you want to select. The cells will change color to indicate that they have been selected. If you drag the mouse outside the cell area (top or bottom, left or right), the table will automatically scroll and continue to select.

- Release the mouse button. The selected cells are highlighted.

- To select additional cells hold the Ctrl Key down and follow the same technique.

• **To select ALL cells from a row or column**

- Entire Rows or columns of cells can be selected by clicking in the appropriate part of the grid.



- Hold the Ctrl key down and click to select non-contiguous rows/column (shown above). To select contiguous rows or columns use the Shift key and select the row or column for the other end of the range.

• **To select ALL cells in a Table**
Choose Select All from the Edit menu; or press Ctrl+A.

• **To deselect cells**
Press the **Esc** key; or from the **Edit** menu, select "**Cancel Selection**".

Now that cells have been selected for copying you have the choice of copying the text associated with the selected cells (table title, stubs, column headers,…) or just copying the data in the selected cells.

- **To copy cell values and associated text**

- Click on the toolbar; or from the **Edit** menu, select **Copy**; or press **Ctrl+C**.

- **To copy cell values only**

- Select **Copy Cells Only** from the **Edit** menu.

Once the cells are copied to the clipboard they can be pasted into a word processor, spreadsheet or any software that accepts tabular or text format.

See also: Saving Tables as Text, HTML or Rich Text

How To ...

Distribute Finished Tables to Other Users

The tables from a tabulation application can be saved in several different file formats. The format you choose depends on what you wish to do with the tables. To include tables in a word processing document, use the rich text format (.rtf). To bring the table data into a spreadsheet so that users can do calculations on it, use the tab-delimited format. To post tables to the web, use the HTML format. If you want your users to be able to use your tables in all of these ways, you can distribute them in the CSPro Table Viewer format (.tbw). Users can then view your tables in the Table Viewer tool and from there they can copy and paste or save tables in any of the formats supported by CSPro. Table Viewer will also allow them to print, and change the formatting of your tables, but it will not allow them to modify the table data.

If the tabulations use area processing, the table viewer files can also be imported into a CSPro Table Retrieval System (TRS). This is a much more structured way to distribute tables, especially for large numbers of tables (or documents, for that matter).

See also: Save Tables for the Table Viewer, Saving Tables as Text, HTML or Rich Text, Introduction to Table Viewer, Introduction to Table Retrieval Setup

Copy Table Data to a Spreadsheet or Word Processor

CSPro allows you to transfer either part of a table, an entire table or multiple tables at once to a word processor or spreadsheet application.

To copy and paste a single table or part of a table follow the procedure outlined in Select and Copy Table Data to Other Applications.

To copy multiple tables, use the one of the file format options given for "Save Tables" (Tab Delimited Text, HTML, or Rich Text). These options are available for single tables or multiple tables. In some cases, you may need to save single tables in individual files to achieve the desired results.

See also: Select and Copy Table Data to Other Applications, Saving Tables as Text, HTML or Rich Text

Prepare Tables for Posting to the Web

In most cases this means saving the tables as HTML files. HTML tables can only created one at a time. So the process must be repeated for each table to be saved in HTML format.

In Design or Display View (not Print Preview) select the Save Tables icon (or equivalent under File menu).

- If only one table is in the set then you will go directly to the Save As menu. If there is more than one table then the "Select tables to save" dialog box is displayed. Select only one from this menu.
- Click **OK** after selection.
- Select HTML Files (.htm) as Save as type.
- Enter the name of the file to be created or browse to select the name of an existing HTML file to be replaced.
- Click Save button. If file already exists you will be asked if you want to replace it.

These files can be viewed in a browser, e.g., Internet Explorer

See Also: Saving Tables as Text, HTML or Rich Text

Table Post Calculation

Introduction to Table Post Calculation

In some cases you may need to do additional processing on tables after all of the tabulation is completed. For example, you may need to calculate ratios or sums using the tabulated data. CSPro allows you to do this by adding program logic in the Postcalc section of the Tally Attributes (Table) dialog. You use program logic, as you would in a batch or data entry application, to access and modify the cells in a table. Tables are treated as two dimensional arrays or matrices. You can use the full range of CSPro arithmetic operations on individual cells, as well as on rows, columns and cell ranges of tables.

This section contains the following information:

Adding Rows and Columns For Post Calculation
Post Calculation For Individual Cells
Post Calculation For Rows, Columns and Ranges
Row and Column Indexing for Post Calculation

Adding Rows and Columns For Post Calculation

In order to use table post calculations to add additional calculations to the tabulated data, you first need a place in the table to put these calculations. This section describes how to add additional rows or columns to a table.

As an example, take adding a third column to the following table in which we will place the male to female ratio.

Table 1. Sex

	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484

There are two ways to add a new column. We can add a new value to the existing value set for the existing variable or we can drag a new variable onto the table next to the existing one. Adding a new value to the value set creates a new column under the existing spanner while adding a new variable creates a new column under a new spanner.

Table 1. Sex

	Sex			
	Total	Male	Female	Male/Female Ratio
Total				

Male/Female Ratio column added as new value to value set for the variable Sex. The new column is under the Sex spanner.

Table 1. Sex

	Sex			Male/Female Ratio
	Total	Male	Female	
Total				

Male/Female Ratio column added by dragging a new variable onto the table. The new column is under a separate spanner.

If you add a new value to the value set, make sure to set the value for the new value set entry to a number that does not exist in your data file, otherwise your totals will be incorrect. In the case of Sex, you can use 3, which should not appear in the edited data.

If you would rather add a new variable to the table to create the additional column, you can add any variable in the dictionary since the numbers in the new column will be overwritten by the postcalc logic. You can either add a new value set to the variable which uses the text you want for the column header (for example "Male/Female Ratio") or you can use an existing value set and customize the column header text once you have placed the variable on the table.

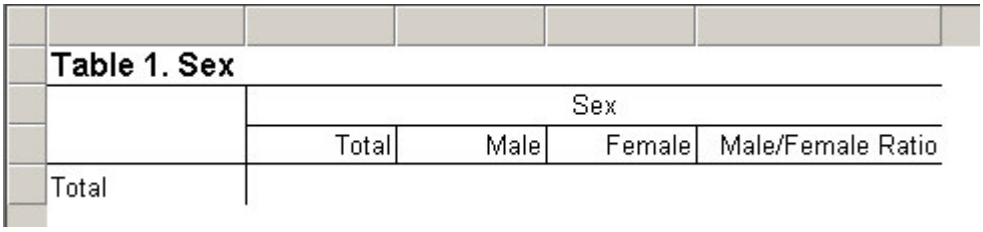
New rows can be added in the same ways, adding a new value to the value set of a variable on the rows or dragging a new variable onto the rows of the table.

See also: Add a Variable to a Tabulation, Implications of Data Dictionary Value Sets

Post Calculation For Individual Cells

CSPro supports modifying individual cells in a table through postcalc logic.

As an example, take adding a third cell to the following table in which we will place the male to female ratio. The male to female ratio is simply the number of males divided by the number of females. We need add a new column to the table and then add postcalc logic to divide the number of males by the number of females and put the result in the new cell.



First, we must add the new column to the table as described in Adding Rows and Columns For Post Calculation.

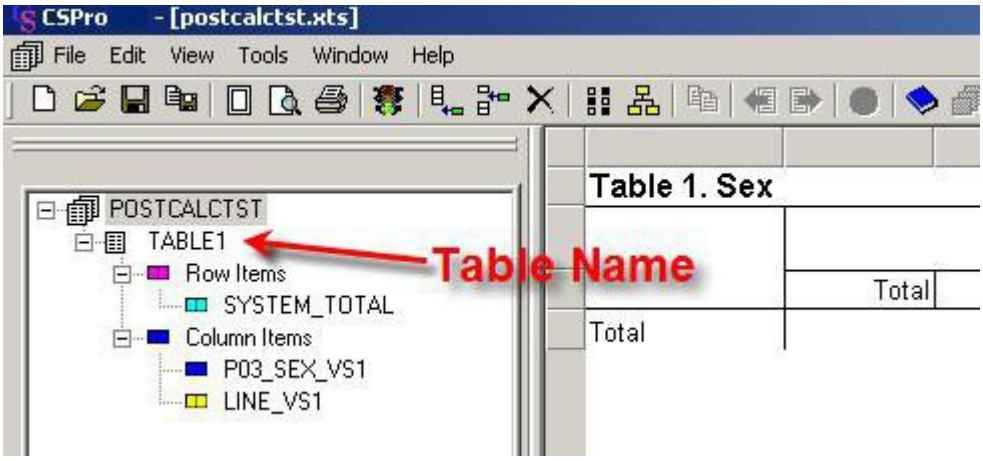
Once the new column has been created, all that is left is to add the postcalc logic. Postcalc logic in CSPro tabulation applications is similar to working with arrays in a CSPro batch edit or data entry application. To access an individual cell in a table, use the table name followed by the indices of the row and column in parentheses:

<table name>(<row>, <column>)

where:

<table name> is the name of the table.

To see the names of the tables, click on the Tables tab in the bottom left of the CSPro window to show the Tables Tree and then select "Names in Trees" from the View menu or press Ctrl+T. This toggles between showing the table titles and the table names in the Tables Tree.



<row> is the row number in the table.

<column> is the column number in the table.

Row and column numbering starts at zero.

In this example, the following logic divides the number of males (row 0, column 1) by the number of females (row 0, column 2) and puts the result in male/female ratio (row 0, column 3):

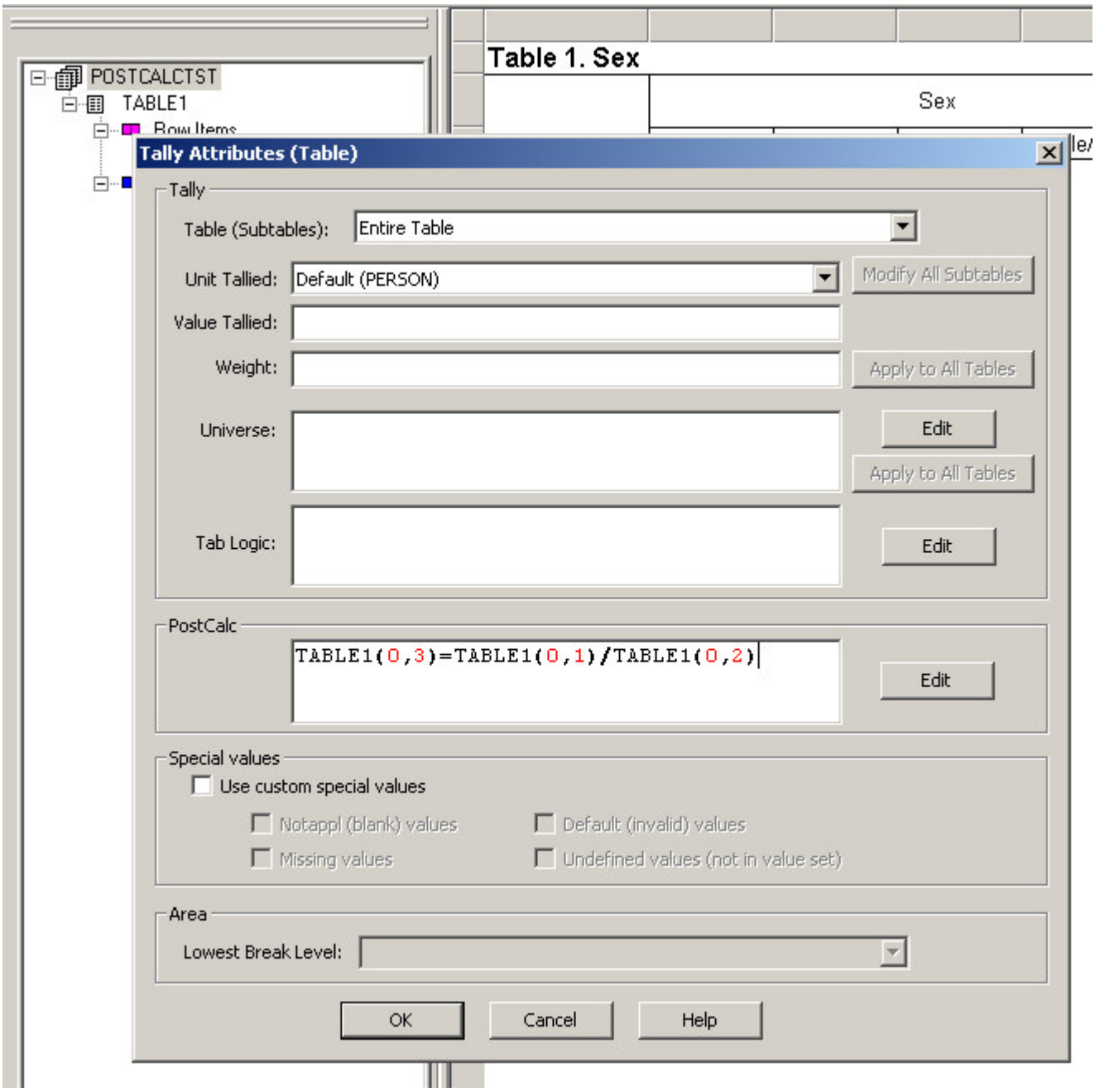
```
TABLE1(0,3) = TABLE1(0,1) / TABLE1(0,2);
```

Note that column and row numbers start at zero, so the first column is column zero, the second column is column is column 1, etc…

Table 1. Sex

	0	1	Sex 2	3
	Total	Male	Female	Male/Female Ratio
Total 0	24,271	11,787	12,484	0.94

To enter postcalc logic for a table, open the Tally Attributes (Table) dialog. You can then either type postcalc logic directly into the postcalc area or click on the "Edit" button next to the postcalc area to bring up a larger window to type in.



Once you have typed in the logic, press OK to dismiss the dialog and then run the table. You may need to change the number of the decimal places in the row, column or cell containing the calculated values.

Table 1. Sex

	Sex			
	Total	Male	Female	Male/Female Ratio
Total	24,271	11,787	12,484	0.94

Postcalc logic can contain numeric constants as well as table values. For example if you wanted men per 100 women rather than male to female ratio, you could use the following logic:

```
TABLE1(0,3) = 100 * TABLE1(0,1) / TABLE1(0,2);
```

This multiplies the result of the division by 100 to give the number of men per 100 women.

Table 1. Sex				
	Sex			
	Total	Male	Female	Males per 100 Females
Total	24,271	11,787	12,484	94.42

Multiple statements may be entered in postcalc, provided that each one is terminated by a semicolon. For example, if you wanted to calculate male to female ratio in column 3 and males per 100 females in column 4 you would write:

```
TABLE1(0,3) = TABLE1(0,1) / TABLE1(0,2);
TABLE1(0,4) = 100 * TABLE1(0,1) / TABLE1(0,2);
```

In fact, postcalc logic can contain nearly any of the statements and functions available in program logic in batch edit and data entry applications.

See also: Post Calculation For Rows, Columns and Ranges, Row and Column Indexing for Post Calculation

Post Calculation For Rows, Columns and Ranges

In addition to performing operations on individual cells, you can perform operations on cell ranges, rows and columns. As an example, take the following table, in which an additional column is added to calculate the male to female ratio for each group.

Table 1. Age in 5 year groups by Sex				
Age in 5 year groups	0	1	Sex 2	3
	Total	Male	Female	Male/Female Ratio
Total	24,271	11,787	12,484	0.94
0 to 4 years	4,130	2,088	2,042	1.02
5 to 9 years	3,337	1,646	1,691	0.97
10 to 14 years	3,250	1,606	1,644	0.98
15 to 19 years	2,967	1,394	1,573	0.89
20 to 24 years	2,243	1,061	1,182	0.90
25 to 29 years	1,686	789	897	0.88
30 to 34 years	1,508	761	747	1.02
35 to 39 years	1,300	600	700	0.86
40 to 44 years	957	474	483	0.98
45 to 49 years	661	323	338	0.96
50 to 54 years	588	303	285	1.06
55 to 59 years	497	225	272	0.83
60 to 64 years	418	194	224	0.87
65 to 69 years	307	141	166	0.85
70 to 74 years	213	90	123	0.73
75 to 79 years	115	51	64	0.80
80 to 84 years	63	31	32	0.97
85 to 89 years	17	8	9	0.89
90 to 94 years	14	2	12	0.17
95 years and over	-	-	-	-

For information on how to add the additional column, see the previous section Adding Rows and Columns For Post Calculation.

The postcalc logic for assigning rows and columns is similar to that of individual cells, however there is a simplified syntax for working with cell ranges, rows and columns. You could set each cell individually with multiple statements as follows:

```
TABLE1(0,3) = TABLE1(0,1) / TABLE1(0,2);
TABLE1(1,3) = TABLE1(1,1) / TABLE1(1,2);
                  …
TABLE1(2,3) = TABLE1(2,1) / TABLE1(2,2);
TABLE1(20,3) = TABLE1(20,1) / TABLE1(20,2);
```

However, that would take a lot of code. Instead, you can specify this one statement using a range or a wild card. To use a range, give the lower and upper limits separated by a colon:

```
TABLE1[0:20,3] = TABLE1[0:20,1] / TABLE1[0:20,2];
```

This means divide the cell in column 1 by the cell in column 2 and put the result in the cell in column 3 for each row from 0 to 20. You can also use a wildcard (an asterisk) to specify an entire row or column:

```
TABLE1[*,3] = TABLE1[*,1] / TABLE1[*,2];
```

Using an asterisk in place of the row index means that the operation applies to all rows in the table. In this case it is the same as specifying the range 0:20 since the table has 21 rows. An asterisk can also be used in place of the column index to specify every column in a table:

```
TABLE1[1,*] = TABLE1[2,*]; { copy row 2 into row 1 }
```

Note that when working with rows, columns and ranges, you must use square brackets "[]" rather than parentheses "( )". Parentheses may only be used when specifying individual cells.

You can only assign ranges or wildcards to each other if the dimensions of the ranges match. For example:

```
TABLE1[0:2, 0:3] = TABLE1[3:5, 0:3];
```

copies one 3 by 4 region of the table to another 3 by 4 region. However, the following code will fail since it attempts to copy a 3 by 4 region to a 3 by 3 region:

```
TABLE1[0:2, 0:2]= TABLE1[3:5, 0:3]; { This does not work ! }
```

See also: Post Calculation For Individual Cells, Row and Column Indexing for Post Calculation

Row and Column Indexing for Post Calculation

When determining the indices to use for particular rows and columns, always remember that the first row or column is at index zero and that caption rows (rows with no data) should be skipped. Also note that hidden data rows and columns should be counted even if they are not visible in the table. Always make sure that view hidden parts is turned on when determining row and column indices.

Table 2. Sex and Literacy by Urban/Rural for Age 15 and Above

		Urban/Rural		
		Total	Urban	Rural
	Sex			
	Total			
	Literacy			
0	Total	13,554	8,025	5,529
1	Literate	11,313	7,179	4,134
2	Illiterate	2,241	846	1,395
3	Percent Literate	83.5	89.5	74.8
	Male			
	Literacy			
4	Total	6,447	3,749	2,698
5	Literate	5,567	3,424	2,143
6	Illiterate	880	325	555
7	Percent Literate	86.4	91.3	79.4
	Female			
	Literacy			
8	Total	7,107	4,276	2,831
9	Literate	5,746	3,755	1,991
10	Illiterate	1,361	521	840
11	Percent Literate	80.8	87.8	70.3

For example, in the table above, in order to calculate the percentage of people in each sex category (Male, Female, Total), we set the values in rows 3, 7 and 9. The caption rows, "Sex", "Total", "Literacy", "Male", and "Female" are not counted, but the hidden rows, the three "Illiterate" rows, are counted. The postcalc logic follows:

```
TABLE2[3,*] = 100 * TABLE2[1,*]/TABLE2[0,*]; {% literate Total }
TABLE2[7,*] = 100 * TABLE2[5,*]/TABLE2[4,*]; {% literate Male }
TABLE2[11,*] = 100 * TABLE2[9,*]/TABLE2[8,*]; {% literate Female}
```

When a table includes percents, the row and column indices are a bit more complicated. Although the percent rows/columns are interleaved with the rows/columns for the counts, the percent rows/columns are numbered after the counts. In other words the index for the first percent row or column in a subtable always starts after the last count row or column. For example, in the table below, the rows for the values of marital status under the male caption are counted one after another as indices 0 through 4 and are then followed by the percent rows for marital status as indices 5 through 9. The first percent row is at index 5 which follows the last count row (Never Married) at index 4.

Table 3. Sex and Marital status by Urban/Rural for Age 15 and Above

		Urban/Rural		
		Total	Urban	Rural
	Male			
0	Married	3,434	1,888	1,546
5	Percent	25.3	23.5	28.0
1	Divorced	199	86	113
6	Percent	1.5	1.1	2.0
2	Separated	-	-	-
7	Percent	-	-	-
3	Widowed	148	68	80
8	Percent	1.1	0.8	1.4
4	Never Married	2,666	1,707	959
9	Percent	19.7	21.3	17.3
	Female			
10	Married	3,539	1,963	1,576
15	Percent	26.1	24.5	28.5
11	Divorced	599	311	288
16	Percent	4.4	3.9	5.2
12	Separated	-	-	-
17	Percent	-	-	-
13	Widowed	638	354	284
18	Percent	4.7	4.4	5.1
14	Never Married	2,331	1,648	683
19	Percent	17.2	20.5	12.4
20	Unmarried men per 100 unmarried women	84.4	80.5	91.8

In this example, to calculate the ratio of unmarried men per 100 unmarried women, we need to divide the number of unmarried men (divorced men + separated men + widowed men + never married men) by the number of unmarried women (divorced women + separated women + widowed women + never married women) and multiply the result by 100. The postcalc logic is therefore:

```
TABLE3[20,*] =
100 *
(TABLE3[1,*] + TABLE3[2,*] + TABLE3[3,*] + TABLE3[4,*])/
(TABLE3[11,*] + TABLE3[12,*] + TABLE3[13,*] + TABLE3[14,*]);
```

See also: Post Calculation For Individual Cells, Post Calculation For Rows, Columns and Ranges

Run Production Tabulations

Introduction to Production Tabulations

Production applications can be run either interactively from CSPro or using batch (*.BAT) files to execute one CSPro program after another. For small surveys all the tabulation work can usually be done within CSPro. For a population census or large survey, you may want to create batch files to perform processing.

When you run tabulation from CSPro, you can produce the entire tabulation in one run or you can tabulate in parts, that is one step, or process, at a time. The entire tabulation process consists of three processes:

Tabulate – One or more data files are processed to produce table matrices, row and columns of tabulated numbers.

Consolidate – Tables matrices are added together for different lower geographic areas, say districts, to produce tables for higher geographic areas such as provinces and country. This process is performed only when area specifications are given.

Format – Table matrices are surrounded with titles, headings, and stubs to be displayed or printed for the end user.

You can have CSPro Run in Parts to perform each of these processes one at a time.

Run All in Batch

You can run a complete tabulation from a batch program by executing **CSTab.exe** and using a PFF file as the command line parameter. For example, if your PFF file name is "MyTabs.pff", you can launch CSTab by:

```
Start /wait "C:\Program Files (x86)\CSPro 7.7\CSTab.exe" MyTabs.pff
```

This launches the program CSTab.exe to run with the parameters specified in the PFF file MyTabs.pff. Note that using Start /wait is not strictly necessary; it simply ensures that the command does not terminate until CSTab.exe has finished processing. This is useful when there are other commands that follow which depend on CSTab completing before they can be executed.

You can create a PFF file in two ways:

- Run the tabulation from CSPro. It will save the *.PFF file it generates in the same folder with your tabulation application. The *.PFF will have the same name as the application with .PFF appended. Rename and modify this file with a text editor (such as Notepad or Wordpad).
- Create a new *.PFF file using a text editor.

The following shows an example of a tabulation PFF file. Note that a PFF file is not case sensitive. You can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Tabulation
Operation=All

[Files]
Application=.\MyTabs.xtb
InputData=.\MyData.dat
Listing=.\MyTabs.xtb.lst
AreaNames=.\MyAreaNames.anm
OutputTBW=.\MyTables.xtb.tbw
```

```
[Parameters]
ViewListing=OnError
ViewResults=Yes
```

The **[Run Information]** block is required and must appear exactly as shown in the example above.

The **[Files]** block is required and defines all files used in the tabulation run. A description of the files is as follows:

- **Application =** the tabulation edit application you created
- **InputData =** the data file to be tabulated -- If there is more than one input data file, insert multiple InputData lines.
- **Listing =** a report of the tabulation processing
- **AreaNames =** the areanames file used only if there is area processing
- **OutputTBW =** the output formatted tables

If any required files are not coded or are missing, the file association dialog box will be displayed allowing you to fill in or change the missing file names.

The **[Parameters]** block is optional. If allows to specify additional aspects of the tabulation run.

- **ViewListing =** specifies how the tabulation run listing is displayed. If ViewListing is missing, Always is assumed.

  Always - the listing is always displayed
  OnError - the listing is displayed only when an error or an invalid subscript warning occurred
  Never - the listing is never displayed

- **ViewResults =** specifies whether or not the formatted tables file (*.TBW) is displayed in TableViewer at the end of the run. If ViewResults is missing, Yes is assumed.

  Yes - the tables are displayed
  No - the tables are not displayed

Run in Parts

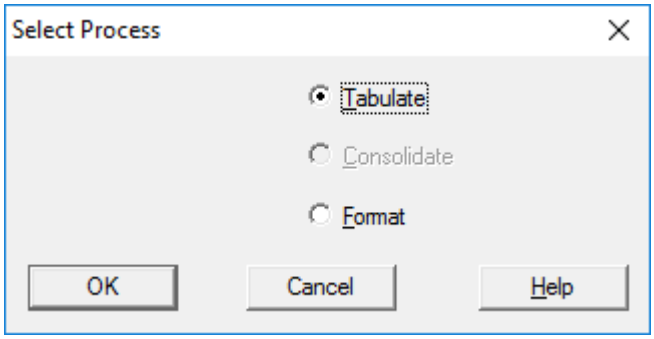
Introduction to Run in Parts

When CSPro runs a tabulation application, it automatically performs two or three processes. These processes are to:

- **Tabulate** the data file;
- **Consolidate** geographic areas (if the application uses area processing); and
- **Format** the tables with text.

Run in parts allows you perform each of these processes separately. Run in parts is also used when you need to save the intermediate .tab files for later use.

To run in parts interactively, from the **File** menu, select **Run in Parts**. Then select the process you want to run. (The **Consolidate** process will be grayed out if you are not using area processing in your application.)

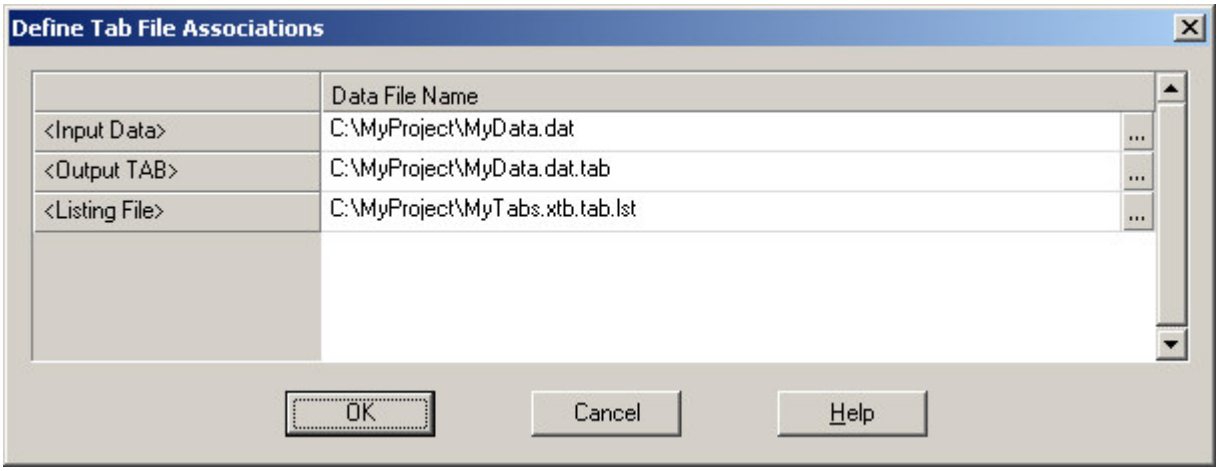


See also: Run Tabulate Interactively, Run Consolidate Interactively, Run Format Interactively, Run Tabulate in Batch, Run Format in Batch, Run Consolidate in Batch

Run Tabulate Interactively

The Tabulate process reads the input data file(s), runs the tabulation application, and produces a set of table matrices that are stored in a tables file (*.TAB).

When the Tabulate process is selected, the following files are requested:



Input Data: The data file(s) being tabulated. There is NO required extension for CSPro data files. Multiple input data files

can be selected using the browse button.

Output TAB: The table matrix file created during tabulation. The *.TAB extension is required. The name should be associated with the corresponding data file especially if multiple data files are tabulated using the same application.

Listing File: The diary-type report generated by the tabulation run showing record and case counts and any problems encountered.

This is a sample of a listing file where no problems were encountered during tabulation:

```
CSPRO Process Summary
+-----------------------------------------------------+
| 29143 Records Read ( 100% of input file) |
| 0 Ignored ( 0 unknown, 0 erased) |
| 0 Messages ( 0 U, 0 W, 0 E) |
+---------+--------------+--------------+-------------+
| Level | Input Case | Bad Struct | Level Post |
+---------+--------------+--------------+-------------+
| 1 | 4872 | 0 | 4872 |
+---------+--------------+--------------+-------------+

Process Messages

CSPRO Executor Normal End
-----------------------------------------------------
```

See also: Run Consolidate Interactively, Run Format Interactively

Run Tabulate in Batch

You can run the tabulate process from a batch program by executing **CSTab.exe** and using a PFF file as the command line parameter. For example, if your PFF file name is "MyTabs.pff", you can launch CSTab by:

```
Start /wait "C:\Program Files (x86)\CSPro 7.7\CSTab.exe" MyTabs.tab.pff
```

This launches the program CSTab.exe to run with the parameters specified in the PFF file MyTabs.pff. Note that using Start /wait is not strictly necessary; it simply ensures that the command does not terminate until CSTab.exe has finished processing. This is useful when there are other commands that follow which depend on CSTab completing before they can be executed.

You can create a PFF file in two ways:

- Run the tabulation from CSPro. It will save the *.PFF file it generates in the same folder with your tabulation application. The *.PFF will have the same name as the application with .PFF appended. Rename and modify this file with a text editor (such as Notepad or Wordpad).
- Create a new *.PFF file using a text editor.

The following shows an example a PFF file for the tabulation process. Note that a PFF file is not case sensitive. You can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Tabulation
Operation=Tab

[Files]
Application=.\MyTabs.xtb
InputData=.\MyData.dat
Listing=.\MyTabs.xtb.tab.lst
```

```
TabOutputTAB=.\MyData.dat.tab

[Parameters]
ViewListing=OnError
```

The **[Run Information]** block is required and must appear exactly as shown in the example above.

The **[Files]** block is required and defines all files used in the tabulation run. A description of the files is as follows:

- **Application =** the tabulation application you created
- **InputData =** the data file to be tabulated – If there is more than one input data file, insert multiple InputData lines.
- **Listing =** a report of the tabulate process
- **TabOutputTAB =** the output table matrices file

If any required files are not coded or are missing, the file association dialog box will be displayed allowing you to fill in or change the missing file names.

The **[Parameters]** block is optional. If allows to specify additional aspects of the tabulation run.

- **ViewListing =** specifies how the tabulation run listing is displayed. If ViewListing is missing, Always is assumed.

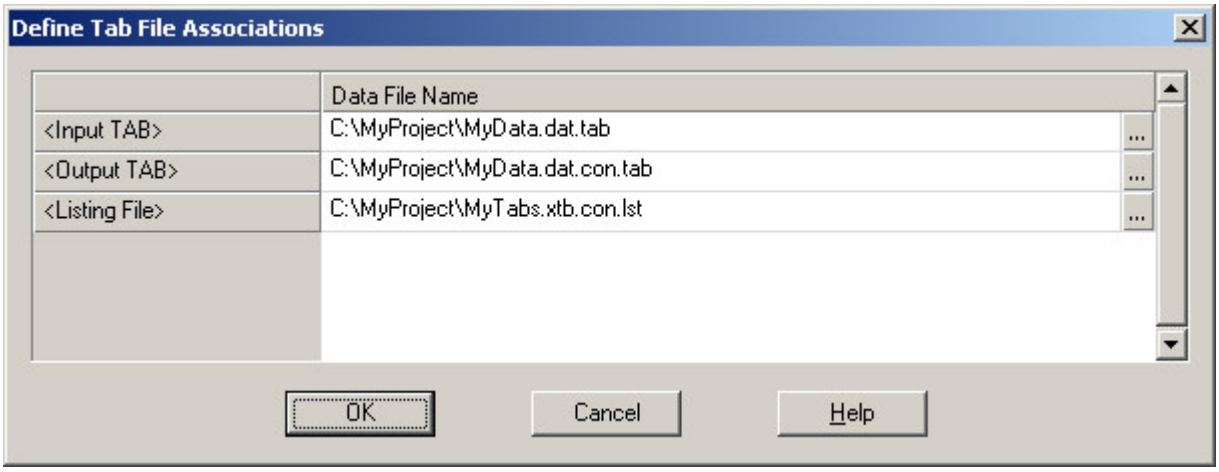
  Always - the listing is always displayed
  OnError - the listing is displayed only when an error or an invalid subscript warning occurred
  Never - the listing is never displayed

See also: Run Consolidate in Batch, Run Format in Batch

Run Consolidate Interactively

The Consolidation process reads the table matrix file(s) output from Tabulate process, and produces another table matrix file (*.TAB) based on consolidation specifications given in the tabulation application.

When the Consolidate process is selected, the following files are requested:



Input TAB: The tabulated table matrices file(s) to be consolidated. Multiple input *.TAB files can be selected using the browse button. The *.TAB extension is required.

Output TAB: The consolidated table matrices file created [or replaced] from the input TAB file(s). This TAB name must be different from the input name and the extension is required.

Listing File: The report generated for the run showing area codes found in the data and tables available for those codes. Blank 'codes' indicate summary levels. The .LST extension is not required.

This is a sample of a listing file for a Consolidation process:

```
CSPro Process Summary
+----------------------------------------------+
| 1568 Slices Read ( 100% of Input file) |
+----------------------------------------------+


Process Messages

PROVINCE DISTRICT Table Names
-------- -------- -----------
TABLE1 TABLE2 TABLE3 TABLE4
1 TABLE1 TABLE2 TABLE3 TABLE4
1 1 TABLE1 TABLE2 TABLE3 TABLE4
1 2 TABLE1 TABLE2 TABLE3 TABLE4
1 3 TABLE1 TABLE2 TABLE3 TABLE4
1 4 TABLE1 TABLE2 TABLE3 TABLE4
1 5 TABLE1 TABLE2 TABLE3 TABLE4
  . . .
```

Tables 1, 2, 3 and 4 are available for the following geographies:

Total [Country] indicated by Province and District = blank
Province 1 indicated by Province = 1 and District = blank
Province 1 and District 1 indicated by Province = 1 and District = 1
Etc.

See also: Run Tabulate Interactively, Run Format Interactively

Run Consolidate in Batch

You can run the consolidation process from a batch program by executing **CSTab.exe** and using a PFF file as the command line parameter. For example, if your PFF file name is "MyTabs.pff", you can launch CSTab by:

```
Start /wait "C:\Program Files (x86)\CSPro 7.7\CSTab.exe" MyTabs.con.pff
```

This launches the program CSTab.exe to run with the parameters specified in the PFF file MyTabs.pff. Note that using Start /wait is not strictly necessary; it simply ensures that the command does not terminate until CSTab.exe has finished processing. This is useful when there are other commands that follow which depend on CSTab completing before they can be executed.

You can create a PFF file in two ways:

- Run the tabulation from CSPro. It will save the *.PFF file it generates in the same folder with your tabulation application. The *.PFF will have the same name as the application with .PFF appended. Rename and modify this file with a text editor (such as Notepad or Wordpad).
- Create a new *.PFF file using a text editor.

The following shows an example of a PFF file for the consolidation process. Note that a PFF file is not case sensitive. You can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Tabulation
Operation=Con

[Files]
```

```
Application=.\MyTabs.xtb
ConInputTAB=.\MyData.dat.tab
Listing=.\MyTabs.xtb.con.lst
ConOutputTAB=.\MyData.dat.con.tab

[Parameters]
ViewListing=OnError
```

The **[Run Information]** block is required and must appear exactly as shown in the example above.

The **[Files]** block is required and defines all files used in the tabulation run. A description of the files is as follows:

- **Application =** the tabulation application you created
- **ConInputTAB =** the input table matrices file – These are the output from the tabulate process. If there are multiple input table matrices files, insert multiple ConInputTab lines.
- **ConOutputTAB =** the output table matrices file
- **Listing =** a report of the consolidate process

If any required files are not coded or are missing, the file association dialog box will be displayed allowing you to fill in or change the missing file names.

The **[Parameters]** block is optional. If allows to specify additional aspects of the tabulation run.

- **ViewListing =** specifies how the tabulation run listing is displayed. If ViewListing is missing, Always is assumed.

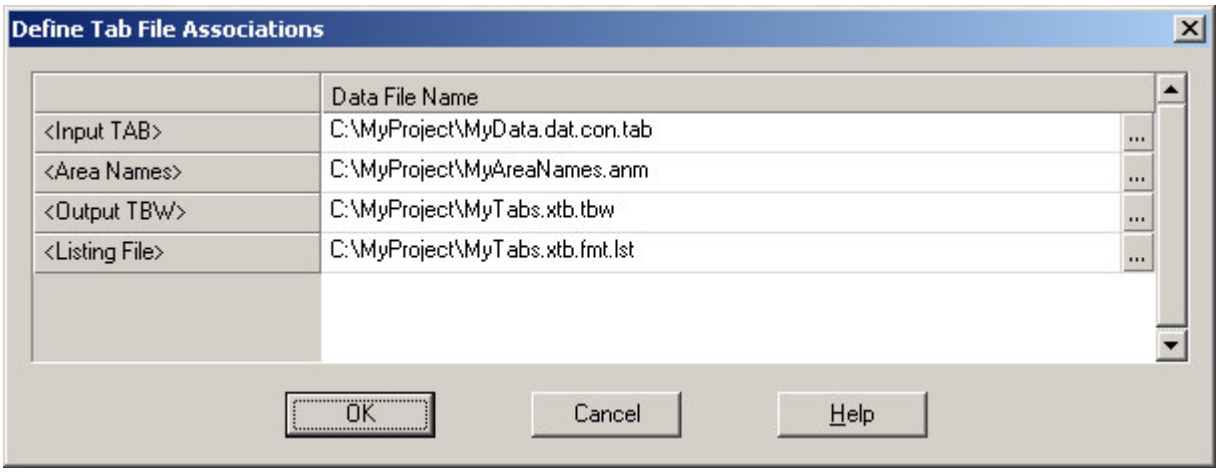
  Always - the listing is always displayed
  OnError - the listing is displayed only when an error occurred
  Never - the listing is never displayed

See also: Run Tabulate in Batch, Run Format in Batch

Run Format Interactively

The Format process reads the table matrix file(s) output from Tabulate or Consolidations process, and produces tables files (*.TBW) based on format specifications given in the tabulation application.

When the Format process is selected, the following files are requested:



Input TAB: The tabulated or consolidated matrices file. If application does not have area processing, it was created in the Tabulate process, if it has area processing was performed, it was created in the Consolidate process. The .TAB extension is required.

Area Names: Only used for applications with area processing. An Area Names File is used to associate the areas

codes in the TAB file with their descriptive text. The .ANM extension is required.

Output TBW: The CSPro Table Viewer file created by merging the matrices from the input TAB file, the table format specifications from the tabulations application, and the optional area names. The .TBW extension is required.

Listing File: The report of the Format process.

This is a sample of a listing file for a Format process:

```
CSPRO Process Summary
+-----------------------------------------------------+
| 8 Slices Read ( 100% of input file) |
| 0 Ignored ( 0 unknown, 0 erased) |
| 0 Messages ( 0 U, 0 W, 0 E) |
+-----------------------------------------------------+

Process Messages

CSPRO Executor Normal End
-----------------------------------------------------------------------
```

See also: Run Tabulate Interactively, Run Consolidate Interactively, Introduction To Table Viewer

Run Format in Batch

You can run the format process from a batch program by executing **CSTab.exe** and using a PFF file as the command line parameter. For example, if your PFF file name is "MyTabs.pff", you can launch CSTab by:

```
Start /wait "C:\Program Files (x86)\CSPro 7.7\CSTab.exe" MyTabs.fmt.pff
```

This launches the program CSTab.exe to run with the parameters specified in the PFF file MyTabs.pff. Note that using Start /wait is not strictly necessary; it simply ensures that the command does not terminate until CSTab.exe has finished processing. This is useful when there are other commands that follow which depend on CSTab completing before they can be executed.

You can create a PFF file in two ways:

- Run the tabulation from CSPro. It will save the *.PFF file it generates in the same folder with your tabulation application. The *.PFF will have the same name as the application with .PFF appended. Rename and modify this file with a text editor (such as Notepad or Wordpad).
- Create a new *.PFF file using a text editor.

The following shows an example a PFF file for the format process. Note that a PFF file is not case sensitive. You can use any combination of upper and lower case text.

```
[Run Information]
Version=CSPro 7.7
AppType=Tabulation
Operation=Format

[Files]
Application=.\MyTabs.xtb
FormatInputTAB=.\MyData.dat.con.tab
AreaNames=.\MyAreaNames.anm
Listing=.\MyTabs.xtb.fmt.lst
OutputTBW=.\MyTables.xtb.tbw

[Parameters]
ViewListing=OnError
ViewResults=No
```

The **[Run Information]** block is required and must appear exactly as shown in the example above.

The **[Files]** block is required and defines all files used in the tabulation run. A description of the files is as follows:

- **Application =** the tabulation application you created
- **FormatInputTAB =** the input table matrices file – This is output from the tabulate or consolidate process.
- **Listing =** a report of the tabulation processing
- **AreaNames =** the area names file, only if there is area processing
- **OutputTBW =** the output formatted tables

If any required files are not coded or are missing, the file association dialog box will be displayed allowing you to fill in or change the missing file names.

The **[Parameters]** block is optional. If allows to specify additional aspects of the tabulation run.

- **ViewListing =** specifies how the tabulation run listing is displayed. If ViewListing is missing, Always is assumed.

  Always - the listing is always displayed
  OnError - the listing is displayed only when an error occurred
  Never - the listing is never displayed

- **ViewResults =** specifies whether or not the formatted tables file (*.TBW) is displayed in TableViewer at the end of the run. If ViewResults is missing, Yes is assumed.

  Yes - the tables are displayed
  No - the tables are not displayed

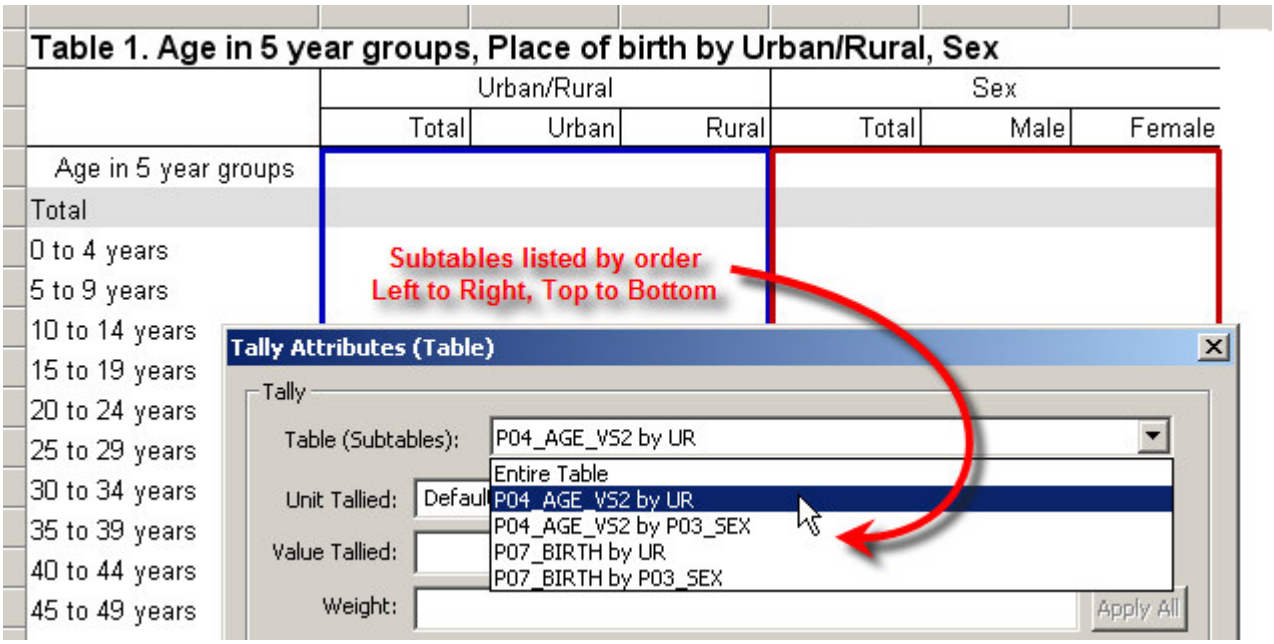
See also: Run Tabulate in Batch, Run Consolidate in Batch

Advanced Table Topics

Using Subtables

Subtables are separate and individual tabulations ("sub") contained within the definition of a single table. Subtables can be thought of as merged or concatenated to create the whole table.

Each subtable, being independent, has its own set of Tally Attributes as well as the Tally Attributes for the "Entire Table". To access the attributes of a subtable either right click while the cursor is in the area of the subtable and then select Tally Attributes (Subtable) or right click anywhere in the table, select Tally Attributes (Table), and then select the desired subtable from the drop down menu as shown below.



In the menu subtables are listed as they appear in the "big" table, from left to right then from top to bottom. The "Names" of the subtables are created from the value sets present. If an item has only one value set then the name of the item itself appears, e.g., P03_SEX in example. If an item has more than one value set then the name of the value set appears, e.g., P04_AGE_VS2 in the example.

Any attributes entered in the "Entire Table" menu except "Unit Tallied" are also incorporated into the subtables. Entries in this menu apply to all subtables in the table.

In the Tally Attributes (Table)

- If an item or value is given for "Value Tallied" or "Weight" it is also MULTIPLIED by any corresponding value give in any subtable.
- Any entry in the "Universe" box is combined with any universe in a subtable using "and". In other words, both universe criteria must be met in order for a tally to be made.
- Of course, any checked "Special values" or "Lowest Break Level" (applicable only for area processing) always refer to the entire table including any subtables.

With the exception of attributes applying to entire table, individual subtables can each be assigned different weights, universes, etc.

See Also: Create Tabulations with Multiple Variables, Tally Attributes for a Table

Changing the Unit of Tabulation

This attribute is available on the Tally Attribute (Table/Subtable) menu.

"Unit Tallied" allows you to change the unit of computation for the table or subtable. The unit of computation is the level, record or item in the dictionary that is counted for the tally. It is basically a "looping" unit for the tally, i.e., for each level, record or item do the tally.

For example, when tallying categories of the rent variable on the housing record, by default, the unit would be the housing record so each household would be tallied once in the appropriate category. However, if the "unit of tally" were the person record, this would tally in the household's rent category once for each person in the case.

For example, a household paying $850/month rent would be in the $800 to $899 category. If the "unit of tally" were the household record then the tally would be done once for this case. If the "unit of tally" were the person record and there are 4 persons in the household, then a tally is done for each person in the household resulting in 4 tallies for this case (instead of one).

Note: In most cases, the unit is left as the default assigned by CSPro.

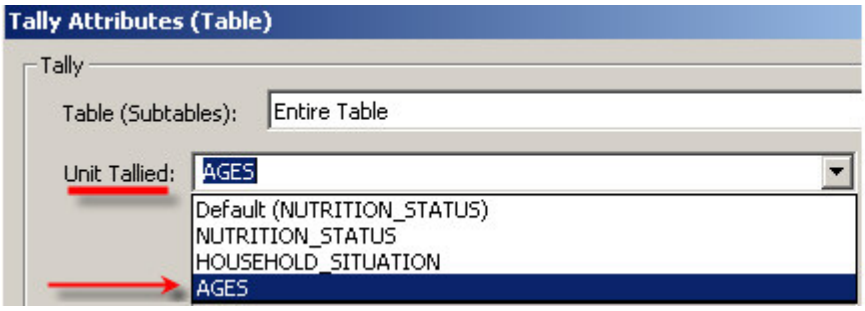
Basic rules of "units" (these refer to both tables or subtables):

- If table has all items from one record then the "unit of tally" is the record.
- If table has some items for a non-repeating record, e.g., house, and a repeating record, e.g., person, the "unit of tally" is the repeating record (person).
- If table has items from different non-repeating records then the "unit of tally" can be either record.
- If a non-repeating record contains an item with occurrences then the repeating item can also be the "unit of tally."
- If table has only ID items then the "unit of tally" can be the case (Quest) or any repeating record or repeating item.
- No table can contain items from different repeating records unless there is a relation defined between the records in the data dictionary (see Relation Description). For relations, the "unit of tally" must be the name of the link.

(Non-repeating means MAXimum records is one)

Note: If a table/subtable contains items from a non-repeating record and another record, there is no tally if the non-repeating record is not present for the case. For example, if the case is missing its House record then no tallies involving items from the House will be done for this case.

For example, the table has an item from the Household_Situation record and one from the Nutrition_Status record, both of which are non-repeating. AGES is an item with occurrences on the Household_Situation record. The available "Unit Tallied" are:



Nutrition_Status and Household_Situation are non-repeating records; Ages is item occurring on Household_Situation record

If the "unit of tally" is either non-repeating record then only tally is made per case. If the "unit of tally" is the occurring item AGES, then a tally is made for each occurrence of age.

See Also: Tabulations Using Relations

Table Logic (tablogic)

Table logic (or tablogic) allows you to add CSPro logic that is executed during tabulation. This is mainly used to add your own "recoded" variables to tables instead of using existing variables from your dictionary. This is useful when the categories you want in your table must be tallied based on the values of more than one variable from the dictionary, in other words, when the categories you want must be computed based on the values of multiple variables. In such a case, you can create a new variable with the categories you want and write logic to set its value based on the values of existing variables.

For example, if you want to tabulate houses that have "complete plumbing," meaning that they have piped water, bathing facilities and a toilet inside the housing unit. The categories for "complete plumbing" are:

- Complete – piped water inside the unit, and a private toilet inside the unit and bathing facilities inside the unit.
- Some but not all –one or more of the three conditions above, but not all three.
- None – none of the above conditions.

The variable "complete plumbing" does not exist in the main dictionary. It can be however, be determined based on the values of the following three variables that are in the dictionary:

N	Value Set Label	Value Set Name	Value Label	From
☐	Source of water	H10_WATER_VS1		
☐			Inside piped	1
☐			Outside piped	2
☐			Bottled or canned	3
☐			Closed well or close	4
☐			Open well or spring	5
☐			River/Stream/Other	6

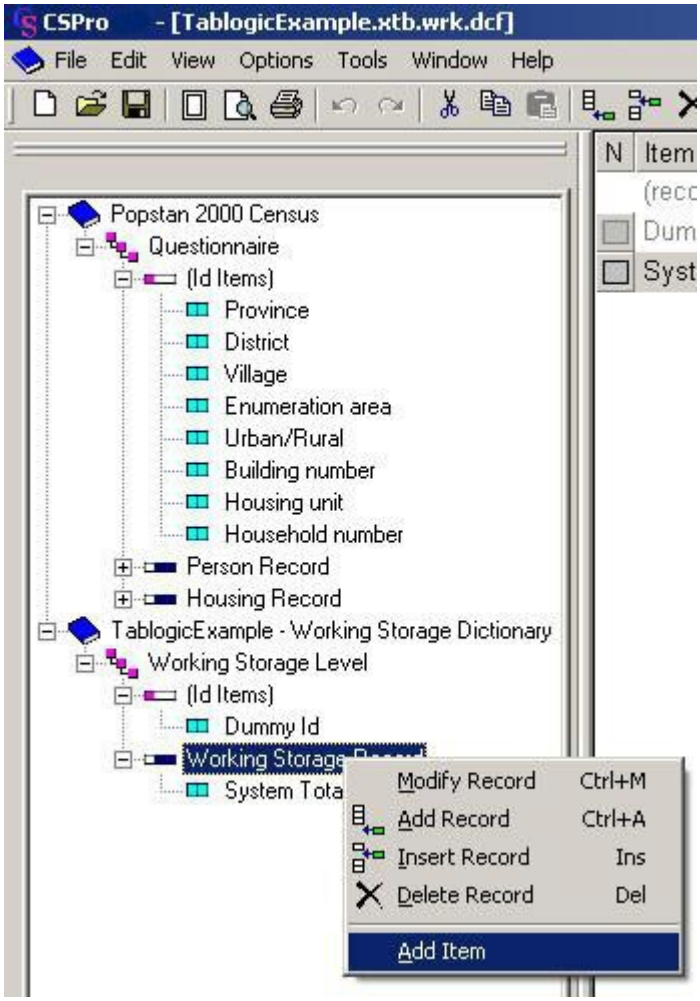
N	Value Set Label	Value Set Name	Value Label	From
☐	Type of bathing fac	H09_BATH_VS1		
☐			Exclusive	1
☐			Shared	2
☐			Hand basin	3
☐			Portable tub or basir	4
☐			Other	5
☐			None	6

N	Value Set Label	Value Set Name	Value Label	From
☐	Type of toilet faciliti	H08_TOILET_VS1		
☐			Private toilet	1
☐			Shared toilet	2
☐			Outhouse	3
☐			Pit	4
☐			Other	5
☐			None	6

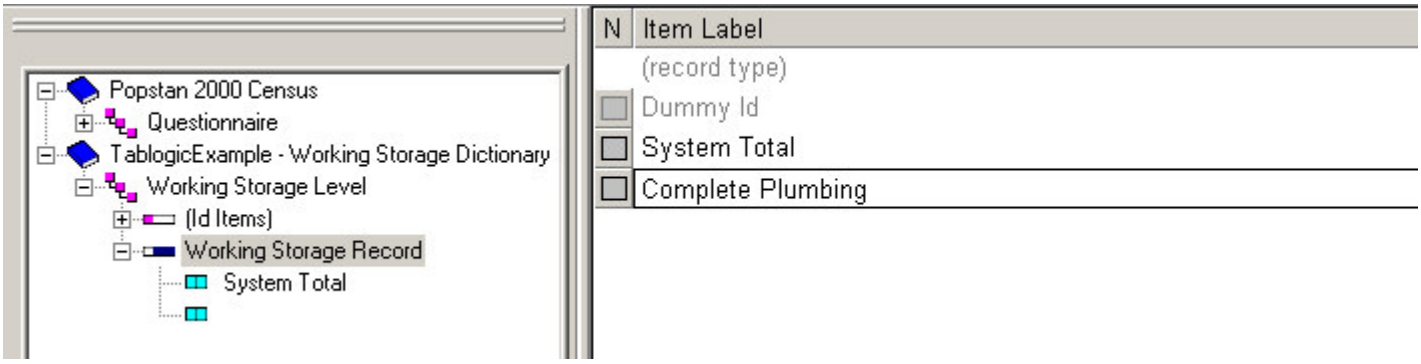
Rather than adding new variables to your existing dictionary, you can add new variables to the working storage dictionary. The working storage dictionary is a second dictionary that is created automatically when you create a tabulation application. This dictionary is similar to other dictionaries, however there is no data file associated with it. The variables in the working storage dictionary must be set by program logic.

In this example, we will add the new "complete plumbing variable" to the working storage dictionary. The working storage dictionary appears just below the main dictionary in the dictionary tree. Adding a new variable to the working storage

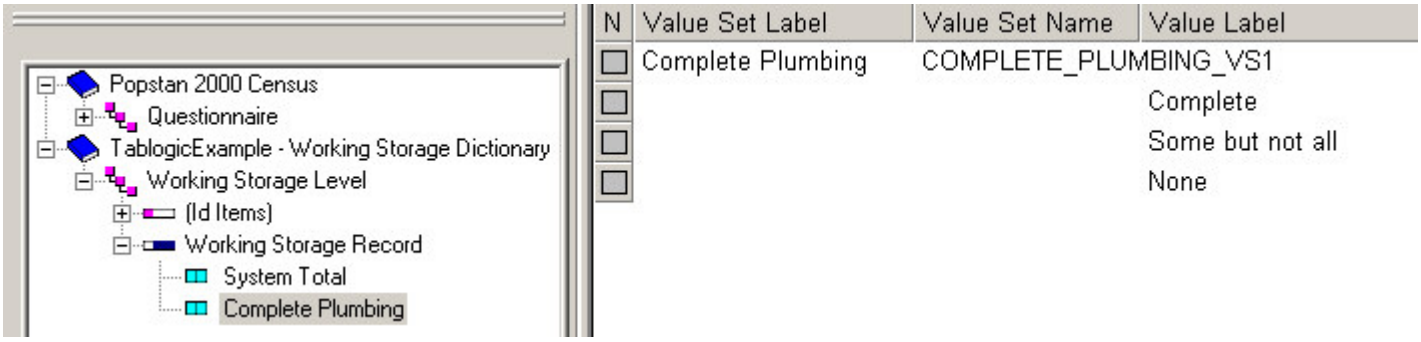
dictionary is the same as adding a variable to any dictionary. Right-click on the "Working Storage Record" under the working storage dictionary in the dictionary tree and choose "Add Item".



Then fill in the label, name and other properties for the new variable.



Next, create the value set for the new variable.

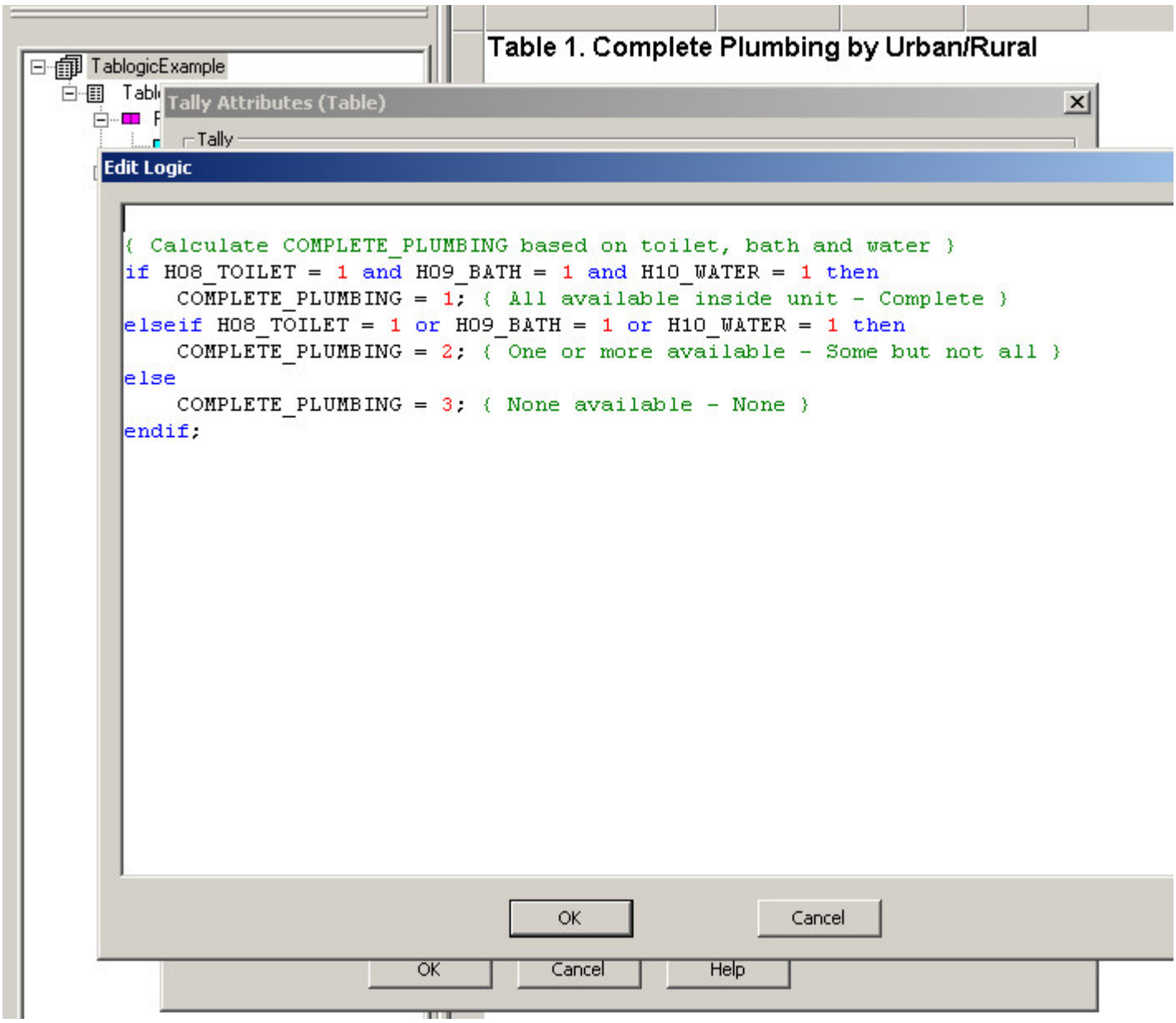


Now drag the new variable onto the table just as if it were a variable in the main dictionary.

Table 1. Complete Plumbing by Urban/Rural

Complete Plumbing	Urban/Rural		
	Total	Urban	Rural
Total			
Complete			
Some but not all			
None			

Finally add the tablogic to set the value of complete plumbing based on the values of source of water, toilet facilities and bathing facilities. Bring up the Tally Attributes (Table) dialog and enter the following code in the tab logic edit box:



The above logic will be executed once for each housing record to set the value of complete plumbing for that household. This will result in the following table:

Table 1. Complete Plumbing by Urban/Rural

Complete Plumbing	Urban/Rural		
	Total	Urban	Rural
Total	4,872	2,821	2,051
Complete	106	106	-
Some but not all	361	354	7
None	4,405	2,361	2,044

When using variables from the working storage dictionary, it is important to pay attention to the unit of tabulation. When using variables from the main dictionary, CSPro can determine the unit of tabulation based on the variables used. If you choose variables from the Person record, such as sex, the unit will be set to the person record. If you choose variables from the housing record, the unit will be set to the housing record. However, when you drag variables from the working storage dictionary onto the table, CSPro does not know which record or records the recodes for these variables will come from. It always sets the unit of tabulation to the first level (questionnaire). In the above example, this does not cause any problems because there is exactly one housing record per questionnaire. However, had we created a recode of variables from the person record, we would need to set the unit to the person record, otherwise we could get errors during tabulation. For more information see Changing the Unit of Tabulation.

Note that often it is preferable to create recoded variables in a batch edit program rather than during tabulation. This recode could have been accomplished by adding a new variable to the main dictionary for complete plumbing and writing a batch edit program to set the value of this variable for each case. Then the new complete plumbing variable could be dropped on the table directly, rather than using a working storage variable. The advantage of using batch edit is that the logic to set the value of the new variable is done once in a batch edit program rather than done as tablogic in each table in which the variable is used.

Another alternative is to use multiple subtables and value sets to simulate a recoded variable. This can be easier for simple variables, although for recoded variables with more than a few categories, it is generally simpler to use tablogic or batch edit. For more information see Recodes in Tables Using Value Sets and Subtables.

Tabulations Using Relations

Relations provide a way of linking one multiple record or item to one or more multiple records or items. Once the items on different records are linked through a relation, tallies are done just as if all the items belonged to one record.

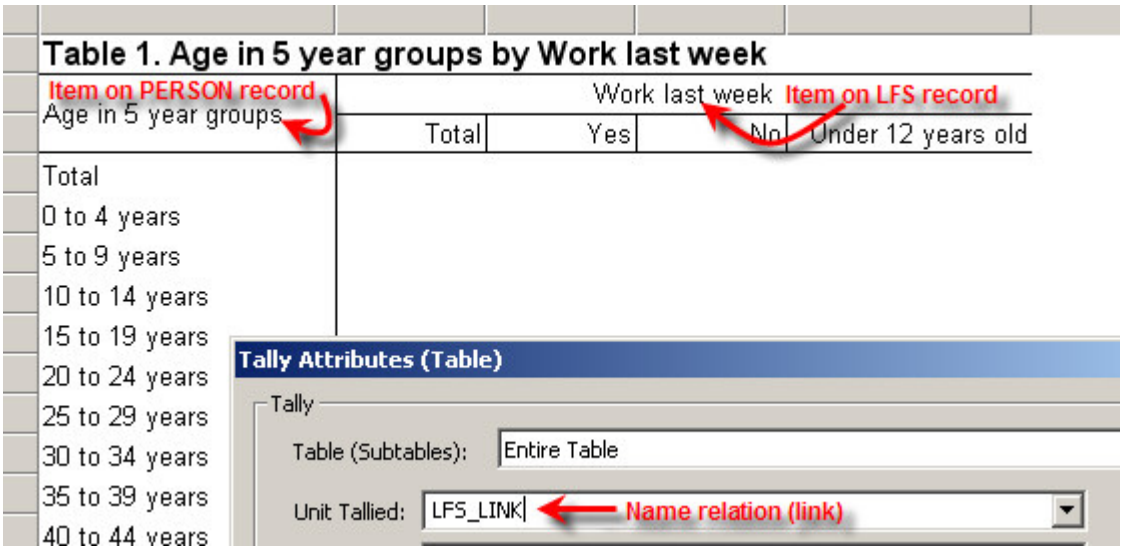
For example: LFS_LINK is a relation linking a labor force record (LFS) with a person record (PERSON). Both of these records are repeating records. The value that links two records is a number on each record (LFS_NUMBER and SERIAL_NUMBER) which identifies a single person, i.e., is the same on both records.

Here is the definition of the Link.

Relation Name Primary Linked by Secondary Linked by

LFS_LINK LFS LFS_NUMBER PERSON SERIAL_NUMBER

Here is table that tallies an item from both records:



If items from different multiply occurring records are used in the same subtable, CSPro will automatically choose the relation between them as the unit of tabulation if such a relation exists.

See Also: Relation Description

How To ...

Tabulate Items in Relations

A "relation" is created in the data dictionary. It links two record types through values on each record or occurrence number. If the two items have the same value then the records are linked, i.e., the items on the two record types are merged so all items "appear" to be on one big record.

A table or subtable using items from two record types linked through a relation is created in the same manner as any other table, i.e., drag and drop the items from each record. CSPro will automatically set the unit of tabulation to the relation between the two items. When the table is run, the tallies will be made from the linked records only.

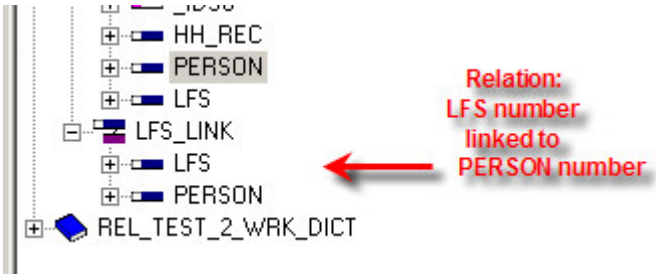
Example:
In a Labor Force Survey, the record with labor force information is related to the record with person information through sequence numbers.

Here is the definition of the Link.

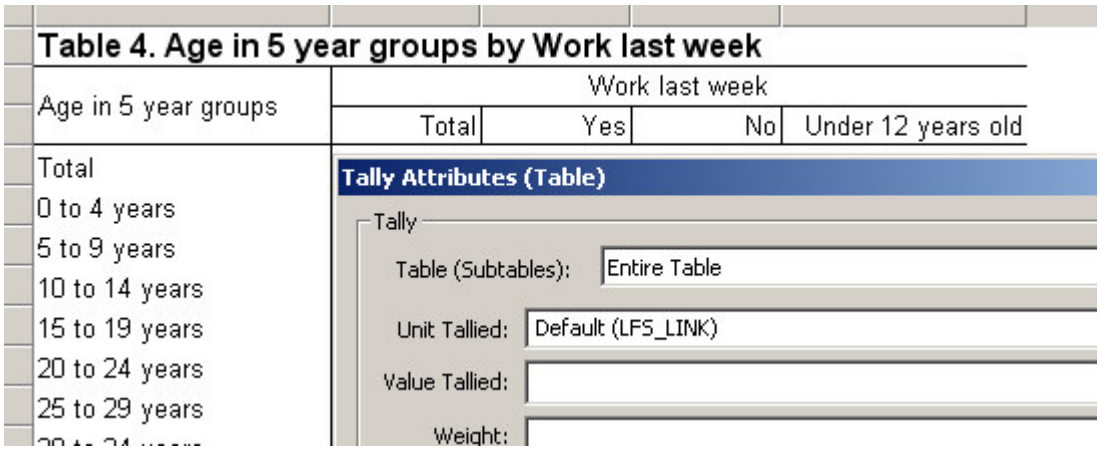
Relation Name Primary Linked by Secondary Linked by

LFS_LINK LFS LFS_NUMBER PERSON SERIAL_NUMBER

And how it is displayed in the data dictionary:



If "Age" value set is dropped for the Person record and the "Work last week?" value set is dropped from the LFS record then the following table is created.



Since the LFS record is the primary link, the tallies will be done for each occurrence of the LFS record. (This assumes that every LFS record in the case can be linked to a Person record in the case and that no "universe" has been specified.)

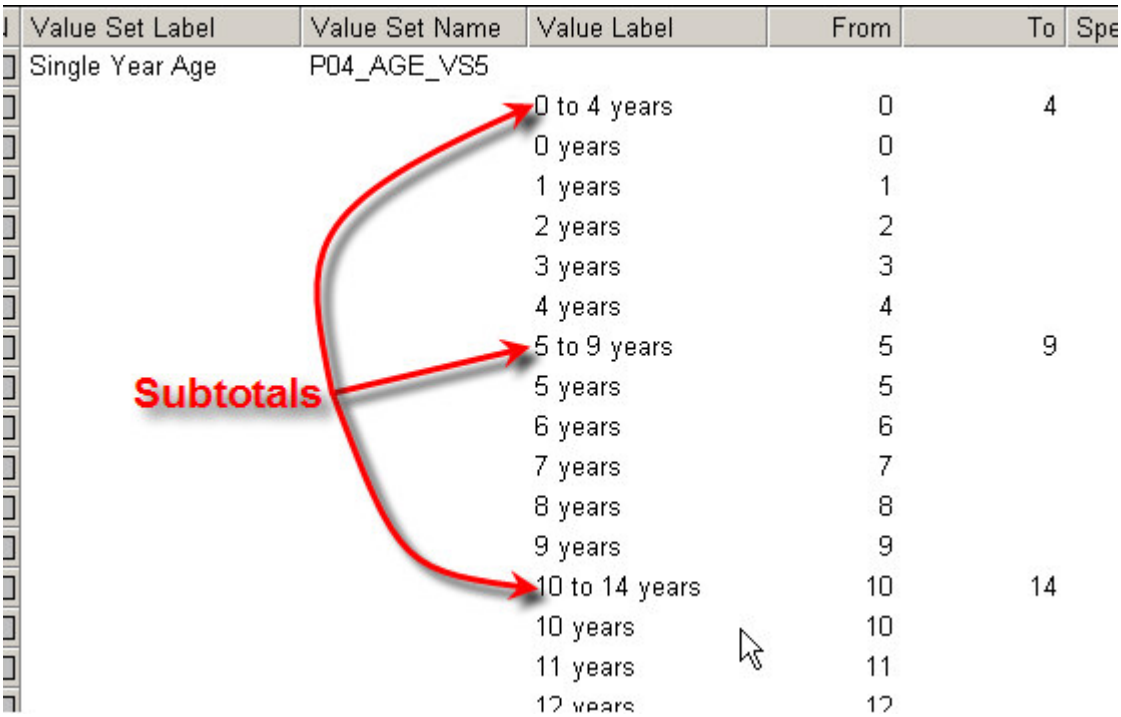
Note: The "Unit Tallied" must be the relation defined in the data dictionary. CSPro will automatically set the default to the relation if such a relation exists.

See Also: Relation Description

Table Tips and Tricks

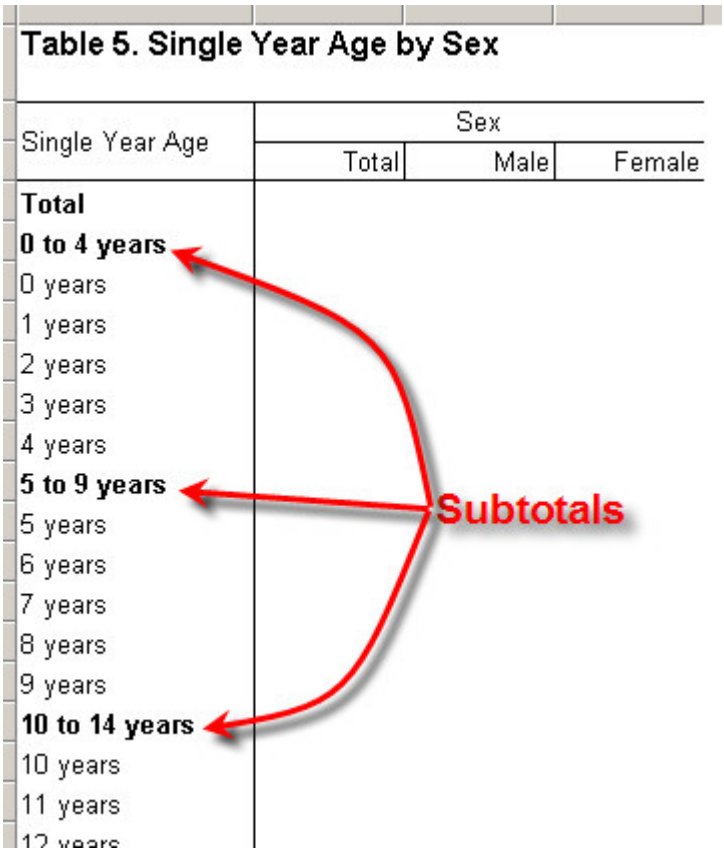
Add Subtotals to a Table

You can add subtotals to a table using overlapping value sets. In your value set, add an item for each subtotal in addition to the existing items. For example, if you have a value set with single years of age and you wanted to add subtotals for each 5-year age group, you would insert additional items in the value set for each of the 5-year groups:



Value Set Label	Value Set Name	Value Label	From	To	Spe
Single Year Age	P04_AGE_VS5				
		0 to 4 years	0	4	
		0 years	0		
		1 years	1		
		2 years	2		
		3 years	3		
		4 years	4		
		5 to 9 years	5	9	
		5 years	5		
		6 years	6		
		7 years	7		
		8 years	8		
		9 years	9		
		10 to 14 years	10	14	
		10 years	10		
		11 years	11		
		12 years	12		

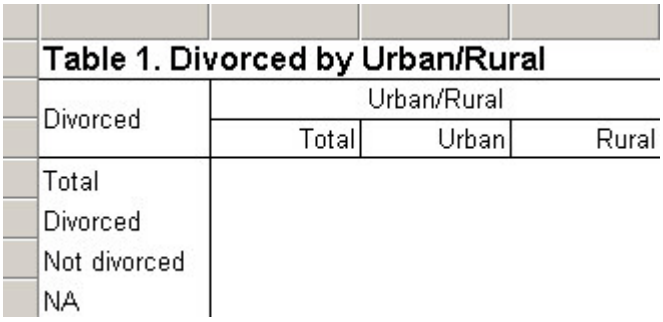
Then drag the modified value set onto the table. For additional clarity you can format the stubs corresponding to the subtotals to make them standout. In the table below, the font for the subtotal stubs is set to bold.



Tabulate Categories With Disjoint Values

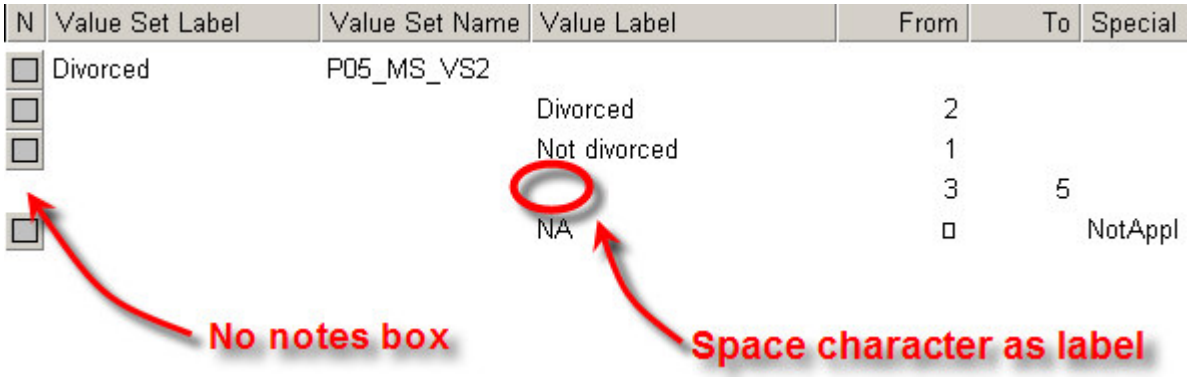
At times it is useful to create categories in a tabulation that contain disjoint values. For example, if you have the following value set for marital status where divorced is code 2 and you wish to create a table with only two categories: "divorced" and "not divorced".

N	Value Set Label	Value Set Name	Value Label	From	To	Special
☐	Marital status	P05_MS_VS1				
☐			Married	1		
☐			Divorced	2		
☐			Separated	3		
☐			Widowed	4		
☐			Never Married	5		
☐			NA	◻		NotAppl



To create such a table, we need a value set that contains only the two categories divorced and not divorced. Creating the category for "divorced" is simple, it includes only the value 2. The "not divorced" category, however, needs to contain the value 1 (Married) plus the values 3, 4 and 5 (Separated, Widowed, Never Married). These values do not make up a single range. In order to create the category you must create two entries in the value set, one that contains the value 1 and the second that contains the values 3 through 5. The first entry must contain the label for the category and the label for the second entry must be a single space character. CSPro recognizes the single space as a continuation of the previous category rather than a separate category. In this case it will merge the value range in the second entry with that of the

first entry, creating a single category that includes the value 1 and the values 3 through 5.



It is important that you use a space rather than no text at all since CSPro will treat an empty label all as a new category in the value set. You can tell when CSPro has combined the entries by the absence of the notes box in the second entry.

You can combine any number of entries in the value set into a single category.

See also: Implications of Data Dictionary Value Sets

Hiding Rows and Columns in Subgroupings

When you have a table that uses subgroupings (one variable that was dropped onto another), there are often problems when trying to hide rows or columns in those subgroupings. When you hide or format a row or column in a subgrouping, the corresponding rows/columns in the other sub-groupings are also hidden. This makes it difficult to hide or format a row or column in one subgrouping but not the others. In addition, it is not possible to hide an entire subgrouping by hiding the associated caption or spanner. Fortunately, using value sets and multiple subtables, there are workarounds for these restrictions.

Hiding Subgroupings

There are two ways to hide an entire subgrouping. One way is to modify the value set used and the other is to use multiple subtables rather than subgroupings. Take the example of a table with Sex and Literacy as subgroupings on the row where you want to only show the sub-groupings for Total and Female. In other words, you would like to hide the subgrouping for Male.

Table 1. Sex and Literacy by Urban/Rural

	Urban/Rural		
	Total	Urban	Rural
Sex			
Total			
Literacy			
Total			
Literate			
Illiterate			
Male			
Literacy			
Total			
Literate			
Illiterate			
Female			
Literacy			
Total			
Literate			
Illiterate			

Male subgrouping

If you try to hide the rows for the Male subgrouping using hide in the Format (Stub) dialog, it will not work. Hiding the Literate and Illiterate rows under Male will also hide those rows under Total and Female. Hide is not available in the Format (Caption) dialog if you right-click on the Male caption.

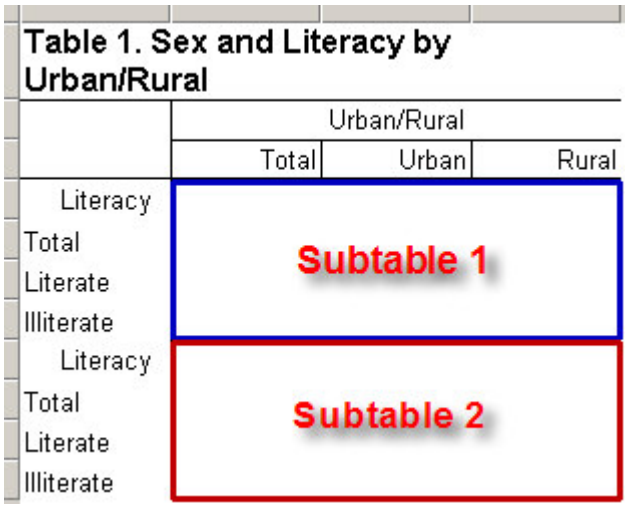
N	Value Set Label	Value Set Name	Value Label	From	To	Special
☐	Sex	P03_SEX_VS2				
☐			Total	1	2	
☐			Female	2		

One way to work around this is to modify the value set for Sex to remove the Male category. Note that removing the Male category will also affect the Total (since males will no longer be counted). To fix this, add your own category to the Sex value set called Total, which includes the values for Male and Female. Finally hide the system generated total in the table (see Hide or Change the Position of the Total). The result is the following table:

Table 1. Sex and Literacy by Urban/Rural

	Urban/Rural		
	Total	Urban	Rural
Sex			
Total			
Literacy			
Total	16,804	9,968	6,836
Literate	13,908	8,840	5,068
Illiterate	2,896	1,128	1,768
Female			
Literacy			
Total	8,751	5,275	3,476
Literate	7,063	4,607	2,456
Illiterate	1,688	668	1,020

A second approach is to use multiple subtables rather subgrouping. In our example you would drop the Literacy variable twice onto the rows to make two separate subtables. This would create the following table:



Next, modify the first subtable so that it becomes the subgrouping for Total and modify the second subtable so that it becomes the subgrouping for Male. To do this, first edit the captions for the two subtables to be "Total" and "Female" (see Customize Table Text). Next modify the universe of the second subtable to include only females (see Restrict a Universe). The first subtable will give the counts for all persons while the second subtable will give the counts for only females. This will give the same results as using the subgroupings:

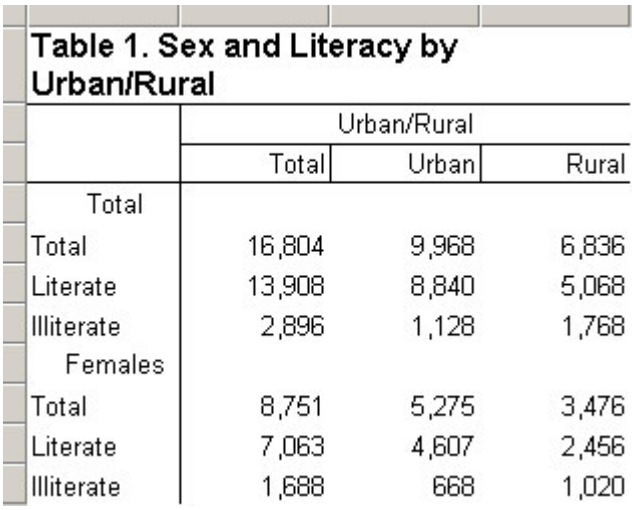


Table 1. Sex and Literacy by Urban/Rural

	Urban/Rural		
	Total	Urban	Rural
Total			
Total	16,804	9,968	6,836
Literate	13,908	8,840	5,068
Illiterate	2,896	1,128	1,768
Females			
Total	8,751	5,275	3,476
Literate	7,063	4,607	2,456
Illiterate	1,688	668	1,020

Formatting and Hiding Rows/Columns in a Subgrouping

To hide or format a row or column in one subgrouping without effecting the other subgroupings you must use multiple subtables rather than subgroupings. This is because changing a row or column in one subgrouping will automatically cause the same change to the corresponding row in the other subgroupings. This is true for all format changes (font, color, alignment,…) as well as hiding rows/columns and modifying stub/column head text.

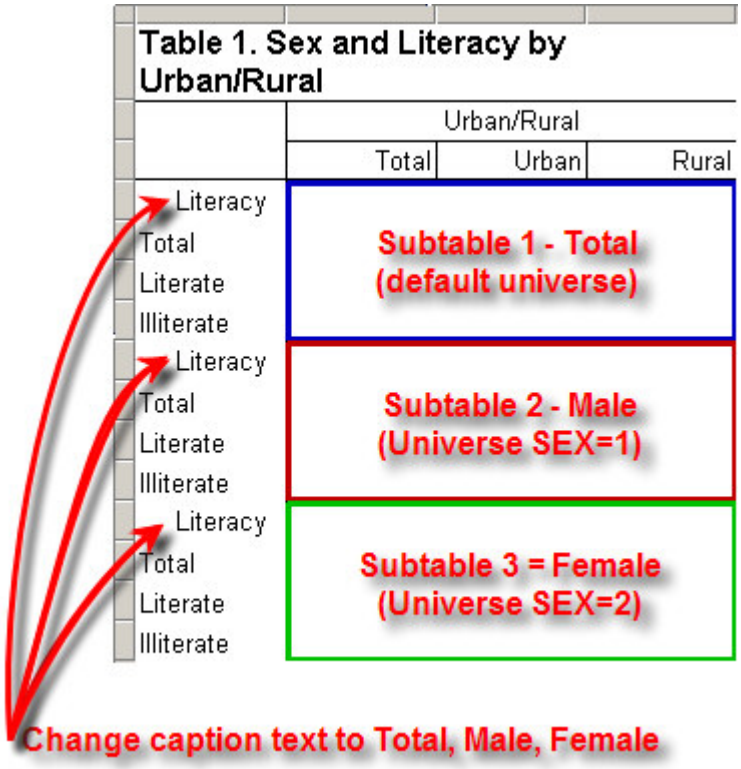
Take the earlier example of the Sex and Literacy table and try to make the text in the literacy row under the Total subgrouping bold without making the other literacy rows bold. If you simply set the format for this row to bold, the literacy rows in the Male and Female subgroupings will also become bold.

Table 1. Sex and Literacy by Urban/Rural

	Urban/Rural		
	Total	Urban	Rural
Sex			
Total			
Literacy			
Total	16,804	9,968	6,836
Literate	13,908	8,840	5,068
Illiterate	2,896	1,128	1,768
Male			
Literacy			
Total	8,053	4,693	3,360
Literate	6,845	4,233	2,612
Illiterate	1,208	460	748
Female			
Literacy			
Total	8,751	5,275	3,476
Literate	7,063	4,607	2,456
Illiterate	1,688	668	1,020

Format only this row

Instead of using subgroupings, drag the variable Literacy onto the rows three times to create separate subtables for the Total, Male and Female subgroupings.



Then edit the universe for the second subtable to only include males (see Restrict a Universe). In this case set the universe to SEX=1. Next edit the universe for the third subtable to include only females (SEX=2). Note that you do not need to edit the universe for the first subtable since you want to include both males and females (the entire population) which is the default universe. Next, edit the captions of the subtables to be "Total", "Male" and "Female" (see Customize Table Text). Finally right-click on the Literacy stub in the first (Total) subtable and set the font to bold. This results in the following table:

Table 1. Sex and Literacy by Urban/Rural			
	Urban/Rural		
	Total	Urban	Rural
Total			
Total	16,804	9,968	6,836
Literate	**13,908**	**8,840**	**5,068**
Illiterate	2,896	1,128	1,768
Male			
Total	8,053	4,693	3,360
Literate	6,845	4,233	2,612
Illiterate	1,208	460	748
Female			
Total	8,751	5,275	3,476
Literate	7,063	4,607	2,456
Illiterate	1,688	668	1,020

The same procedure works for hiding or modifying the stub text of an item in a subgrouping, only the last step changes. Rather than setting the font to bold, you would hide the row or modify the stub/column head text.

See also: Hide or Show a Row or Column, Hide or Change the Position of the Total, Formats for a Part of a Table, Implications of Data Dictionary Value Sets, Using Subtables

Recodes in Tables Using Value Sets and Subtables

Often the variables in your dictionary do not map exactly to the categories desired in your tables and you need to "recode" one or more variables to map it to the groupings you want in the rows and columns of your table. Recodes involving a single variable can be accomplished by creating additional value sets while recodes of multiple variables can be accomplished using multiple subtables.

Single variable recodes

To recode a single variable, simply create an additional value set for that variable in your dictionary, which regroups the values as desired. Then drag this new value set onto the table.

For example if you have the following value set for marital status but would like to count only two categories, Married and Unmarried, in your table.

N	Value Set Label	Value Set Name	Value Label	From	To	Special
☐	Marital status	P05_MS_VS1				
☐			Married	1		
☐			Divorced	2		
☐			Separated	3		
☐			Widowed	4		
☐			Never Married	5		

Create a new value set that regroups the original values for marital status into these new categories.

N	Value Set Label	Value Set Name	Value Label	From	To	Special
☐	Marital Status	P05_MS_VS2				
☐			Married	1		
☐			Unmarried	2	5	

Now drag the new value set onto your table.

Table 1. Marital Status by Urban/Rural			
Marital Status	Urban/Rural		
	Total	Urban	Rural
Total	24,271	13,920	10,351
Married	6,985	3,854	3,131
Unmarried	17,286	10,066	7,220

Value sets may have disjoint values, one category which contains values from different ranges, for example, one category for values 23, 45, and 67-71 (see Tabulate categories with disjoint values).

Multiple variable recodes

To do a recode using multiple variables, create multiple subtables by dragging the variables onto the table multiple times and set universes on these subtables. For example, if you want to tabulate houses that have "complete plumbing", meaning that they have piped water, bathing facilities and a toilet inside the housing unit. This involves the following three variables:

N	Value Set Label	Value Set Name	Value Label	From
☐	Source of water	H10_WATER_VS1		
☐			Inside piped	1
☐			Outside piped	2
☐			Bottled or canned	3
☐			Closed well or close	4
☐			Open well or spring	5
☐			River/Stream/Other	6

N	Value Set Label	Value Set Name	Value Label	From
☐	Type of bathing fac	H09_BATH_VS1		
☐			Exclusive	1
☐			Shared	2
☐			Hand basin	3
☐			Portable tub or basir	4
☐			Other	5
☐			None	6

N	Value Set Label	Value Set Name	Value Label	From
☐	Type of toilet faciliti	H08_TOILET_VS1		
☐			Private toilet	1
☐			Shared toilet	2
☐			Outhouse	3
☐			Pit	4
☐			Other	5
☐			None	6

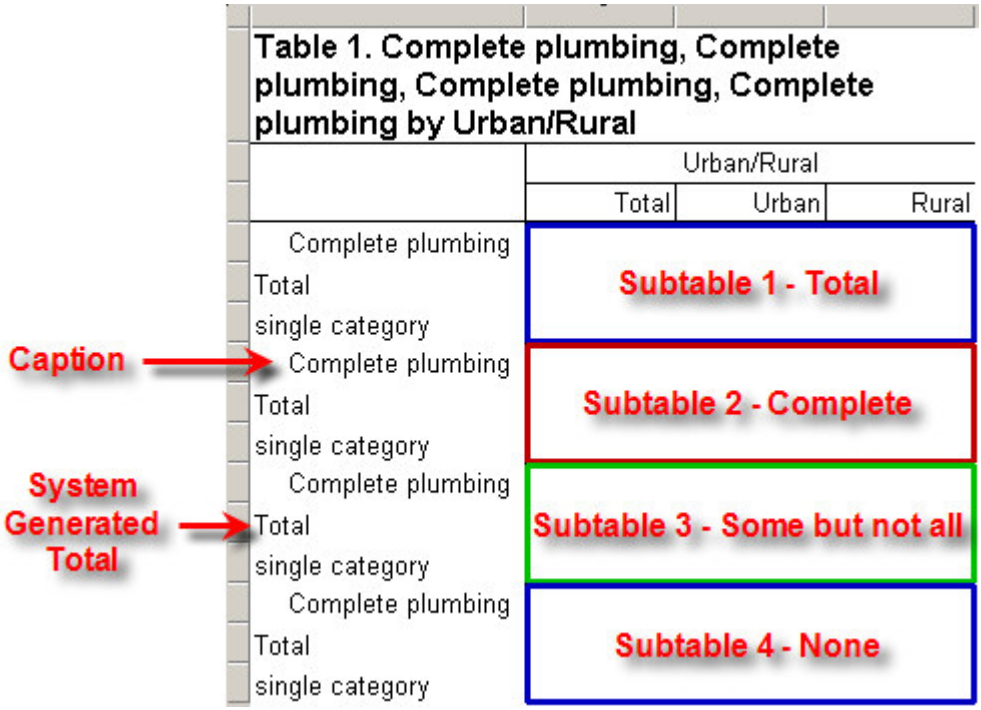
The "recoded" variable, "complete plumbing", has three categories:

- Complete – piped water inside the unit (source of water = 1), and a private toilet inside the unit (type of toilet = 1) and bathing facilities inside the unit (type of bathing = 1).
- Some but not all – any one of the three variables (piped water, flush toilet, bathing facilities) inside the unit but not all three.
- None – all other cases.

To create the table for complete plumbing, use 3 subtables, one for each of the 3 possible values of the recoded variable (complete, some but not all, none). Each of these subtables should have only one row. The easiest way to create a subtable with one row is to drag a variable whose value set has only one category onto the table. If no such value set exists, you can create one. In this case, create the value set on the variable "source of water". In fact, you could use any variable on the same record as the variables we are using in the recode (the housing record in this case). Note that the single category in this value set must include all valid values for the variable so that all housing units will be counted when tabulating the variable.

N	Value Set Label	Value Set Name	Value Label	From	To
☐	Source of water	H10_WATER_VS1			
☐			Inside piped	1	
☐			Outside piped	2	
☐			Bottled or canned	3	
☐			Closed well or close	4	
☐			Open well or spring	5	
☐			River/Stream/Other	6	
☐	Complete plumbing	H10_WATER_VS2			
☐			single category	1	9

Drag this new value set onto the table 4 times, once for each of three possible values of complete plumbing and once for the total.



Since we only need one row for each subtable, hide the system generated total in each of the four subtables (see Hide or Change the Position of the Total). Also hide the captions for each of the subtables (see Formats for a Part of a Table).

Table 1. Complete plumbing, Complete plumbing, Complete plumbing, Complete plumbing by Urban/Rural			
	Urban/Rural		
	Total	Urban	Rural
Complete plumbing single category			
Complete plumbing single category			
Complete plumbing single category			
Complete plumbing single category			

Next customize the text in the stubs to match the categories for the recode (Total, Complete, Some but not all, None). Also edit the title of the table and put the name of the variable in the Stub Head. See Customize Table Text for how to edit text in the table.

Table 1. Complete plumbing by Urban/Rural			
Complete Plumbing	Urban/Rural		
	Total	Urban	Rural
Complete plumbing Total			
Complete plumbing Complete	WATER = 1 and TOILET = 1 and BATH = 1		
Complete plumbing Some but not all	(WATER = 1 or TOILET = 1 or BATH = 1) and not (WATER = 1 and TOILET = 1 and BATH = 1)		
Complete plumbing None	WATER <> 1 and TOILET <> 1 and BATH <> 1		

Now set the universes for the subtables to correspond to the appropriate values of the complete plumbing recode. The first subtable will be the total so it should use the default universe, which includes all cases. The second subtable represents "complete plumbing" and must include only those cases where piped water, flush toilet AND bathing facilities all equal 1. The third subtable represents "some but not all" and must include the cases where one or two of the variables are equal to 1 but not all of them. The fourth subtable represents "none" and must include all cases where none of the variables are equal to 1. For more information on setting the universe on a subtable see Restrict a Universe. This produces the following table:

Table 1. Complete plumbing by Urban/Rural			
Complete Plumbing	Urban/Rural		
	Total	Urban	Rural
Total	4,420	2,469	1,951
Complete	106	106	-
Some but not all	361	354	7
None	3,953	2,009	1,944

Note that often it is preferable to create recoded variables in a batch edit program rather than during tabulation. This recode could have been accomplished by adding a new variable to the dictionary for complete plumbing and writing a batch edit program to set the value of this variable for each case. Then the new complete plumbing variable could be dropped on the table directly rather than creating multiple subtables with universes. This makes the creation of the table much simpler although it involves creating a batch edit application and writing a small amount of logic. Using batch edit is best when the recoded variable will be used in multiple tables.

Another alternative would be to create the recoded variable in the working storage dictionary and use tablogic to set its

value to each case. See Table Logic (tablogic) for more information.

Use Expressions in Universe and Value Tallied

You can enter any valid CSPro expression as a universe, weight or value tallied. This includes not only variables, constants, logical operators (`and`, `or`, `not`), and mathematical operators (+, -, *, /, ...) but also CSPro functions (`count`, `sum`, `max`, `special`, ...). Making full use of these expressions allows you to create complex tables very easily. Here are some examples:

Universe

Tabulate only households where at least one child was born last year with the following universe (assuming that CHILDREN_BORN_LAST_YEAR is a variable on the person record):

```
count(CHILDREN_BORN_LAST_YEAR) > 0
```

Count only households with "married couple families," i.e. a household with both a head of household and a spouse present, with the following universe:

```
count(PERSON where RELATIONSHIP = 1) > 0 and count(PERSON where RELATIONSHIP = 2) > 0
```

Restrict a table to only households with total household income of greater than $20,000 by using the following universe:

```
sum(PERSONAL_INCOME) > 20000
```

This assumes that PERSONAL_INCOME is a variable on the person record. The sum of the incomes of each person in the household is the total income for the entire household.

Restrict a table to only households where the head of household is female with the following universe:

```
SEX(1) = 2
```

This assumes that the head of the household is always the first person record (occurrence 1). This will only be true if your data entry program and/or edit program ensure this. If this is not true, then you would need a more complicated expression such as:

```
count(PERSON where RELATIONSHIP = 1 and SEX = 2) = 1
```

This will be true only if there is a person in the household who is the head of household and is female. This assumes that there is only one head of household in the household, which should be the case for properly edited data.

Value Tallied

Often fertility information is captured separately for male and female children and you wish to tabulate it for both sexes. For example, you have variables for `MALE_CHILDREN_BORN` and `FEMALE_CHILDREN_BORN` but no variable for total children born and you want to count the total number of children born. You can use the sum of the two variables in the value tallied:

```
MALE_CHILDREN_BORN + FEMALE_CHILDREN_BORN
```

Note that if one or more of the variables is a special value, the counts will not be correct. This is because the sum of a special value and a number is a special value.

See also: Restrict a Universe, Tabulate Values Instead of Frequencies, Tally Attributes for a Table

CSPro Statements and Functions

Statement Format Symbols

The formats of statements and function use the following symbols:

reserved Name of the statement or function, or a another reserved word.

keyword Keyword used within this statement or function.

element Type of object used as a part of a statement or argument of a function.

A | B One or the other of the symbols A or B may be used.

[A] The symbol A is optional.

A, B, ... More symbols of the same type.

Alphabetical List of Functions and Statements

The following table lists functions and statements that can be used in logic. In addition, some CSPro objects have functions that can be invoked on the object. A list of such functions can be found in the summary pages for each of these objects:

- array
- audio
- document
- file
- freq
- geometry
- hashmap
- image
- list
- map
- pff
- report
- SystemApp
- valueset

Function/Statement	Description
abs	Returns the absolute value of a numeric expression.
accept	Returns the number of a choice from a list made by the data entry operator.
adjlba	Adjusts the lower bound of the CMC of an event based on an age.
adjlbi	Adjusts the lower bound of the CMC of an event offset by a number of months.
adjuba	Adjusts the upper bound of the CMC of an event based on an age.
adjubi	Adjusts the upper bound of the CMC of an event offset by a number of months.
advance	Moves forward field by field to a specified field during data entry.
alias	Creates an aliased name for a dictionary item, typically to shorten or standardize names.
alpha	Declares alphanumeric variables used in the application.
array	Declares an array object for storing a collection of numbers or strings.
ask	Conditionally jumps forward to the next field during data entry.

audio	Declares an audio object for recording or playing audio.
average	Returns the average of an item that occurs multiple times.
barcode.read	Reads a barcode and returns its value as a string.
break	Exits a do, while, or for loop early and continues execution with the first statement after the enddo.
changekeyboard	Modifies the keyboard input associated with certain fields.
clear	Initializes the memory values of data items defined in external files to zero or blank.
close	Closes a previously opened file.
cmcode	Returns the number of months since the year 1900 given a month and year.
compare	Returns the alphabetical order (i.e., collating sequence) of the two strings.
compress	Compresses some number of files into a single .zip file.
concat	Joins two or more strings into one string.
config	Declares string variables prefilled with values defined in the configuration settings.
connection	Determines whether the device has access to the Internet.
count	Returns the number of occurrences for a repeating form or roster.
countcases	Returns the number of cases in an external file that meet a certain criterion.
countnonspecial	Returns the number of non-special values in a group of data items.
curocc	Returns the current occurrence number for a repeating form, roster, or record.
currentkey	Returns the key (ID string) comprised of the case's current IDs.
dateadd	Calculates a new date from a starting date and a period of elapsed time.
datediff	Calculates the difference between two dates.
datevalid	Determines whether a date in the format YYYYMMDD is valid.
decompress	Decompresses a .zip file, extracting some number of files.
delcase	Marks a case for deletion in an external file based on a key.
delete	Removes a record or item occurrence from the current case.
demode	Returns the current data entry mode.
diagnostics	Returns troubleshooting information about CSPro.
dircreate	Creates a new directory with the given directory name.
dirdelete	Deletes an empty directory or group of empty directories.
direxist	Determines whether a directory exists.
dirlist	Returns a list containing the file and directory listing of the specified directory.
display	Displays a message (this function has been superceded by errmsg).
do	Executes one or more statements repeatedly while a logical condition remains true or until a logical condition is no longer true.
document	Declares a document object for storing text or binary files.
edit	Converts a number to a string.
editnote	Displays the data entry note box for adding or changing.
encode	Encodes special characters to facilitate writing to HTML or CSV files and writing to other formats.
endcase	Ends batch editing for the current case (but outputs the case to file).

endgroup	Ends data entry for the current record or group/roster.
endlevel	Ends data entry for the current level.
enter	Enters data from a secondary form file.
errmsg	Displays or writes a message.
ensure	Conditionally declares numeric variables used in the application.
execpff	Starts another CSPro application.
execsystem	Starts another application or process.
exit	Ends a procedure before normal processing is expected to end.
exp	Returns the value of e raised to a given power.
export	Writes a record to an export file.
file	Declares one or more files used in the application.
fileconcat	Concatenates a list of files or a set of files described by a wildcard specification.
filecopy	Copies a file to another file.
filecreate	Creates a new file with the given file name.
filedelete	Deletes a file or group of files.
fileempty	Determines whether a file exists but is empty.
fileexist	Determines whether a file exists.
filename	Returns the file name currently associated with a dictionary or a file.
fileread	Reads a text line from a file into an item or variable.
filerename	Changes the name of a file.
filesize	Returns the size of a file in bytes.
filetime	Returns the last modification date/time of a file or a directory.
filewrite	Writes a line of text to a file.
find	Determines the existence of a case in a data file that matches a condition.
for	Loops through multiple records or items.
for (dictionary)	Loops through the cases selected during a selcase operation.
forcase	Loops through the cases in an external file.
freq	Declares a freq object that allows for the creation of frequencies.
function	Declares a user-defined function.
geometry	Declares a geometry object for storing vector geometry.
getbluetoothname	Returns the name of the device as broadcast to other Bluetooth devices.
getbuffer	Returns a string containing the contents of a data item.
getcapturetype	Returns the capture type associated with a field.
getcaselabel	Returns a string containing a case's label (the text that appears in a case listing).
getdeck	Retrieves a value from a DeckArray used for editing data.
getdeviceid	Returns a string that contains a unique value that identifies the device.
getimage	Returns the filename of a value set image.
getlabel	Returns the label of a dictionary symbol or text associated to symbol's value.
getlanguage	Returns the current language being used while in data entry mode.
getnote	Returns a field or case note.
getocclabel	Returns an occurrence label associated with a repeating dictionary item or a roster.

getoperatorid	Returns the text entered in the operator ID.
getorientation	Returns the display orientation.
getos	Returns information about the operating system, including whether it is running Android or Windows.
getproperty	Returns information about an application or field property.
getrecord	Returns the name of the record containing a particular item.
getsymbol	Returns the name of the current procedure being executed.
getusername	Returns the name of the user logged into Windows or the Android device account.
getvalue	Returns the value of a numeric variable whose identity is only known at runtime.
getvaluealpha	Returns the value of a alpha variable whose identity is only known at runtime.
getvaluenumeric	Returns the value of a numeric variable whose identity is only known at runtime.
gps	Controls and returns values from a GPS receiver.
has	Determines whether any of a group of repeating items is within a range of values.
hash	Calculates the hash value of a string.
hashmap	Declares a hashmap object for storing a collection of numbers or strings in an associative array.
hideocc	Visually hides an occurrence row in a roster.
high	Returns the maximum value in a group of numeric expressions.
highlighted	Returns whether a field is on path or reached during data entry.
htmldialog	Displays a custom dialog designed in HTML and JavaScript.
if	Executes statements conditionally.
image	Declares an image object for storing and manipulating images.
impute	Assigns a value to a data item and logs the frequency of assignments.
in	Determines whether a variable is within a range of values.
inadvance	Indicates if an advance is in progress.
inc	Increments a numeric item.
insert	Creates a record or item occurrence in the current case.
int	Returns the integer portion of a numeric expression.
invalueset	Determines whether a data item's value is within the item's value set.
invoke	Executes a user-defined function using runtime binding.
ischecked	Returns whether a code is checked as part of a check box field's selections.
ispartial	Determines whether a case was opened from a partial case or not.
isverified	Determines whether a case was verified by double-keying.
key	Returns the key (ID string) of the case's IDs as they existed when the case was last loaded or written.
keylist	Returns a list containing the keys (ID strings) of cases in a file.
killfocus	Declares that the following statements are executed after the object stops being active.
length	Returns the length of a dictionary item or a string.
list	Declares a list object for storing a collection of numbers or strings.
loadcase	Reads a case from an external file into memory (usually based on a

loadsetting	Returns a string containing a setting saved using the savesetting function.
locate	Finds but does not load a case in an external file that matches a condition.
log	Returns the base-10 logarithm of a numeric expression.
logtext	Writes a user-defined message to the paradata log file.
low	Returns the minimum value in a group of numeric expressions.
maketext	Returns a formatted string with inserted values.
map	Declare a map variable that can be used to display interactive maps.
max	Returns the maximum value of an item that occurs multiple times.
maxocc	Returns the maximum number of occurrences of a multiply occurring object.
maxvalue	Returns the maximum value in an item's range of acceptable values.
min	Returns the minimum value of an item that occurs multiple times.
minvalue	Returns the minimum value in an item's range of acceptable values.
move	Moves backwards or forwards to a specified field during data entry.
next	Ends a do, while, or for loop early and continues execution with the next iteration of the loop.
nmembers	Returns the number of cases selected during a selcase operation.
noccurs	Returns the number of occurrences for a repeating form or roster.
noinput	Prevents input for the current field during data entry.
numeric	Declares numeric variables used in the application.
OnChangeLanguage	Provides control over actions to occur after a user has changed the language.
OnChar	Allows users to trap characters in order to perform special actions or to change the action of the character.
onfocus	Declares that the following statements are executed just before an object becomes active.
OnKey	Allows users to trap keystrokes in order to perform special actions or to change the action of the key.
OnRefused	Provides a way to override the selection of a refused value.
onoccchange	Declares that the following statements are executed when the group's current occurrence changes.
OnStop	Provides control over stopping or exiting data entry.
OnSyncMessage	Allows users to respond to a message sent by a Bluetooth client using syncmessage.
OnSystemMessage	Provides a way to override the displaying of system error messages.
open	Opens a dictionary or external file.
paradata	Performs an action on the paradata log.
path.concat	Combines multiple strings into one string representing a file path.
path.getDirectoryName	Extracts the directory name from a file path.
path.getExtension	Extracts the file extension from a file path.
path.getFileName	Extracts the file name from a file path.
path.getFileNameWithoutExtension	Extracts the file name without its extension from a file path.
path.getRelativePath	Returns a relative path based on a path's location relative to a directory.
path.selectFile	Shows the operator a visual listing of the file system and returns the

	path of the file the operator selected.
pathconcat	Combines multiple strings into one string representing a file path.
pathname	Returns the path of various special directories.
pff	Declare a pff variable that can be used to dynamically modify the way other CSPro applications are executed.
pos	Returns the position of a string within another string.
poschar	Returns the position of any of a series of characters within another string.
preproc	Declares that the following statements are executed at the beginning of a block.
proc	Declares the beginning of a new procedure.
prompt	Prompts the operator to enter an alphanumeric string, also supporting a password masking option.
protect	Modifies the protected property of a field.
postproc	Declares that the following statements are executed at the end of a block.
publishdate	Returns the date and time that the code was compiled (which can be used to determine when a .pen file was created).
putdeck	Puts a value in a DeckArray used for editing data.
putnote	Puts the contents of string to a field or case note.
random	Returns a pseudo-random integer in a given range.
randomin	Returns a pseudo-random integer in a non-continuous range, or using a value set.
randomizevs	Randomizes the order of a value set.
recode	Assigns a value to a variable based on the value of one or more other variables.
reenter	Forces the data entry operator to reenter a previous entered field.
regexmatch	Returns whether a string matches the regular expression.
relation	Defines a relation between multiple records or items.
replace	Returns a string with one of more instances of a substring replaced with new text.
report	Generates a templated report.
retrieve	Loads a case from the current position of an external file.
round	Returns a value rounded to the nearest integer.
savepartial	Saves the current case as a partially added, modified, or verified case.
savesetting	Saves, using an attribute-value pair, a setting retrievable with the loadsetting function.
seed	Initializes the random number generator to a particular starting place.
seek	Searches a multiply occurring item for an item that meets a condition.
seekmax	Searches a multiply occurring item for an item with the highest value that meets a condition.
seekmin	Searches a multiply occurring item for an item with the lowest value that meets a condition.
selcase	Allows a data entry operator to select and load a case from an external file.
set	Sets the values of various system parameters.
set_access	Sets the order in which cases in a data file are processed.
set_attributes	Modifies the field properties in a data entry application.
set_behavior	Modifies parameters related to field validation or sets export

	parameters.
set first	Positions the first case in a data file as the next case to be processed.
set errmsg	Modifies the way that error messages are displayed on desktop CSEntry.
set last	Positions the last case in a data file as the next case to be processed.
setbluetoothname	Sets the name of the device as broadcast to other Bluetooth devices.
setcapturepos	Allows the manual declaration of the window coordinates of the capture type.
setcapturetype	Sets the capture type associated with a field.
setcaselabel	Sets a case's label (the text that appears in a case listing).
setfile	Assigns a new physical file to a dictionary or declared file.
setfont	Changes the default display font in data entry applications.
setlanguage	Modifies the current language being used while in data entry mode.
setlb	Returns the lower bound of the CMC of an event.
setocclabel	Sets the occurrence label associated with a repeating dictionary item or a roster.
setoperatorid	Changes the operator ID associated with the data entry session.
setorientation	Modifies the display orientation, allowing dynamic rotation of the display.
setoutput	Changes the output file where cases are saved in a batch application.
setproperty	Modifies an application or field property.
setreportdata	Stores a piece of data to be used in a templated report.
setub	Returns the upper bound of the CMC of an event.
setvalue	Assigns a value to a variable whose identity is only known at runtime.
setvalueset	Dynamically modifies the value set of an item.
setvaluesets	Dynamically modifies the value sets for all items in the dictionary.
show	Displays the occurrences of one or more items from a record or multiply occurring item.
showarray	Displays the values of one or more elements from an array.
showocc	Visually shows (or hides) an occurrence row in a roster.
skip	Jumps forward to a specified field during data entry.
skip case	Ends processing of the current case in a batch edit run.
soccurs	Returns the number of occurrences of a record.
sort	Sorts occurrences of records or items based on the value of an item.
special	Determines whether a variable's value is missing, refused, notappl, or default.
sqlquery	Executes a query on a SQLite database and returns a result set in a variety of formats.
sqrt	Returns the square root of a numeric expression.
startswith	Returns whether a string begins with a specified prefix.
stop	Ends a data entry session or batch edit run.
string	Declares variable-length alphanumeric variables used in the application.
strip	Removes trailing blanks from a string.
sum	Returns the sum of an item that occurs multiple times.
swap	Switches the order of two occurrences within a group.
syncapp	Updates an application from a new version of the deployed package on a server.
syncconnect	Connects to a CSWeb, Dropbox, or FTP server, or to another device using Bluetooth, to begin a synchronization session.
syncdata	Sends or receives cases between the current device and a server or another device.

syncdisconnect	Ends the current synchronization session.
syncfile	Sends or receives files between the current device and a server or another device.
syncmessage	Sends and receives a string message to a Bluetooth server.
syncparadata	Sends or receives paradata between the current device and a server or another device.
syncserver	Starts running a server on the current device, to which other devices can connect using Bluetooth.
synctime	Returns the time of the last synchronization of a data file or a specific case.
sysdate	Returns the current system date.
sysparm	Returns a parameter a PFF file.
SystemApp	Declares a SystemApp object for interacting with external applications.
systime	Returns the current system time.
timestamp	Returns the current UNIX time.
timestring	Returns a custom formatted date and time string.
tolower	Changes all uppercase letters in a string to lowercase.
tonumber	Converts a string to a number.
totocc	Returns the total occurrences for a repeating form, roster, or record.
toupper	Changes all lowercase letters in a string to uppercase.
tr	Translates a string literal into an appropriate string based on the application's current language.
trace	Controls a window or file that can contain debugging information.
universe	Specifies a condition that must be true for execution of a procedure or function to continue.
userbar	Controls the optional display bar in a data entry application.
uuid	Returns a human-readable string containing a universally unique identifier.
valueset	Declares a valueset object for storing the contents of a dynamic value set.
view	Launches the system's default viewer to display a file or website.
visualvalue	Returns the value of a data item prior to its input.
warning	Displays a "soft check" message only when the operator is not advancing in the case.
when	Executes a statement based on the value of one or more other variables.
while	Executes one or more statements repeatedly while a logical condition remains true.
write	Write to a text file.
writecase	Writes a case from memory to an external file.

See also: List of Reserved Words

List of Reserved Words

CSPro does not allow certain names to be used as dictionary unique names, or as variables in the programming logic,

as they are part of CSPro's procedural language. During compilation, the system will notify you if you have used a reserved word.

In addition to the list of reserved words below, there are a few reserved words used internally by CSPro. When you are writing logic, reserved words are shown in blue, so if you attempt to create a variable using one of these reserved words, you will know this name is not available when it turns blue.

In the table below, reserved words have been linked to the function of the same name, if one exists. If no link exists for a word, it is either because there was more than one association for the word or the word is for internal usage only.

abs	export	maketext	setreportdata
accept	false	map	setub
add	file	max	setvalue
adjlba	file_revisions	maxocc	setvalueset
adjlbi	fileconcat	maxvalue	setvaluesets
adjuba	filecopy	mean	show
adjubi	filecreate	meta	showarray
advance	filedelete	min	showocc
alias	fileempty	minvalue	skip
all	fileexist	missing	smean
alpha	filename	modify	soccurs
and	fileread	move	sort
array	filerename	multiple	special
ascending	filesize	next	specialvalues
ask	filetime	nmembers	specific
audio	filewrite	noauto	sql
average	find	nobreak	sqlquery
barcode	for	noccurs	sqrt
box	forcase	noconfirm	stable
break	freq	nofreq	startswith
by	frequency	noinput	stat
case	function	noprint	statistics
case_id	geometry	not	stop
cases	getbluetoothname	notappl	string
cell	getbuffer	notes	strip
changekeyboard	getcapturetype	numeric	stub
clear	getcaselabel	occ	subtable
close	getdeck	onfocus	sum
cmcode	getdeviceid	onoccchange	summary
column	getimage	open	swap
compare	getlabel	optional	sync
compress	getlanguage	or	sync_history
concat	getnote	outofrange	syncapp
config	getocclabel	page	syncconnect
confirm	getoperatorid	paradata	syncdata
connection	getorientation	path	syncdisconnect
continue	getos	pathconcat	syncfile
count	getproperty	pathname	syncmessage
countcases	getrecord	pff	syncparadata
countnonspecial	getsymbol	pos	syncserver
countvalid	getusername	poschar	synctime
crosstab	getvalue	postcalc	sysdate
curocc	getvaluealpha	postproc	sysparm

currentkey	getvaluelabel	preproc	systemapp
dateadd	getvaluenumeric	proc	systime
datediff	global	prompt	table
datevalid	gps	protect	tablogic
decompress	group	publishdate	tally
default	has	putdeck	tblcol
delcase	hash	putnote	tbllay
delete	hashmap	random	tblmed
demode	hideocc	randomin	tblrow
denom	high	randomizevs	tblsum
descending	highest	rec_name	then
diagnostics	highlighted	rec_type	timestamp
dircreate	hotdeck	recode	timestring
dirdelete	htmldialog	reenter	title
direxist	if	ref	to
dirlist	image	refused	tolower
disjoint	impute	regexmatch	tonumber
display	in	relation	totocc
do	inadvance	replace	toupper
document	inc	report	tr
edit	include	retrieve	trace
editnote	insert	round	true
else	int	row	unit
elseif	intervals	save	universe
encode	invalueset	savepartial	until
end	invoke	savesetting	update
endbox	ischecked	seed	userbar
endcase	ispartial	seek	using
enddo	isverified	seekmax	uuid
endfor	item	seekmin	valueset
endgroup	itemlist	selcase	varying
endif	key	select	vector_clock
endlevel	keylist	set	verify
endlogic	killfocus	setbluetoothname	view
endrecode	layer	setcapturepos	visualvalue
endsect	length	setcapturetype	vset
endunit	level	setcaselabel	warning
endwhen	linked	setfile	weight
ensure	list	setfont	weighted
enter	loadcase	setlanguage	when
errmsg	loadsetting	setlb	where
exclude	locate	setocclabel	while
execpff	log	setoperatorid	write
execsystem	logtext	setorientation	writecase
exit	low	setoutput	xtab
exp	lowers	setproperty	

See also: Alphabetical List of Functions and Statements

Deprecated Features

As part of a process to simplify and make the CSPro programming language more efficient, some features are going to be deprecated. In future versions, these features may no longer be supported and in fact may be removed from CSPro altogether. When compiling code, you will see deprecation warnings if you are using these features. To turn off these warnings, use the **Deprecation Warnings** choices on the **Options** menu.

Feature	Version Removed	Resolution
alpha		Use `string` variables.
box		Use `recode`.
changekeyboard		Use the `setproperty` function with the argument `"Keyboa`
demenu	CSPro 7.6	*
display		Use `errmsg` with the `case` keyword.
endsect		Use `endgroup`.
exec	CSPro 7.0	Use the file... series of functions, or the `loadsetting` and `sa`
getcapturetype		Use the `getproperty` function with the argument `"Captur`
impute / vset		Use the `valueset` option instead of `vset`.
ioerror	CSPro 7.0	Check the return value of the `setfile` function.
report	CSPro 7.7	Use the updated Templated Reports.
set array save	CSPro 7.6	Use the SaveArray PFF attribute to specify the name of the save
set attributes assisted		Set the field's capture type using the Field Properties dialog or v
set attributes protect/native		Use the `protect` function.
set attributes refresh	CSPro 7.1	*
set attributes return/autoskip		Use the `setproperty` function with the argument `"UseEnt`
set behavior canenter		Use the `setproperty` function with the argument `"CanEnt`
set behavior specialvalueszero		Use the `setproperty` function with the argument `"Specia`
set file	CSPro 7.3	Use the `setfile` function.
set format	CSPro 7.2	*
set implicit	CSPro 7.2	Use the `ensure` statement to conditionally declare variables.
set language	CSPro 7.3	Use the `setlanguage` function.
set path	CSPro 7.3	Use the Data Entry Options dialog to choose operator or systen
setcapturepos		Use the `setproperty` function with the argument `"Captur`
setcapturetype		Use the `setproperty` function with the argument `"Captur`
setreportdata	CSPro 7.7	Use the updated Templated Reports.
setvalueset with arrays		Use the `valueset` object to construct dynamic value sets.
sync		Use the `syncconnect`, `syncfile`, and `syncdisconnect`
writeform	CSPro 7.3	Consider writing out the contents of the form to a HTML file.
XML export	CSPro 7.6	*

** These features do not have workarounds. If you think that you need this functionality, email cspro@lists.census.gov and describe why this feature should not be deprecated.*

Declaration Statements

Numeric Statement

Format:
```
numeric var1[, var2[..., var-n]];
```

Description:

The **numeric** statement declares temporary numeric variables used only in this application. They will not be saved to a data file defined by a dictionary. A numeric variable is an integer or decimal number significant to 15 digits.

Local variables, with a limited scope, can also be declared in functions and PROCs. The variable will only exist and be accessible from within the function or PROC where it is declared. (In the case of a PROC, it is available in all events after it is declared. For example, a numeric declared in the preproc will still be accessible in the postproc.) A local variable cannot have the same name as a globally declared variable.

Example:
```
PROC GLOBAL
  numeric NumOfKids;

PROC CHILDREN
  NumOfKids = NumOfKids + 1;
```

See also: Array Statement, Alpha Statement, String Statement, Ensure Statement

String Statement

Format:
```
string var-1[, var-2[..., var-n]];
```

Description:

The **string** statement is used to define alphanumeric variables temporarily used in the application. It is similar to the **alpha** statement but, unlike variables created with that statement, variables created as strings have no fixed length. Strings can be used anywhere an alpha variable can be used. (Strings were only introduced starting in CSPro 6.0 so much of the documentation refers to alpha variables.) Strings start out with a length of 0 and then automatically resize to match whatever is assigned to it. This feature make strings the preferred text variable to use, unless the length of the string is needed for formatting and calculation purposes.

Examples:

```
PROC GLOBAL

alpha (10) myAlpha;
string myString;

PROC EXAMPLE

    // myString starts out empty, with length 0

    myString = "This is a test." // myString's length is now 15

    myAlpha = "Hello";
    myString = myAlpha; // myString's length is now 10: "Hello     "

    myString = "Hello"; // myString's length is now 5: "Hello"
```

See also: Alpha Statement, Array Statement, Numeric Statement

Config Statement

Format

```
config variable1[,...,variableN];
```

Description

The `config` statement declares a `string` variable. When the application begins, the variable's value is automatically set to a value defined in the configuration settings. If there is no value defined, the compiler will issue a warning message and the string variable will be blank. The statement must be coded in the `PROC GLOBAL` section of the application.

You can use a `config` variable anywhere in logic as you would a `string` variable.

Example

```
PROC GLOBAL

config serverUrl;

PROC EXAMPLE

    syncconnect(CSWeb,serverUrl);
```

See also: User and Configuration Settings, String Statement

Alpha Statement

Feature Upgrade: *Since CSPro 6.0, you can now use variable-length strings. You are encouraged to use strings instead of fixed-width alpha variables whenever possible.*

Format:

```
alpha [(len)] var-1[, var-2[..., var-n]];
```

Description:

The **alpha** statement is used to define alphanumeric variables used in the application. The **len** is the number of characters in the variable. The **len** applies to all variables which follow in the same statement. If no **len** is given, 16 is assumed. The maximum string length that can be declared is 8,192. If you attempt to assign to an alpha variable a string that is longer than the variable's size, the string will be truncated from the right. Conversely, it you assign a string that is shorter than the variable's size, the trailing character positions will be blank-filled.

The following two examples, using the declaration of x below, should clarify this point:

Example 1:

```
PROC GLOBAL
  alpha(10) x,y;

PROC A1
  x = "hi mom";

     x will now equal "hi mom    "
                      1234567890

  x = "good night, mom";
```

```
x will now equal "good night"
                  1234567890
```

Example 2:

```
PROC GLOBAL
  alpha (3) reply;
  alpha flag;

PROC Q5
if q5 = 1 then
  reply = "Yes";
  flag = "Y";
endif;
```

If the user attempts to assign the string "Not reported" to the variable "reply," CSPro would place the letters "Not" in the variable and drop the remaining characters of the string.

See also: String Statement, Array Statement, Numeric Statement,

Relation Statement

Format:

relation *relation-name primary* **to** *secondary-1 method*
[**to** *secondary-2 method*] ... [**to** *secondary-n method*];

where *method* is

parallel | **linked by** *arith-exp* | **where** *condition*

Description:

The **relation** statement allows you define additional relations beyond those defined in the data dictionary.

The *relation-name* is a unique CSPro name which contains only letters, numbers, or the underscore ('_') character. It must begin with a letter.

The *primary* is the name of a multiply occurring record or item. Items defined as *secondary* are linked to the *primary* by the *method* specified.

The *secondary* is the name of a multiply occurring record or item which is linked to the *primary*.

The *method* type is specified by one of the keywords parallel, linked by, or where.

In the **parallel** method corresponding occurrences of the primary record or item and secondary record or item are linked, that is first occurrences are linked, second occurrences are linked and so on.

In the **linked by** method the value of the arithmetic expression containing values from one record item is a pointer to the occurrence in the other record or item.

In the **where** method the value of an item on the primary record is compared to the value of an item on the secondary record. If the values are equal, the records are linked.

Example 1:

```
PROC GLOBAL
  relation PERSON POP1 to POP2 parallel
      to POP3 parallel;
```

Example 2:

```
PROC GLOBAL
  relation MOTHER-CHILD CHILD to MOTHER linked by MOTHER_LINE;
```

Example 3:

```
PROC GLOBAL
  relation MOTHER-ALL PERSON to MOTHER
      where PERSON_LINE = MOTHER_LINE;
```

Function Named Arguments

Some CSPro functions have optional arguments that can only be specified by supplying the name of the parameter, followed by the `:=` operator, and then the argument. The general form is:

```
function_name(required_arguments, parameter_name1 := argument1, ...);
```

For example, the `valueset.add` function allows you to specify the name of an image to associate with a value:

```
flowers_vs.add("Rose", 1, image := "red-rose.jpg");
```

Multiple named arguments can be supplied in any order. Both of these statements are the same:

```
flowers_vs.add("Rose", 1, image := "red-rose.jpg", textColor := "#ff0000");
flowers_vs.add("Rose", 1, textColor := "#ff0000", image := "red-rose.jpg");
```

For functions that use named arguments, the arguments that can be specified will be documented in the help page for that function.

Function Statement

Format

```
function return_type function_name(parameter1_type parameter1_name, ...,
                                   parameterN_type parameterN_name)

    // statements

    function_name = return_value;

end;
```

Description

The `function` statement defines a user-defined function with the name ***function_name***. Once defined, the function can be called in other user-defined functions or in procedures throughout your application.

Numeric expressions, string expressions, objects (such as arrays and file handlers), and function pointers can be passed to a user-defined function as arguments. These parameters are defined by specifying the variable type (e.g., ***parameter1_type***) and the name of the variable (e.g., ***parameter1_name***). If no type is specified, the variable will be considered numeric. Preceding the parameter type with the `optional` keyword marks the parameter as an optional parameter for which a caller does not need to supply a corresponding argument.

The names used in the parameter list of a function are local to the function. They may not be the same as names that are defined in any dictionary or in `PROC GLOBAL`, but the names can be reused in other functions or procedures.

Numeric, string, and alphanumeric variables are local to the function. That is, if a variable is passed as an argument, its value in the rest of the application will not be changed by actions within the function (this is called "pass by value"). On the other hand, objects (such as arrays and file handlers) passed as arguments refer to the source variable and interactions on the variable affect the source variable (this is called "pass by reference"). If you want to pass a numeric or string variable by reference, you can use the `ref` keyword to signify that changes made in the function should affect the source variable.

Other than arrays, parameters are defined the same way you would in other parts of logic. However, with arrays, you do not define the size of the array because the array will match the size of the array passed as an argument. By default, an array is one-dimensional. To specify more than one dimension, use parentheses with the number of dimensions specified using commas. For example, a three-dimensional array must include `(,,)` after the parameter name. Within the function, the function `length` returns the dimension sizes of the passed array.

Functions always return a value, either a numeric, alphanumeric, or string value. If ***return_type*** is not specified, the function will by default return a numeric value. To assign the function's return value, assign a value to ***function_name***. If no return value is specified, the value `default` or a blank string is returned, depending on the return type. The return value can also be specified using the `exit` statement.

When using functions in logic, CSPro will check that the function call is valid at during compilation. However, advanced users may find it useful to program function calls where the function name is only known at runtime. You can use the `invoke` function to execute such functions using runtime binding.

Example 1

```
PROC GLOBAL

function CompareValues(numeric value1, numeric value2)

    if value1 <> value2 then
        errmsg("The values entered, %d and %d, are not equal.", value1, value2);
        reenter;
    endif;

end;

PROC BIRTH_DATE

    CompareValues(BIRTH_DATE, BIRTH_DATE_FROM_PREVIOUS_ROUND);
```

Example 2

```
PROC GLOBAL

function string GetNameOfFirstFemale()

    numeric female_index = seek(SEX = 2);

    if female_index > 0 then
        GetNameOfFirstFemale = NAME(female_index);
    endif;

end;
```

Example 3

```
PROC GLOBAL

function numeric SumList(list numeric numeric_list)

    numeric summed_value = 0;

    do numeric counter = 1 while counter <= length(numeric_list)
        inc(summed_value, numeric_list(counter));
    enddo;

    SumList = summed_value;

end;
```

Example 4

```
PROC GLOBAL

function string CapitalizeFirstLetters(string text)

    do numeric counter = 1 while counter <= length(text)

        if counter = 1 or text[( counter - 1 ):1] = " " then
            text[counter:1] = toupper(text[counter:1]);
        endif;

    enddo;

    CapitalizeFirstLetters = text;

end;
```

See also: User-Defined Functions, Optional Function Parameters, Passing Function Arguments by Reference, Additional Examples of User-Defined Functions, Exit Statement

Optional Function Parameters

Format

```
function FunctionName(optional numeric parameter1, optional string parameter2)

function FunctionName(numeric parameter1 = default_number, string parameter2 = "default string")
```

Description

User-defined functions can have optional parameters, which you can specify in two ways. The first way is to use the `optional` keyword before the parameter type. This works with all variable types (numerics, strings, and objects) except for arrays, which cannot be used as optional parameters. The second way is by using an = sign when specifying the parameter. This works for numeric and string parameters only.

Using the Optional Keyword

```
function MyFunction(optional numeric numeric_value, optional string string_value)
```

When you call the function, you do not need to supply arguments for optional parameters. If not otherwise defined with the = sign, a number will be set to `notappl` and a string will be set to a blank string. Any object (such as a hashmap, list, etc.) will be set to its default state (an empty hashmap, a blank list, etc.). Once you specify that a parameter is optional, then all parameters after it must also be optional.

You can call the function like this:

```
// both arguments specified
MyFunction(3, "ABC");

// one argument specified (string_value set to "")
MyFunction(3);

// no arguments specified (numeric_value set to notappl, string_value set to "")
MyFunction();
```

Using an = Sign

Using an = sign is for numeric and string parameters only and cannot be used for objects. It is useful for assigning a numeric constant or string literal as the default value:

```
function MyFunction(optional numeric numeric_value = 30, optional string string_value = "Hello")
```

When using the = sign, you do not need to use the `optional` keyword, but you can specify it if you want.

Example 1

```
PROC GLOBAL

function string GetName(optional numeric person_occurrence)

    // if the argument for the occurrence number (person_occurrence) is not passed
    // to the function, person_occurrence will be notappl and the function will
    // return the name from the current record; otherwise, person_occurrence will
    // be defined and the function will return the name for the specified record

    if person_occurrence = notappl then
        person_occurrence = curocc(POPULATION_RECORD);
    endif;

    exit strip(PERSON_NAME(person_occurrence));

end;


PROC AGE

    errmsg("%s is %d years younger than the head of household (%s)",
        GetName(), AGE(1) - AGE, GetName(1));
```

Example 2

```
PROC GLOBAL

// This function calculates the age based on the birth date and the enumeration date.
// If the birth date is not passed to the function or is not set, the function returns
// a calculated age of 998 which is used to indicate that the age could not be calculated.
// If the enumeration date is not passed to the function, the function calculates the
// date based on the default date of March 11, 2020; otherwise it uses the date passed
// as a function argument.

function numeric CalculateAge(optional numeric birth_date, optional numeric enumeration_date = 2
0200311)

    if special(birth_date) then
        exit 998;
    else
        exit datediff(birth_date, enumeration_date, "y");
    endif;

end;

// ...

DOB = 19520218;
ACTUAL_ENUMERATION_DATE = 20200401;

errmsg("%d", CalculateAge(DOB, ACTUAL_ENUMERATION_DATE)); // 68
errmsg("%d", CalculateAge(DOB));                          // 67
errmsg("%d", CalculateAge());                             // 998
```

See also: User-Defined Functions, Function Statement

Passing Function Arguments by Reference

Format

```
function FunctionName(numeric function_variable)
    function_variable = 500;
end;

// using ref, any changes made to function_variable
// in the function will be assigned to MY_VALUE
FunctionName(ref MY_VALUE);
```

Description

By default, numeric and string variables that are passed as an argument to a function are not changed by actions within the function. This is called "pass by value." On the other hand, objects (such as arrays and file handlers) passed as arguments refer to the source variable and interactions on the variable within the function affect the source variable. This is called "pass by reference."

If you want to pass a numeric or string variable by reference, you can use the `ref` keyword to signify that changes made in the function should affect the source variable. You cannot use `ref` with expressions, only with numeric and string variables (such as dictionary items, array cells, list cells, etc.).

Example 1

```
function MyFunc(numeric numeric_value)
    numeric_value = 999;
end;

// ...

numeric test_value = 100;

// the value of test_value (100) is passed (copied) to the function
// and the function has no effect on the contents of test_value
MyFunc(test_value);
errmsg("%d", test_value); // pass by value, prints 100

// by reference (ref) indicates the variable location itself is passed
// to the function; performing any operation on the variable inside the
// function is the same as performing that operation outside of the function
MyFunc(ref test_value);
errmsg("%d", test_value); // pass by reference, prints 999
```

Example 2

```
function numeric ReadGPS(optional numeric latitude, optional numeric longitude)

    numeric successful_reading = false;

    if gps(open) then

        // attempt to read the GPS for up to two minutes
        if gps(read, 120) then
            successful_reading = true;
            latitude = gps(latitude);
            longitude = gps(longitude);
        endif;

        gps(close);

    endif;

    exit successful_reading;

end;

// ...

// because HH_LATITUDE and HH_LONGITUDE are passed by reference (ref),
// they will contain the values as updated in the function
if not ReadGps(ref HH_LATITUDE, ref HH_LONGITUDE) then
    errmsg("GPS reading error");
endif;
```

See also: User-Defined Functions, Function Statement

Additional Examples of User-Defined Functions

These are examples of more advanced ways to work with user-defined functions in CSPro.

Recursive Functions

```
function numeric NumberDescendantsInHH(numeric ptrPerson)

    numeric cntDescendants;

    do numeric hhCtr = 1 while hhCtr <= totocc(PERSON_REC)

        if LN_FATHER(hhCtr) = ptrPerson or LN_MOTHER(hhCtr) = ptrPerson then
            inc(cntDescendants,NumberDescendantsInHH(hhCtr));
        endif;

    enddo;

    NumberDescendantsInHH = cntDescendants;

end;
```

Function Pointers

(The `invoke` function may provide an alternative to some uses of function pointers.)

```
function numeric FilteredCountInHH(function filter(numeric))

    numeric cnt;

    do numeric hhCtr = 1 while hhCtr <= totocc(PERSON_REC)

        if filter(hhCtr) then
            inc(cnt);
        endif;

    enddo;

    FilteredCountInHH = cnt;

end;

function numeric IsTeenager(numeric ptrPerson)
    IsTeenager = ( AGE(ptrPerson) in 13:19 );
end;

function numeric IsFertileWoman(numeric ptrPerson)
    IsFertileWoman = ( SEX(ptrPerson) = 2 and AGE(ptrPerson) in 15:49 );
end;


PROC ANALYSIS

    numeric cntTeenagers = FilteredCountInHH(IsTeenager);
    numeric cntFertileWomen = FilteredCountInHH(IsFertileWoman);
```

See also: Function Statement, Optional Function Parameters, Passing Function Arguments by Reference

Logic Objects and Dot Notation

A CSPro logic object is a variable that is similar to the objects that exist in other programming languages that are "object-oriented." CSPro logic is not fully object-oriented, but some variables have functions (sometimes called "methods") that can be called by using both the variable name as well as the function name.

```
variable_name.function_name();
```

The following objects have functions that can be called on the object:

- array
- audio
- document
- file
- freq
- geometry
- hashmap
- image
- list
- map
- pff
- report
- SystemApp
- valueset

A list of available functions can be found in each of those summary pages.

One way to understand dot notation is to think of it as a shorthand way to specify what variable should be used when performing an operation. For example, here are two ways to work through the elements of a list:

```
list string household_members;

// CSPro 7.2 and earlier:
do numeric counter = 1 while counter <= length(household_members)

// CSPro 7.3 and later:
do numeric counter = 1 while counter <= household_members.length()
```

Unlike other CSPro functions, the functions that are associated with an object will not show as blue and they are not

reserved words.

Using a dot specifies that you are accessing a function or variable (property) of the object. For example, a value set object contains two internal variables, codes and labels, both of which are lists, that can be accessed as in:

```
list string province_labels = PROVINCE_VS.labels;
numeric selected_province = province_labels.show("Select a province");
```

This could also be written using multiple dots:

```
numeric selected_province = PROVINCE_VS.labels.show("Select a province");
```

See also: Variables

Array Object

Array Statement

Format

```
array [array_type] array_name(dimension1[, ..., dimensionN]) [save];
```

Description

The ***array*** statement creates an array with the name ***array_name***. Only one array at a time can be declared with the array statement. The array name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare arrays in `PROC GLOBAL` or locally in functions or procedures.

Arrays can be numeric, alphanumeric or string. By default an array is numeric, but the type can be modified by specifying the ***array_type***. If creating an alphanumeric array, the length of each array element can be specified by definining the ***array_type*** as follows:

```
alpha [(array_length)]
```

If ***array_length*** is not specified, each element will be 16 characters.

The size of each dimension is specified by supplying a constant positive numeric value, ***dimension1*** to ***dimensionN***. An array must have at least one dimension. CSPro supports arrays of an unlimited number of dimensions. A previously defined numeric value can also be used to specify the dimension size. The `array.length` function can be used to query the size of a dimension.

The initial values of the elements of an array can be set when declaring the array by listing each value, separated by a comma. If some values are defined and then followed by `...`, that set of values will be used over and over until the entire array has been initialized.

With a numeric array, each element starts with the value 0. For alphanumeric and string arrays, each element starts as a blank string. If using a numeric saved array, the initial array contents are `default`.

Saved Arrays

The optional keyword `save` indicates that the array values should be saved to a file and loaded from that file when the program is run again. This allows you to maintain the values of the arrays across multiple runs of the same program. When one or more arrays in the program are marked with `save`, the first time the application is run, a saved array file is created and the values of the arrays are written to the file at the end of program execution. On consecutive runs of the program, the initial values of the arrays are read in from the file. This is particularly useful for setting the initial values of hot decks. In this scenario, the program is run twice. The first run fills the hot deck and saves the hot deck array to the file. The second run loads the values saved from the first run and uses them as the initial values for the hot deck for imputation. See Initialize Hot Decks in Program Logic for more information.

All arrays marked with `save` in the application are written to the same file. By default this file has the same name as the application but with a .sva file extension appended to it (for example, MyApplication.bch.sva). You can modify the name of the saved array file by using the PFF SaveArray attribute.

Example 1 (Numeric)

```
PROC GLOBAL

array numeric MinimumAgeMarriage(2) = 15, // male
                                      12; // female

PROC MARRIAGE_AGE

    if MARRIAGE_AGE < MinimumAgeMarriage(SEX) then
        errmsg("The minimum age of marriage is %d", MinimumAgeMarriage(SEX));
        reenter;
    endif;
```

Example 2 (String)

```
PROC GLOBAL

array string Months(12) = "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec";

function string FormatDate()
    FormatDate = maketext("%d %s %d", sysdate("DD"), Months(sysdate("MM")), sysdate("YYYY"));
end;
```

Example 3 (Assignment)

```
numeric NumberProvinces = 12;
array numeric SexCountsByProvince(NumberProvinces, 2); // province by sex

PROC SEX

    inc(SexCountsByProvince(PROVINCE, SEX));

PROC DISPLAY_COUNTS

    do numeric counter = 1 while counter <= SexCountsByProvince.length(1)
        errmsg("Province %d has %d males and %d females", counter,
            SexCountsByProvince(counter, 1), SexCountsByProvince(counter, 2));
    enddo;
```

See also: Array Object, HashMap Object, List Object, Numeric Statement, Alpha Statement, String Statement, Saved Arrays File (.sva)

Array.Length Function

Format

```
i = array_name.length([dimension]);
```

Description

The `array.length` function returns the size of an array dimension. When applied to a one-dimensional array, the ***dimension*** does not need to be specified.

Return Value

The function returns the length of the array dimension.

Example

```
array string usmca_countries(3) = "Canada", "Mexico", "United States";
usmca_countries.length(); // returns 3;

// ...

array AgeSexDeckArray(AGE_VS, SEX_VS);

do numeric age_counter = 1 while age_counter <= AgeSexDeckArray.length(1)
    do numeric sex_counter = 1 while sex_counter <= AgeSexDeckArray.length(2)
        // ...
    enddo;
enddo;
```

See also: Array Object, Length Function

Array.Clear Function

Format

```
b = array_name.clear();
```

Description

The `array.clear` function resets all values of an array to their default value. For numeric arrays, this is 0, unless the array is a saved array, in which case the value is `default`. For alphanumeric and string arrays, the default value is a blank string.

Return Value

The function returns a logical value of 1 (true).

Example

```
array string HouseholdMemberNames(100);

// clear the array of household member names so that
// any previously stored names are removed
HouseholdMemberNames.clear();

do numeric counter = 1 while counter <= count(NAME)
    HouseholdMemberNames(counter) = NAME(counter);
enddo;

numeric selection = showarray(HouseholdMemberNames, title("Select a Household Member"));
```

See also: Array Object

Audio Object

Audio Statement

Format

```
audio audio_name;
```

Description

The `audio` statement creates an audio object with the name ***audio_name***. The audio name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare audio objects globally in `PROC GLOBAL` or locally in functions or procedures.

Audio recording and playback are currently only supported on Android.

Example

```
PROC AUDIO_QUESTION

onfocus

// Variable to store the recording
audio recording;

// Save recording to a file that includes case id-items to differentiate it from
// audio saved in other cases
string nameOfAudioFile = maketext("%v%v%vAUDIO_QUESTION.m4a", PROVINCE, DISTRICT, EA);

// If no audio is recorded for this question then the only option is to record,
// otherwise allow user to re-record, play or clear recording before continuing
valueset vs;
if fileexist(nameOfAudioFile) and recording.load(nameOfAudioFile) then
    vs.add("Re-record", 1);
    vs.add("Play recording", 2);
    vs.add("Clear recording", 3);
    vs.add("Next question", 4);
else
    vs.add("Record", 1);
endif;
setvalueset($, vs);

postproc

if $ = 1 then
    // Record/re-record
    recording.clear();
    string message = "Record the respondents answer to this question";
    numeric seconds = recording.recordinteractive(message);
    if seconds > 0 then
        recording.save(nameOfAudioFile);
    else
        errmsg("No audio recorded. Please try again");
    endif;
    reenter;
elseif $ = 2 then
    // Play back recording
    recording.play();
    reenter;
elseif $ = 3 then
    // Clear the recording
    filedelete(nameOfAudioFile);
    reenter;
elseif $ = 4 then
    // Continue to next question
endif;
```

See also: Audio Object

Audio.Load Function

Format

```
b = audio_name.load(audio_filename);
```

Description

The `audio.load` function reads the audio file named ***audio_filename*** and stores the audio data contained in the file in an audio object. The loaded audio can then be played back using `audio.play` or combined with other audio using `audio.concat`.

CSPro can load and play most audio files in any of the following formats: m4a, mp4, mp3, wav, 3gp, flac, ogg, mkv.

However, only m4a files (AAC encoded audio in an MPEG-4 container) can be used with the functions `audio.concat`, `audio.length`, `audio.record`, and `audio.recordInteractive`.

Return Value

The function returns a logical value of 1 (true) if the file was loaded and 0 (false) if the file could not be loaded.

Example

```
audio recording;
if recording.load("myrecording.m4a") then
    recording.play();
endif;
```

See also: Audio Object, Audio.Save Function, Audio.Play Function

Audio.Save Function

Format

```
b = audio_name.save(audio_filename);
```

Description

The `audio.save` function saves the audio recording in an audio object to the file specified by ***audio_filename***. If the audio file already exists, the recording will be appended to the existing audio.

Audio recorded using the functions `audio.record` and `audio.recordInteractive` is stored in m4a format (AAC encoded in an MPEG-4 container). Audio in this format should be saved with the extension .m4a so that it will be recognized by media players.

Return Value

The function returns a logical value of 1 (true) if the file was successfully saved or 0 (false) otherwise.

Example

```
PROC GLOBAL
audio recordingQ1;

PROC Q1
preproc
recordingQ1.record();

postproc
recordingQ1.stop();
string audioFilename = maketext("recording%v.m4a", HOUSEHOLD_ID);
recordingQ1.save(audioFilename);
```

See also: Audio Object, Audio.Load Function

Audio.Play Function

Format

```
b = audio_name.play([message]);
```

Description

The `audio.play` function launches an audio player to play the audio in an audio object. Before calling this function, the audio object must contain audio data recorded using `audio.recordInteractive`, `audio.record`, or loaded using `audio.load`. If the optional argument ***message*** is provided, the message text is displayed over the audio player.

CSPro can load and play most audio files in any of the following formats: m4a, mp4, mp3, wav, 3gp, flac, ogg, mkv. However, only m4a files (AAC encoded audio in an MPEG-4 container) can be used with the functions `audio.concat`, `audio.length`, `audio.record`, and `audio.recordInteractive`.

Return Value

The function returns a logical value of 1 (true) if the audio can be played and a logical value of 0 (false) otherwise.

Example

```
audio recording;
if recording.load("myrecording.m4a") then
    recording.play();
endif;
```

See also: Audio Object, Audio.Load Function, Audio.Record Function, Audio.RecordInteractive Function

Audio.RecordInteractive Function

Format

```
d = audio_name.recordInteractive([message]);
```

Description

The `audio.recordInteractive` function launches an interactive audio recorder that stores a recording to an audio object. When the function is called, the audio recorder screen is displayed. It contains record, save, and clear buttons that control the audio recording. If the optional string argument ***message*** is specified, the message will be displayed at the top of the recorder screen. The `audio.recordInteractive` function does not return until the user has finished recording and taps the save button. This is useful for synchronous recording, for example to have the interviewer record the response to a single question. To record the audio in the background while the user continues to interact with the survey, use the function `audio.record`.

The recorded audio is stored in the audio object and can be played back using `audio.play` or saved to a file using `audio.save`. If the audio object already contains an audio recording, the recorded audio will be appended to the existing recording. To replace the existing recording rather than adding to it, call the `audio.clear` function before calling `audio.recordInteractive`.

Recorded audio is stored in m4a format (AAC encoded in an MPEG-4 container). Files should be saved with the extension .m4a so that they will be recognized by media players.

Audio recording is only supported on Android. This function will do nothing on Windows.

Return Value

The function returns the length of the recorded audio in seconds (and fractional seconds) or `default` if there was an error during recording.

Example

```
PROC AUDIO_QUESTION

onfocus

// Variable to store the recording
audio recording;

// Save recording to a file that includes case id-items to differentiate it from
// audio saved in other cases
string nameOfAudioFile = maketext("%v%v%vAUDIO_QUESTION.m4a", PROVINCE, DISTRICT, EA);

// If no audio is recorded for this question then the only option is to record,
// otherwise allow user to re-record, play or clear recording before continuing
valueset vs;
if fileexist(nameOfAudioFile) and recording.load(nameOfAudioFile) then
    vs.add("Re-record", 1);
    vs.add("Play recording", 2);
    vs.add("Clear recording", 3);
    vs.add("Next question", 4);
else
    vs.add("Record", 1);
endif;
setvalueset($, vs);

postproc

if $ = 1 then
    // Record/re-record
    recording.clear();
    string message = "Record the respondents answer to this question";
    numeric seconds = recording.recordinteractive(message);
    if seconds > 0 then
        recording.save(nameOfAudioFile);
    else
        errmsg("No audio recorded. Please try again");
    endif;
    reenter;
elseif $ = 2 then
    // Play back recording
    recording.play();
    reenter;
elseif $ = 3 then
    // Clear the recording
    filedelete(nameOfAudioFile);
    reenter;
elseif $ = 4 then
    // Continue to next question
endif;
```

See also: Audio Object, Audio.Record Function

Audio.Record Function

Format

```
b = audio_name.record([seconds]);
```

Description

The `audio.record` function starts recording audio in the background and stores the recorded audio in an audio object. The call to record begins audio recording and returns immediately. Recording continues until the `audio.stop` function is called. If the optional parameter ***seconds*** is specified, recording will stop after the specified number of seconds have elapsed—or `audio.stop`—is called, whichever action comes first.

While the audio is recording, CSPro continues to run and the user is able to enter responses and move around the questionnaire as usual. This is useful if you wish to record portions of the interview for audit purposes. If instead you wish to record audio synchronously, for example to record the response to a single question, use the `audio.recordInteractive` function instead.

The recorded audio is stored in the audio object and can be played back using `audio.play` or saved to a file using `audio.save`. If the audio object already contains an audio recording, the recorded audio will be appended to the existing recording. To replace the existing recording rather than adding to it, call the `audio.clear` function before calling `audio.record`.

Recorded audio is stored in m4a format (AAC encoded in an MPEG-4 container). Files should be saved with the extension .m4a so that they will be recognized by media players.

Audio recording is only supported on Android. This function will do nothing on Windows.

Return Value

The function returns a logical value of 1 (true) if the recording is started successfully and 0 (false) otherwise.

Example

```
logic>
PROC GLOBAL
audio recordingQ1;

PROC Q1
preproc
recordingQ1.record();

postproc
recordingQ1.stop();
string audioFilename = maketext("recording%v.m4a", HOUSEHOLD_ID);
recordingQ1.save(audioFilename);
```

See also: Audio Object, Audio.Stop Function, Audio.RecordInteractive Function, Audio.Clear Function

Audio.Stop Function

Format

```
d = audio_name.stop();
```

Description

The `audio.stop` function ends the background recording session for an audio object. Background recording is started with the function `audio.record`. The call to `audio.stop` should be made to finish the recording session started with `audio.record`. This function is **not** used for interactive recording with `audio.recordInteractive`.

Return Value

The function returns the duration of the recorded audio in seconds (and fractional seconds) or `default` if the audio failed to record.

Example

```
PROC GLOBAL
audio recordingQ1;

PROC Q1
preproc
recordingQ1.record();

postproc
recordingQ1.stop();
string audioFilename = maketext("recording%v.m4a", HOUSEHOLD_ID);
recordingQ1.save(audioFilename);
```

See also: Audio Object, Audio.Record Function

Audio.Concat Function

Format

```
b = audio_name.concat(audio_filename || audio_object);
```

Description

The `audio.concat` function appends an audio recording stored in a file specified by the argument ***audio_filename*** or an audio object specified by the argument ***audio_object*** to the current recording in an audio object. The result is a new recording that combines both recordings and is stored in the initial audio object.

The `audio.concat` function only works with audio files stored in m4a format (AAC encoded audio stored in an MPEG-4 container). This is the format of files recorded in CSPro using the `audio.record` and `audio.recordInteractive` functions. Audio files that come from sources other than CSPro will most likely not work with `audio.concat`.

Return Value

The function returns a logical value of 1 (true) if the recordings were concatenated successfully and a logical value of 0 (false) if there was an error combining the recordings.

Example 1

```
audio recording;

recording.load("recording1.m4a");
recording.concat("recording2.m4a");
recording.save("combined.m4a");
```

Example 2

```
PROC GLOBAL
audio recordingQ1;
audio recordingQ2;

PROC SURVEY_FF

PROC SURVEY_LEVEL

// at end of questionnaire combine audio recordings from Q1 and Q2 and save
audio combined;
combined = recordingQ1;
combined.concat(recordingQ2);
string audioFilename = maketext("recording%v.m4a", HOUSEHOLD_ID);
combined.save(audioFilename);

PROC Q1
preproc
recordingQ1.record();

postproc
recordingQ1.stop();

PROC Q2
preproc
recordingQ2.record();

postproc
recordingQ2.stop();
```

See also: Audio Object

Audio.Length Function

Format

```
d = audio_name.length();
```

Description

The `audio.length` function returns the length in seconds of the recording in an audio object.

The `audio.length` function only supports audio recorded as AAC encoded audio in an mp4 container (.m4a). This is the format used for audio recorded by CSPro so it will work correctly for audio recorded using the `audio.record` or `audio.recordInteractive`. The length function will return `default` for audio files recorded by other tools that use a different format such as mp3 or wav.

Return Value

The function returns the duration in seconds (and fractional seconds) of the audio recording or `default` if the duration cannot be computed.

Example

```
audio recording;
numeric seconds = recording.recordinteractive();
if seconds > 0 then
    recording.save("recording.m4a");
else
    errmsg("No audio recorded. Please try again");
    reenter;
endif;
```

See also: Audio Object

Audio.Clear Function

Format

```
b = audio_name.clear();
```

Description

The `audio.clear` function removes the current recording from an audio object making it empty.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC AUDIO_QUESTION

onfocus

// Variable to store the recording
audio recording;

// Save recording to a file that includes case id-items to differentiate it from
// audio saved in other cases
string nameOfAudioFile = maketext("%v%v%vAUDIO_QUESTION.m4a", PROVINCE, DISTRICT, EA);

// If no audio is recorded for this question then the only option is to record,
// otherwise allow user to re-record, play or clear recording before continuing
valueset vs;
if fileexist(nameOfAudioFile) and recording.load(nameOfAudioFile) then
    vs.add("Re-record", 1);
    vs.add("Play recording", 2);
    vs.add("Clear recording", 3);
    vs.add("Next question", 4);
else
    vs.add("Record", 1);
endif;
setvalueset($, vs);

postproc

if $ = 1 then
    // Record/re-record
    recording.clear();
    string message = "Record the respondents answer to this question";
    numeric seconds = recording.recordinteractive(message);
    if seconds > 0 then
        recording.save(nameOfAudioFile);
    else
        errmsg("No audio recorded. Please try again");
    endif;
    reenter;
elseif $ = 2 then
    // Play back recording
    recording.play();
    reenter;
elseif $ = 3 then
    // Clear the recording
    filedelete(nameOfAudioFile);
    reenter;
elseif $ = 4 then
    // Continue to next question
endif;
```

See also: Audio Object

Barcode and QR Codes

Barcode Namespace

When running a data entry application on an Android device, you can use the device's camera to read a barcode in several formats, including QR codes.

Functionality

Function	Description
`barcode.read`	Reads a barcode and returns its value as a string.

In addition to barcode functions accessible using logic, you can generate a QR code that can be used to install applications on mobile devices.

You can also read barcodes using the Barcode capture type.

Example

```
PROC BLOOD_SAMPLE

preproc

    if BLOOD_SAMPLE = "" then
        BLOOD_SAMPLE = barcode.read("Scan the blood sample barcode");
    endif;
```

See also: Barcode Capture Type, Multimedia Features

Barcode.Read Function

Format

```
s = barcode.read([message]);
```

Description

The `barcode.read` function starts an Android device's camera and allows the operator to scan a barcode. The barcode can be in several formats, including QR codes. An optional string expression, ***message***, is displayed on top of the camera view and can be used to display instructions to the operator.

This function only works on Android devices and returns a blank string if used on another device.

Return Value

The function returns a string value containing the barcode's value. If the operator canceled before a barcode could be successfully read, the function returns a blank string.

Example

```
PROC BLOOD_SAMPLE

preproc

    if BLOOD_SAMPLE = "" then
        BLOOD_SAMPLE = barcode.read("Scan the blood sample barcode");
    endif;
```

See also: Barcode Namespace, Barcode Capture Type

Document Object

Document Statement

Format

```
document document_name;
```

Description

The `document` statement creates a document with the name ***document_name***. The document name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare documents globally in `PROC_GLOBAL` or locally in functions or procedures.

Example

```
document manual;
manual.load("Farmers Markets Survey.pdf");
manual.view();
```

See also: Document Object

Document.Load Function

Format

```
b = document_name.load(document_filename);
```

Description

The `document.load` function reads the text or binary file given by the string expression ***document_filename*** and stores the contents in the document object. The file's entire contents are read at once and will reside in memory as long as the document object exists.

Return Value

The function returns a logical value of 1 (true) if the file was loaded and 0 (false) if the file could not be loaded.

Example

```
document manual;
manual.load("Farmers Markets Survey.pdf");
manual.view();
```

See also: Document Object, Document.Save Function

Document.Save Function

Format

```
b = document_name.save(document_filename);
```

Description

The `document.save` function writes the contents of the document object to a file specified by the string expression ***document_filename***.

Return Value

The function returns a logical value of 1 (true) if the file was saved, 0 (false) if there was an error writing the file, and `default` if the document object was empty.

Example

```
document dna_readings;
// ...
dna_readings.save("DNA Readings.dnax");
```

See also: Document Object, Document.Load Function

Document.View Function

Format

```
b = document_name.view();
```

Description

The `document.view` function displays the contents of the file stored in a document object. The function is equivalent to calling:

```
view(document_name);
```

If the document contains an image or HTML, the content will be shown in an embedded window. For other files, an external application is launched to display the file using the system's default viewer for that file type.

Return Value

The function returns a logical value of 1 (true) if the file was displayed, 0 (false) if there was an error viewing the file, and `default` if the document object was empty.

Example

```
document manual;
manual.load("Farmers Markets Survey.pdf");
manual.view();
```

See also: Document Object

Document.Clear Function

Format

```
b = document_name.clear();
```

Description

The `document.clear` function clears the contents of the document object, leaving the document empty.

Return Value

The function returns a logical value of 1 (true).

See also: Document Object

File Object

File Statement

Format

```
file file_handler1[, ..., file_handlerN];
```

Description

The `file` statement creates a file handler that is used to read from and write to text files that are not not associated with data dictionaries. ***file_handler1*** to ***file_handlerN*** are CSPro names that must be unique and must contain only letters, numbers, or the underscore character. The names must begin with a letter. You can declare files globally in `PROC GLOBAL` or locally in functions or procedures.

If declared globally, the physical name of the file can be specified in the Define File Associations dialog when the application is run. The name can also be specified in a PFF file or at runtime by using the `file.open`, `open`, or `setfile` functions.

Example

```
PROC GLOBAL

file VacantHouseholdsReportFile;
```

See also: File Object

File.Open Function

Format

```
b = file_handler.open(file_name[, update | append | create]);
```

Description

The `file.open` function associates the file ***file_handler*** to a text file on the disk. The ***file_name*** argument is a string expression containing the file name of the file to be associated with the file. This function is equivalent to the `setfile` function.

Using the keyword ***update***, ***append*** or ***create*** is optional. If no keyword is used, the file is opened in ***update*** mode.

If ***update*** is used, you are positioned at the beginning of the file. If ***append*** is used, the file's contents are not changed and you are positioned at the end of the file. If ***create*** is used, all previous content in the file is removed and you are positioned at the beginning of the file.

If ***create*** or ***append*** is used and the file does not already exist, a new empty file will be created. If ***update*** is used and the file does not already exist, the function will fail and return 0.

Return Value

The function returns a logical value of 1 (true) if the new file is successfully assigned and 0 (false) otherwise.

File Example

```
file household_report_file;

household_report_file.open("Household Report.txt", create);
```

See also: File Object, File.Close Function, SetFile Function

File.Close Function

Format

```
b = file_handler.close();
```

Description

The `file.close` function closes the text file associated with ***file_handler***. You must close a file before using it in any other application. If you do not close the file explicitly, it will be closed when the CSPro application terminates. This function is equivalent to the `close` function.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
file report_file;

// ...

report_file.close();

view(filename(report_file));
```

See also: File Object, File.Open Function, Close Function

File.Read Function

Format

```
b = file_handler.read(string_variable || string_list);
```

Description

The `file.read` function reads one or more lines of text from the file associated with ***file_handler***. After the read the file pointer is positioned to the next line in the file. This function reads lines sequentially. This function is equivalent to the `fileread` function.

If a ***string_variable*** is provided as a argument, a single line of text is read and placed in the variable, which is either a temporary `string` variable or an alphanumeric dictionary item. If the dictionary item is longer than the line of text, blanks will be added at the end. If the item is shorter, the line of text will be truncated.

Alternatively, a string list can be used as an argument. In this case, all remaining lines in the file are read and stored in ***string_list***.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
file occupation_codes_file;

occupation_codes_file.open("Occupation Codes.txt");

// approach 1
string code;

while occupation_codes_file.read(code) do
    // do something with the single code
enddo;

// approach 2
list string all_codes;

occupation_codes_file.read(all_codes);
// do something with all of the codes

occupation_codes_file.close();
```

See also: File Object, File.Write Function, FileRead Function

File.Write Function

Format

```
b = file_handler.write(message⌈, argument1, ..., argumentN⌋);
```

Description

The `file.write` function writes one or more lines of text to the file associated with ***file_handler***. This function is equivalent to the `filewrite` function.

The ***message*** is either a string expression or a numeric message number that contains the text that is written to the file. If the text contains any message formatters, the optional arguments ***argument1*** to ***argumentN*** will be inserted into the text. There are some additional options for file output:

- If you want to break a line of text into two lines, place **\n** where you want the line divided.
- If you want a text line to begin on a new page, place **\f** at the beginning of the text string.
- To output **\n** or **\f** as text instead of a new line or a new page, use a double backslash (e.g., **\\n**).

Alternatively, the ***message*** can be a string list. If a list is provided, then each string contained in the list is written to the file, allowing for the output of multiple lines of text with one function call.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
file data_file_report;

data_file_report.open("Data File Report.txt", create);

list string data_file_listing;
dirlist(data_file_listing, "Data", "*.csdb");

data_file_report.write("There are %d files in the data directory:", data_file_listing.length());
data_file_report.write("");
data_file_report.write(data_file_listing);

data_file_report.close();

view(filename(data_file_report));
```

See also: File Object, File.Read Function, FileWrite Function, Message Formatting Options, Encode Function

Freq Object

Freq Statement (Named)

Format

```
freq freq_name
include(variables_to_tabulate)
[exclude(variables_not_to_tabulate)]
[disjoint]
[breakdown(length)]
[formatting_options]
;
```

Description

The `freq` statement creates a freq object with the name ***freq_name***. The freq name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare freqs globally in `PROC GLOBAL` or locally in functions or procedures.

The freq object is used to create frequencies that will be written to the frequencies file (using `freq.save`) or displayed in an embedded web browser (using `freq.view`).

Freq objects are similar to unnamed frequencies except that you control when they are tallied (using the `freq.tally` function). Because of this, the `universe` and `weight` commands, which are part of the unnamed frequencies specification, are not available. You can control the universe (using an `if` statement) and you can control the weight by passing an argument to the `freq.tally` function.

Specifying Variables to Tabulate

The `include` command is used to specify what variables should be tabulated. At least one frequency table will be created for every variable specified in the `include` list. The ***variables_to_tabulate*** is a list of variables, separated by commas, that can include:

- **Dictionary names**: dictionaries, records, and items
- **Form names**: forms, groups, and blocks
- **Logic names**: `numeric` and `string` variables

The optional `exclude` command is used to specify variables that should be removed from the inclusion list. The ***variables_not_to_tabulate*** is a list of variables as defined above. The `exclude` command is particularly useful when including records. For example, if you want to tabulate most items on a record with a couple exceptions, you might code:

```
include(PERSON_REC)
exclude(P25_RELIGION, P26_TRIBE)
```

When including or excluding a name that may contain more than one variable—dictionaries, records, forms, groups, and blocks—CSPro uses a rule to determine whether items contained in that grouping should be included in the list of variables to tabulate:

Default rule: Include the item if it has a value set; if no value set is defined, then include the item if it has length 1 - 4.

You can override the default selection by adding one or more of these flags to the `include`/`exclude` list:

Flag	Include/Exclude All...
numeric	numeric items
float	numeric items with decimals
integer	numeric items without decimals
long	numeric items without decimals and length 3 - 15
short	numeric items without decimals and length 1 - 2
alpha	alphanumeric items

These six flags will never include items that have subitems. You can use a combination of flags; for example, this would include all items from PERSON_REC with the exception of items that have subitems (though the subitems would be included).

```
include(numeric, alpha, PERSON_REC)
```

Leaving the include list empty is as if you coded the primary dictionary name. For example:

```
include()
// may be translated to:
include(CENSUS_DICT)
```

If you do not need to exclude any variables, the include command is optional. For example, these unnamed frequency statements are the same:

```
freq include(P03_SEX);
freq(P03_SEX);
```

Handling Variable Occurrences

By default, when frequencies are tallied for a multiply-occurring item, all of the occurrences of the item are tallied. For example, assuming that P03_SEX is on a record with 50 occurrences, coding this will tally all defined occurrences of P03_SEX:

```
freq include(P03_SEX);
```

If the first house is vacant, then no sex values are tallied; if the second house has three people, then three sex values are tallied; and so on.

If you would like to tally a specific occurrence, you can specify the occurrence in the include/exclude list. For example, this would create two tables, one for all sex occurrences, and one for the head's sex (assuming that the head is the first occurrence):

```
freq include(P03_SEX, P03_SEX(1));
```

You can specify occurrence values when using items or records. PERSON_REC(1), for example, would create tables for the first occurrence of the items in PERSON_REC. If you specify an occurrence, the value will be tallied regardless of whether the occurrence exists. For example, while P03_SEX would not tally vacant households, P03_SEX(1) will include tallies of blank values for vacant households.

An optional command, disjoint, is a shortcut way of indicating that a frequency table should be created for every occurrence of a variable. For example, this code would create a table for each of the occurrences of P03_SEX, resulting in 50 tables (for occurrence 1, occurrence 2, and so on until occurrence 50):

```
freq include(P03_SEX)
     disjoint;
```

When using disjoint, you can use (*) as an occurrence to specify that you would like to ignore the disjoint setting. For example, this code would create 49 tables (for the combined occurrences, for occurrence 3, occurrence 4, and so on until occurrence 50).

```
freq include(P03_SEX(*), P03_SEX)
     exclude(P03_SEX(1), P03_SEX(2))
     disjoint;
```

Alphanumeric Breakdown

The optional command `breakdown` allows you to control how alphanumeric items and string variables are tallied. A positive numeric constant, ***length***, specifies a number used to split these values before tallying. This can be useful when creating frequencies for data collected using checkboxes. For example, assuming `CHECKBOX_FIELD` occurs twice, first as "AB" and then as "BC":

```
freq include(CHECKBOX_FIELD); // results in: "AB" (1)
                              //             "BC" (1)

freq include(CHECKBOX_FIELD)  // results in:  "A" (1)
     breakdown(1);            //              "B" (2)
                              //              "C" (1)
```

Additional Commands

Optional formatting options allow you to control how the frequency tables are generated. The formatting options include the following commands: `valueset`, `distinct`, `vset`, `heading`, `stat`, `percentiles`, `nofreq`, `decimals`, `sort`, `nonetpercents`, and `pagelength`.

Example

```
PROC GLOBAL

// create a freq object that will tally the sex and age of
// the first occurrence (the head)
freq head_sex_age_freq(P03_SEX(1), P04_AGE(1));

PROC CENSUS_DICTIONARY_FF

    // save the frequency, formatting the ages using a
    // 5-year value set
    head_sex_age_freq.save()
                      valueset(P04_AGE_5YEAR_VS);

PROC QUEST

    // only tally households that are not vacant
    if count(PERSON_REC) > 0 then
        head_sex_age_freq.tally();
    endif;
```

See also: Freq Object, Freq Statement (Unnamed), Frequency Formatting Options

Frequency Formatting Options

Frequency formatting options can be specified:

- When using the unnamed frequency command.
- When declaring a named frequency.
- When calling the `freq.save` function.
- When calling the `freq.view` function.

Any options specified as part of a call to `freq.save` or `freq.view` will override options specified when declaring the named frequency.

The formatting options include the following commands: `valueset`, `distinct`, `vset`, `heading`, `stat`, `percentiles`, `nofreq`, `decimals`, `sort`, `nonetpercents`, and `pagelength`.

Value Set Specification

```
valueset(value_set_name1[, ..., value_set_nameN])
```

When creating the frequency table, CSPro tries to find a label to associate with each value in the table. By default, an item's primary value set is used for unnamed frequencies and the current value set is used for named frequencies. To instead specify a specific value set to use to lookup the labels, use the `valueset` command and supply one or more value set names. The names of the value sets must match the items that are part of the frequency.

```
distinct
```

When using the `valueset` command, the frequency table is generated showing only the labels of values. If you would like to see the values in addition to the labels, you can use the `distinct` command, which will result in each value corresponding to a row in the frequency table. Without this option, a value set range such as 0 - 4 would have resulted in all values between 0 and 4 being displayed in the row with the label "0 - 4." With this option, the table would have up to five rows ("0 | 0 - 4," "1 | 0 - 4," and so on).

```
vset
```

The `vset` command typically results in multiple frequency tables being created for each item and is generally used instead of the `valueset` or `distinct` commands. When used, a frequency table is generated using the primary value set with the distinct setting and then frequency tables are generated using each of an item's value sets with the **not** distinct setting (unless distinct was explicitly specified). For example, if an item has three value sets, four tables would be generated (value set 1 distinct, value set 1 not distinct, value set 2 not distinct, value set 3 not distinct).

Specifying a Heading

```
heading(heading_text1[, ..., heading_textN])
```

The command `heading` results in the string expressions ***heading_text1*** to ***heading_textN*** being displayed as a heading to the frequency tables. These headings are only used in HTML and text frequencies.

Generating Statistics and Percentiles

```
stat
```

The command `stat` indicates that statistics will be displayed along with the frequency table. These statistics include a count of how many unique values are part of the frequencies table. If running a numeric frequency, the statistics include values such as the minimum and maximum values, the mean, the standard deviation and variance, the mode, and the median and interpolated median.

```
percentiles(number_percentiles)
```

The `percentiles` command means that percentiles will be displayed along with the frequency table. The numeric constant ***number_percentiles*** must be a value between 2 and 20. For example, 4 will show quartiles and 10 will show deciles.

```
nofreq
```

The command `nofreq`, when used with `stat`, indicates that the frequency table should not be shown. With this combination, only the statistics (and/or percentiles) are shown. This option is only used in HTML and text frequencies.

Formatting Options

```
decimals(number_decimals)
```

The `decimals` command means that the frequency values will be displayed with the specific number of digits after the decimal mark. The numeric constant ***number_decimals*** must be a value between 1 and 5. For example, 2 will show frequency values such as 126.00 (as opposed to 126). This may be useful if you are using non-integer weights when tallying frequencies.

```
sort([ascending || descending] by valueset || code || label || freq)
```

The `sort` command allows you to control the order in which each row of the frequency table appears. The options include:

- `by valueset`: the rows will appear in the order that the values are defined in an item's value set.
- `by code`: the rows will appear in order sorted by each row's value.
- `by label`: the rows will appear in order sorted by each row's label as defined in an item's value set.
- `by freq`: the rows will appear in order based on the frequency of each row's value.

By default, frequency tables are sorted ascending by code.

```
nonetpercents
```

If your frequency table contains blank values, two columns will be added to the frequency table, displaying the net percents (of the frequencies and cumulative frequencies). These net percent columns exclude the blank values from the percent calculations. If you use the `nonetpercents` command, these columns will not be added to the frequency table.

```
pagelength(number_lines)
```

The `pagelength` command specifies the number of lines that should be written to a frequency file before inserting a page break. The numeric constant ***number_lines*** must be a non-negative value. This option is only used in text frequencies.

Freq.Tally Function

Format

```
i = freq_name.tally([weight]);
```

Description

The `freq.tally` function tallies the frequency values that are part of a freq object. An optional numeric expression, ***weight***, can be used to specify a weight; if not provided, a weight of 1 is used during the tallying.

If using `freq.tally` in a data entry application, note that a value's visual value is used in the tally.

Return Value

The function returns the number of values that were tallied. If an invalid weight is provided, the function returns `default`.

Example

```
freq hh_status_freq(HH_STATUS);

forcase LISTING_DICT where FIPS = 69 do
    hh_status_freq.tally();
endfor;

hh_status_freq.view()
                heading("Household Status - Northern Mariana Islands");
```

See also: Freq Object, Freq.Clear Function

Freq.Clear Function

Format

```
b = freq_name.clear();
```

Description

The `freq.clear` function resets all of the tallied frequencies in a freq object, setting each tally to 0.

Return Value

The function returns a logical value of 1 (true).

Example

```
freq hh_status_freq(HH_STATUS);

// do something with hh_status_freq

hh_status_freq.clear();

// do something else with hh_status_freq
```

See also: Freq Object, Freq.Tally Function

Freq.Save Function

Format

```
b = freq_name.save([frequencies_filename]) [formatting_options];
```

Description

The `freq.save` function generates frequencies based on the values in a freq object and saves the frequency tables to a frequencies file. The optional string argument ***frequencies_filename*** allows you to specify the name of the file. If this argument is not provided, the frequency table will be saved to the file specified by the Freqs PFF attribute. If this PFF attribute is not specified, the frequency table will be saved to the listing file (if possible).

Optional formatting options allow you to control how the frequency tables are generated. The formatting options include the following commands: `valueset`, `distinct`, `vset`, `heading`, `stat`, `percentiles`, `nofreq`, `decimals`, `sort`, `nonetpercents`, and `pagelength`.

Return Value

The function returns a logical value of 1 (true) if the frequencies are successfully saved, and 0 (false) otherwise.

Example

```
freq hh_status_freq(HH_STATUS);

forcase LISTING_DICT where FIPS = 13 do
    hh_status_freq.tally();
endfor;

hh_status_freq.save("Georgia Household Status.tbw")
```

See also: Freq Object, Freq.View Function, Frequency Formatting Options

Freq.View Function

Format

```
b = freq_name.view() [formatting_options];
```

Description

The `freq.view` function generates HTML frequencies based on the values in a freq object and displays the frequency tables in an embedded web browser.

Optional formatting options allow you to control how the frequency tables are generated. The formatting options include the following commands: `valueset`, `distinct`, `vset`, `heading`, `stat`, `percentiles`, `nofreq`, `decimals`, `sort`, `nonetpercents`, and `pagelength`.

Return Value

The function returns a logical value of 1 (true) if the frequencies are successfully displayed, and 0 (false) otherwise.

Example

```
freq hh_status_freq(HH_STATUS);

forcase LISTING_DICT where FIPS = 69 do
    hh_status_freq.tally();
endfor;

hh_status_freq.view()
                heading("Household Status - Northern Mariana Islands");
```

See also: Freq Object, Freq.Save Function, Frequency Formatting Options

Geometry Object

Geometry Statement

Format

```
geometry geometry_name;
```

Description

The `geometry` statement creates a geometry with the name ***geometry_name***. The geometry name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare geometry globally in `PROC GLOBAL` or locally in functions or procedures.

Example

```
// Declare a geometry
geometry mygeometry;

// Capture polygon
mygeometry.tracePolygon();

// Compute the area of the captured polygon
AREA = mygeometry.area();

// Save the geometry to a file
mygeometry.save("polygon.geojson");
```

See also: Geometry Object

Geometry.Area Function

Format

```
d = geometry_name.area();
```

Description

The `geometry.area` function computes the area, in square meters, of the polygons contained in the geometry named `geometry_name`. If the geometry object contains multiple polygons, this function returns the sum of the areas of all of these polygons. If the polygons overlap, the results will be incorrect. Points and lines in the geometry are ignored, so if the geometry contains only points and lines, the area will be zero.

The `geometry.area` function uses the Haversine formula to approximate the area of the polygon and makes certain assumptions about the diameter and shape of the earth which may give slightly different results than the area calculated by other software packages.

Return Value

The function returns the sum of areas of all polygons in the geometry object in square meters or `default` if the geometry object is empty.

Example

```
// Declare a geometry
geometry mygeometry;

// Capture polygon
if mygeometry.tracePolygon() then

    // Compute the area of the captured polygon
    AREA = mygeometry.area();

endif;
```

See also: Geometry Object, Geometry.Perimeter Function

Geometry.Clear Function

Format

```
b = geometry_name.clear();
```

Description

The `geometry.clear` removes all data in the geometry named `geometry_name`. Any polygons, points, or lines that were stored in the geometry object by loading a file or tracing a polygon are deleted, leaving the geometry empty.

Return Value

The function returns a logical value of 1 (true).

Example

```
geometry mygeometry;

// Capture polygon
mygeometry.tracePolygon();

// Store the area
AREA = mygeometry.area();

// Clear the polygon and capture a new one
mygeometry.clear();
mygeometry.tracePolygon();
```

See also: Geometry Object

Geometry.Load Function

Format

```
b = geometry_name.load(filename);
```

Description

The `geometry.load` function reads the file specified by the string expression ***filename*** and stores the geometry data contained in the file in the geometry named `geometry_name`. The loaded geometry can be displayed on a map using `map.addGeometry`.

CSPro can load geometry from files in GeoJSON format.

Loading very large GeoJSON files can be slow and may exceed the memory of the device. In some cases it may be necessary to break up large GeoJSON files into smaller files. For example, rather than using a single file containing geometry for an entire country, create smaller files for the geometry in each district and then load only the file for the district that the interviewer is currently working in. Alternatively, rather than using geometry you can save the vector data in a custom base map.

Return Value

The function returns a logical value of 1 (true) if the file was loaded and 0 (false) if the file could not be loaded.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry boundary;
boundary.load("boundary.geojson");

// Add geometry to the map
mymap.addGeometry(boundary);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.Save Function

Geometry.MaxLatitude Function

Format

```
d = geometry_name.maxLatitude();
```

Description

The `geometry.maxLatitude` function finds the largest latitude value of any point on any element in the geometry named `geometry_name`. Together with the `geometry.minLatitude`, `geometry.minLongitude`, and `geometry.maxLongitude` functions it is possible to find the full extent of the geometry.

Return Value

The function returns the largest latitude value of any point or any point on a line or polygon in the geometry. If the geometry is empty the function returns `default`.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry geom;
geom.load("geom.geojson");

// Add geometry to the map
mymap.addGeometry(geom);

// Zoom map to fit the bounds of the geometry with 5% padding
mymap.zoomTo(geom.minLatitude(), geom.minLongitude(), geom.maxLatitude(), geom.maxLongitude(), 5
);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.MinLatitude Function, Geometry.MinLongitude Function, Geometry.MaxLongitude Function

Geometry.MaxLongitude Function

Format

```
d = geometry_name.maxLongitude();
```

Description

The `geometry.maxLongitude` function finds the largest longitude value of any point on any element in the geometry named `geometry_name`. Together with the `geometry.minLongitude`, `geometry.minLatitude`, and `geometry.maxLatitude` functions it is possible to find the full extent of the geometry.

Return Value

The function returns the largest longitude value of any point or any point on a line or polygon in the geometry. If the geometry is empty the function returns `default`.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry geom;
geom.load("geom.geojson");

// Add geometry to the map
mymap.addGeometry(geom);

// Zoom map to fit the bounds of the geometry with 5% padding
mymap.zoomTo(geom.minLatitude(), geom.minLongitude(), geom.maxLatitude(), geom.maxLongitude(), 5
);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.MinLongitude Function, Geometry.MinLatitude Function, Geometry.MaxLatitude Function

Geometry.MinLatitude Function

Format

```
d = geometry_name.minLatitude();
```

Description

The `geometry.minLatitude` function finds the smallest latitude value of any point on any element in the geometry named `geometry_name`. Together with the `geometry.maxLatitude`, `geometry.minLongitude`, and `geometry.maxLongitude` functions it is possible to find the full extent of the geometry.

Return Value

The function returns the smallest latitude value of any point or any point on a line or polygon in the geometry. If the geometry is empty the function returns `default`.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry geom;
geom.load("geom.geojson");

// Add geometry to the map
mymap.addGeometry(geom);

// Zoom map to fit the bounds of the geometry with 5% padding
mymap.zoomTo(geom.minLatitude(), geom.minLongitude(), geom.maxLatitude(), geom.maxLongitude(), 5
);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.MaxLatitude Function, Geometry.MinLongitude Function, Geometry.MaxLongitude Function

Geometry.MinLongitude Function

Format

```
d = geometry_name.minLongitude();
```

Description

The `geometry.minLongitude` function finds the smallest longitude value of any point on any element in the geometry named `geometry_name`. Together with the `geometry.maxLongitude`, `geometry.minLatitude`, and `geometry.maxLatitude` functions it is possible to find the full extent of the geometry.

Return Value

The function returns the smallest longitude value of any point or any point on a line or polygon in the geometry. If the geometry is empty the function returns `default`.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry geom;
geom.load("geom.geojson");

// Add geometry to the map
mymap.addGeometry(geom);

// Zoom map to fit the bounds of the geometry with 5% padding
mymap.zoomTo(geom.minLatitude(), geom.minLongitude(), geom.maxLatitude(), geom.maxLongitude(), 5
);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.MaxLongitude Function, Geometry.MinLatitude Function, Geometry.MaxLatitude Function

Geometry.Perimeter Function

Format

```
d = geometry_name.perimeter();
```

Description

The `geometry.perimeter` function computes the perimeter, in meters, of the polygons contained in the geometry named `geometry_name`. If the geometry object contains multiple polygons, this function returns the sum of the perimeters of all of these polygons. Points and lines in the geometry are ignored, so if the geometry contains only points and lines, the perimeter will be zero.

The `geometry.perimeter` function uses the Haversine formula to approximate the distance between points of the polygon and makes certain assumptions about the diameter and shape of the earth which may give slightly different results than the perimeter calculated by other software packages.

Return Value

The function returns the sum of perimeters of all polygons in the geometry object in meters or `default` if the geometry object is empty.

Example

```
// Declare a geometry
geometry mygeometry;

// Capture polygon
mygeometry.tracePolygon();

// Compute the perimeter of the captured polygon
PERIMETER = mygeometry.perimeter();
```

See also: Geometry Object, Geometry.Area Function

Geometry.Save Function

Format

```
b = geometry_name.save(filename);
```

Description

The `geometry.save` function saves the vector data (points, lines, and polygons) in the geometry named `geometry_name` to the file specified by the string expression ***filename***. The data is saved in GeoJSON format.

Return Value

The function returns a logical value of 1 (true) if the file was successfully saved or 0 (false) otherwise.

Example

```
// Declare a geometry
geometry outline;

// Have the user walk the outline of a polygon
if outline.walkPolygon() then

    // Save the outline to a GeoJSON file
    outline.save("outline.geojson");

endif;
```

See also: Geometry Object, Geometry.Load Function

Geometry.TracePolygon Function

Format

```
b = geometry_name.tracePolygon([map_name]);
```

Description

The `geometry.tracePolygon` function displays a map that the user taps on to place the vertices of a polygon. While tracing the polygon on the map, the user may also drag vertices of the polygon by long pressing on them in order to modify the polygon. The function returns when the user has finished drawing the polygon on the map and taps the save button.

The optional argument ***map_name*** is the name of a map object that will provide the display parameters used for the map that is shown while the user traces the polygon. You can use this parameter to set the base map, title, and location/zoom level of the map. You can also use the map parameter to display markers and geometry on the map, however any callbacks for clicking on markers or on the map itself are ignored while tracing the polygon. Any buttons on the map are replaced by buttons for saving, clearing, and deleting the last point placed while the polygon is being traced. If the argument ***map_name*** is omitted, a map with default settings is displayed while tracing the polygon.

After tracing the polygon, you can save the geometry to a file, compute the area of the polygon, or compute the perimeter of the polygon.

If `geometry_name` already contains a polygon, this polygon will be displayed and the user will be able to modify it by long pressing on the vertices to drag them. If the geometry contains more than one polygon, only the first polygon is shown. All entities in the geometry other than polygons are ignored. If the geometry was loaded from GeoJSON, any polygons in entities of type MultiPolygon are ignored. If you wish to capture a new polygon rather than editing the existing geometry, use the `geometry.clear` function before calling `geometry.tracePolygon`.

Return Value

The function returns a logical value of 1 (true) if the user traced a polygon and 0 (false) if the file user canceled polygon tracing by tapping the back button.

Example 1

```
// Declare a geometry
geometry outline;

// Have the user trace the outline of a polygon
if outline.tracePolygon() then

    // Save the outline to a GeoJSON file
    outline.save("outline.geojson");

endif;
```

Example 2

```
// Declare a geometry
geometry plot;

// Declare a map
map mymap;

// Set map display parameters
mymap.setBaseMap("basemap.mbtiles");
mymap.setTitle("Tap points on the map to outline the plot");

// Have the user trace the outline of a polygon using the map
if plot.tracePolygon(mymap) then

    // Capture the plot area
    PLOT_AREA = plot.area();

endif;
```

See also: Geometry Object, Geometry.WalkPolygon Function

Geometry.WalkPolygon Function

Format

```
b = geometry_name.walkPolygon([map_name]);
```

Description

The `geometry.walkPolygon` function displays an interactive map-based interface allowing the user to create a polygon by walking the perimeter of an area and recording their location using the GPS. The interface is similar to that used by `geometry.tracePolygon` except that rather than tapping on the map to place the vertices of the polygon, there is an add vertex button that places a vertex at the user's current location as determined by GPS. While walking the outline, the user may also drag vertices of the polygon by long pressing on them in order to modify the polygon. The function returns when the user has finished creating the polygon on the map and taps the save button.

The optional argument ***map_name*** is the name of a map object that will provide the display parameters used for the map that is shown while the user walks the polygon. You can use this parameter to set the base map, title, and location/zoom level of the map. You can also use the map parameter to display markers and geometry on the map, however any callbacks for clicking on markers or on the map itself are ignored while walking the polygon. Any buttons on the map are replaced by buttons for adding a vertex, saving, clearing, and deleting the last point placed while the polygon is being walked. If the argument ***map_name*** is omitted, a map with default settings is displayed while walking the polygon.

After walking the polygon, you can save the geometry to a file, compute the area of the polygon, or compute the perimeter of the polygon.

If `geometry_name` already contains a polygon, this polygon will be displayed and the user will be able to modify it by long pressing on the vertices to drag them. If the geometry contains more than one polygon, only the first polygon is shown. All entities in the geometry other than polygons are ignored. If the geometry was loaded from GeoJSON, any polygons in entities of type MultiPolygon are ignored. If you wish to capture a new polygon rather than editing the existing geometry, use the `geometry.clear` function before calling `geometry.walkPolygon`.

Return Value

The function returns a logical value of 1 (true) if the user created a polygon and 0 (false) if the file user canceled polygon walking by tapping the back button.

Example 1

```
// Declare a geometry
geometry outline;

// Have the user walk the outline of a polygon
if outline.walkPolygon() then

    // Save the outline to a GeoJSON file
    outline.save("outline.geojson");

endif;
```

Example 2

```
// Declare a geometry
geometry plot;

// Declare a map
map mymap;

// Set map display parameters
mymap.setBaseMap("basemap.mbtiles");
mymap.setTitle("Tap points on the map to outline the plot");

// Have the user walk the outline of a polygon using the map
if plot.walkPolygon(mymap) then

    // Capture the plot area
    PLOT_AREA = plot.area();

endif;
```

See also: Geometry Object, Geometry.TracePolygon Function

Geometry.GetProperty Function

Format

```
s = geometry_name.getProperty(property_name);
```

Description

The `geometry.getProperty` function returns a string containing the current value associated with the geometry's property specified by the string expression ***property_name***. If a geometry contains multiple features, the function returns the value of the first feature that contains a value for ***property_name***. Property names are case sensitive.

Property values may be set using the `geometry.setProperty` function. When reading a GeoJSON file using the `geometry.load` function, property names and values are read from the properties section of GeoJSON features.

Return Value

The function returns a string with the property information. If no feature in the geometry has a value for the property, a blank string is returned.

Example

```
// Load geometry from a file
geometry geom;
geom.load("geom.geojson");

// Read the name property
string feature_name = geom.getProperty("name");

errmsg("This feature is named %s", feature_name);
```

See also: Geometry Object, Geometry.SetProperty Function

Geometry.SetProperty Function

Format

```
b = geometry_name.setProperty(property_name, property_value);
```

Description

The `geometry.setProperty` function modifies the value associated with the geometry's property specified by the string expression ***property_name***. The property will be modified to the value given in the string or numeric expression ***property_value***. Properties may not be applied to empty geometry objects so you must first call the `geometry.load`, `geometry.tracePolygon`, or `geometry.walkPolygon` functions to add items to the geometry before setting properties. Property names are case sensitive.

When a geometry item contains multiple features, the property is set to ***property_value*** in ALL of the features of the geometry. When a geometry item is saved to a GeoJSON file using the `geometry.save` function, the properties are written to the GeoJSON file.

When displaying the geometry on a map, the following properties may be set to change the visual styling of the vector items:

Name	Value
stroke	Color used to draw lines and polygon outlines as an HTML color string (e.g. "#FF0000").
stroke-width	Width of the lines and outlines of polygons in display pixels.
fill	Color used to draw the interior of polygons as an HTML color string (e.g. "#FF0000").

Return Value

The function returns a logical value of 1 (true) because it will always successfully set a property.

Example 1

```
// Declare a geometry
geometry outline;

// Have the user walk the outline of a polygon
if outline.walkPolygon() then

    // Add the ID items of the case as properties to make it easier to identify when importing
into GIS system
    // PROV, DIST and HHNO are the names used in the GIS system
    outline.setProperty("PROV", PROVINCE);
    outline.setProperty("DIST", DISTRICT);
    outline.setProperty("HHNO", HOUSEHOLD_NUMBER);

    // Save the outline to a GeoJSON file
    outline.save("outline.geojson");

endif;
```

Example 2

```
// Declare a map
map mymap;

// Load geometry from a file
geometry boundary;
boundary.load("boundary.geojson");

// Set the polygon fill color to purple
boundary.setProperty("fill", "#94219");

// Set the outline color to red
boundary.setProperty("stroke", "#ff0000");

// Set the outline width to 3
boundary.setProperty("stroke-width", 3);

// Add geometry to the map
numeric geometryId = mymap.addGeometry(boundary);

// Show the map
mymap.show();
```

See also: Geometry Object, Geometry.GetProperty Function

HashMap Object

HashMap Statement

Format

```
hashmap ⌈value_type⌋ hashmap_name⌈(dimension_type1⌈, ..., dimension_typeN⌋)⌋ ⌈default(default_value)⌋;
```

Description

The `hashmap` statement creates a hashmap with the name ***hashmap_name***. The hashmap name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare hashmaps globally in `PROC GLOBAL` or locally in functions or procedures.

Hashmap values can be `numeric` or `string`. By default a hashmap is numeric, but the type can be modified by specifying the ***value_type***.

Unlike an array, a hashmap's dimensions are not of fixed size because hashmaps dynamically grow or shrink in size as values are added or removed from the hashmap. A hashmap can have one or more dimensions, and when declaring the hashmap you must specify the type of each dimension. Each ***dimension_type*** can be:

Dimension Type	Key Values
`all`	Numeric or string values
`numeric`	Numeric values
`string`	String values

If no dimensions are specified, then the hashmap is created with a single dimension of type `all`.

When assigning a value to a hashmap, all keys will be created as necessary to store the value. However, when retrieving a value, if the keys do not exist, you will get a runtime error and the value `default` or a blank string will be returned. If you want to assign a default value for undefined keys, you can specify a ***default_value***, which must be either a numeric constant or a string literal (based on the value type).

Example 1

```
hashmap string simple_hashmap;

simple_hashmap("Kwanzan") = "Cherry Tree";
simple_hashmap(1603) = "Beginning of Edo period";

errmsg("%s", simple_hashmap(1868)); // runtime error (1868 is an undefined key)
```

Example 2

```
hashmap three_dimensional_hashmap(all, numeric, string) default(0);

// proper access:
three_dimensional_hashmap(1, 2, "Z");
three_dimensional_hashmap("A", 2, "Z");

// compile-time errors:
three_dimensional_hashmap("A", "M", "Z"); // the second key must be a number
three_dimensional_hashmap(1, 2, 3);       // the third key must be a string

// with a default value defined, accessing this undefined key results in the displaying of 0
errmsg("%d", three_dimensional_hashmap(99, 88, "YY"));
```

See also: HashMap Object, Array Object, List Object

HashMap.GetKeys Function

Format

```
i = hashmap_name.getKeys(list_name [, key_value1, ..., key_valueN]);
```

Description

The `hashmap.getKeys` function fills a list with the keys at a specific location in the hashmap. Each ***key_value*** argument must be a numeric or string expression matching the type specified when declaring the hashmap. If no arguments are provided, the first-dimension keys are returned; if one argument is provided, the second-dimension keys are returned; and so on.

The argument ***list_name*** is a numeric or string list whose contents will be cleared and then filled with the hashmap's keys. The type of the list must correspond to the type of keys at the specified location. If the dimension type is `numeric`, then the list must be a numeric list. If the dimension type is `string` or `all`, then the list must be a string list. In the case of `all`, any numeric values added as keys to the hashmap will be converted to strings before being inserted in the string list.

Return Value

The function returns the number of keys added to the list.

Example

```
PROC GLOBAL

hashmap invalidValuesByPerson default(0);

PROC CENSUS_LEVEL

    list string person_list;

    invalidValuesByPerson.getKeys(person_list);

    do numeric counter = 1 while counter <= person_list.length()
        errmsg("%s had %d invalid values", strip(person_list(counter)),
                                           invalidValuesByPerson(person_list(counter)));
    enddo;

    invalidValuesByPerson.clear();

PROC SEX

    if not invalueset(SEX) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;

PROC AGE

    if not invalueset(AGE) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;
```

See also: HashMap Object, HashMap.Contains Function, HashMap.Length Function

HashMap.Contains Function

Format

```
b = hashmap_name.contains(key_value1[, ..., key_valueN]);
```

Description

The `hashmap.contains` function returns whether or not a specific key exists in a hashmap. Each ***key_value*** argument must be a numeric or string expression matching the type specified when declaring the hashmap.

Return Value

The function returns a logical value of 1 (true) if the hashmap contains the key, and 0 (false) otherwise.

Example

```
PROC GLOBAL

hashmap casesToOutput;

PROC CENSUS_FF

preproc

    // get a list of special cases to output from the external file
    forcase SPECIAL_CASES_DICT do
        casesToOutput(SPECIAL_CASE_KEY) = true;
    endfor;

PROC CENSUS_LEVEL

preproc

    // do not output this case if it was not in the list of special cases
    if not casesToOutput.contains(key(CENSUS_DICT)) then
        skip case;
    endif;
```

See also: HashMap Object, HashMap.GetKeys Function

HashMap.Length Function

Format

```
i = hashmap_name.length([key_value1, ..., key_valueN]);
```

Description

The `hashmap.length` function returns the number of keys at a specific location in the hashmap. Each ***key_value*** argument must be a numeric or string expression matching the type specified when declaring the hashmap. If no arguments are provided, the number of first-dimension keys is returned; if one argument is provided, the number of second-dimension keys is returned; and so on.

Return Value

The function returns the number of keys or 0 if no keys exist at the specified location.

Example

```
hashmap two_dimensional_hashmap(all, numeric);

two_dimensional_hashmap("A", 9) = 1;
two_dimensional_hashmap("A", 8) = 1;
two_dimensional_hashmap("A", 7) = 1;
two_dimensional_hashmap(100, 6) = 1;

errmsg("%d", two_dimensional_hashmap.length());    // 2 (keys "A" and 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 3 (keys 9, 8, and 7)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)

two_dimensional_hashmap.remove("A", 9);

errmsg("%d", two_dimensional_hashmap.length());    // 2 (keys "A" and 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 2 (keys 8 and 7)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)

two_dimensional_hashmap.remove("A");

errmsg("%d", two_dimensional_hashmap.length());    // 1 (key 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 0 (no keys)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)
```

See also: HashMap Object, HashMap.Contains Function, HashMap.GetKeys Function

HashMap.Remove Function

Format

```
b = hashmap_name.remove(key_value1[, ..., key_valueN]);
```

Description

The `hashmap.remove` function removes a specific key from a hashmap. Each ***key_value*** argument must be a numeric or string expression matching the type specified when declaring the hashmap.

Return Value

The function returns a logical value of 1 (true) if the hashmap contained the key and it was removed, and 0 (false) if the key did not exist.

Example

```
hashmap two_dimensional_hashmap(all, numeric);

two_dimensional_hashmap("A", 9) = 1;
two_dimensional_hashmap("A", 8) = 1;
two_dimensional_hashmap("A", 7) = 1;
two_dimensional_hashmap(100, 6) = 1;

errmsg("%d", two_dimensional_hashmap.length());    // 2 (keys "A" and 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 3 (keys 9, 8, and 7)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)

two_dimensional_hashmap.remove("A", 9);

errmsg("%d", two_dimensional_hashmap.length());    // 2 (keys "A" and 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 2 (keys 8 and 7)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)

two_dimensional_hashmap.remove("A");

errmsg("%d", two_dimensional_hashmap.length());    // 1 (key 100)
errmsg("%d", two_dimensional_hashmap.length("A")); // 0 (no keys)
errmsg("%d", two_dimensional_hashmap.length(100)); // 1 (key 6)
```

See also: HashMap Object, HashMap.Clear Function

HashMap.Clear Function

Format

```
b = hashmap_name.clear();
```

Description

The `hashmap.clear` function removes all values from a hashmap.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL

hashmap invalidValuesByPerson default(0);

PROC CENSUS_LEVEL

    list string person_list;

    invalidValuesByPerson.getKeys(person_list);

    do numeric counter = 1 while counter <= person_list.length()
        errmsg("%s had %d invalid values", strip(person_list(counter)),
                                            invalidValuesByPerson(person_list(counter)));
    enddo;

    invalidValuesByPerson.clear();

PROC SEX

    if not invalueset(SEX) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;

PROC AGE

    if not invalueset(AGE) then
        inc(invalidValuesByPerson(PERSON_NAME));
    endif;
```

See also: HashMap Object, HashMap.Remove Function

Image Object

Image Statement

Format

```
image image_name;
```

Description

The `image` statement creates an image with the name ***image_name***. The image name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare images globally in `PROC GLOBAL` or locally in functions or procedures.

Example

```
image roof_photo;

if roof_photo.takePhoto("Take a photo of the household's roof.") then

    // in case the device's camera takes photos with an unnecessarily
    // large resolution, resample the image to a more reasonable size
    roof_photo.resample(maxWidth := 1600, maxHeight := 1200);

    // save the image using the household key...
    string base_filename = path.concat(application, "Roof Photos", key(HH_DICT));

    // ...with 90 quality to prevent the JPEG from being too large
    roof_photo.save(base_filename + ".jpg", quality := 90);

endif;
```

See also: Image Object

Image.Load Function

Format

```
b = image_name.load(image_filename);

b = image_name.load(valueset_name, value);
```

Description

The `image.load` function reads an image from a file and stores its contents in the image object. In the first version of the function, the string expression ***image_filename*** specifies the location of the image on the disk.

The second version simplifies loading value set images. It is equivalent to calling:

```
image_name.load(getimage(valueset_name, value));
```

The following types of images can be loaded: JPEG (.jpg/.jpeg), PNG (.png), BMP (.bmp), and GIF (.gif). Although GIF files can be loaded, the `image.save` function cannot files in the GIF format.

Return Value

The function returns a logical value of 1 (`true`) if the image was loaded and 0 (`false`) if the image could not be loaded.

Example

```
// create thumbnails for every image in the Photos directory,
// placing the thumbnails in the Thumbnails directory
list string image_listing;
dirlist(image_listing, "Photos", "*.jpg;*.png");

do numeric counter = 1 while counter <= image_listing.length()

    image thumbnail_image;

    if thumbnail_image.load(image_listing(counter)) then

        // create a thumbnail at 25% of the original image size
        thumbnail_image.resample(thumbnail_image.width() / 4, thumbnail_image.height() / 4);

        // prefix the thumbnail filename with the text "tn"

string thumbnail_filename = path.concat("Thumbnails", "tn" + path.getFilename(image_listing(coun
ter)));

        thumbnail_image.save(thumbnail_filename);

    endif;

enddo;
```

See also: Image Object, Image.Save Function

Image.Save Function

Format

```
b = image_name.save(image_filename⌈, quality := quality⌋);
```

Description

The `image.save` function writes the contents of the image object to a file specified by the string expression ***image_filename***. The image can be saved in three formats, with the format determined by the file extension: JPEG (.jpg/.jpeg), PNG (.png), and BMP (.bmp). Although GIF files can be loaded by `image.load`, images cannot be saved to the GIF format.

When saving to JPEG format, the quality can be specified using a named argument. The numeric expression ***quality*** must be a value from 0 to 100, with 100 being the best quality. If not specified, a value of 95 is used. The file size will be bigger the higher the quality number. This value is ignored if not saving to JPEG format.

Return Value

The function returns a logical value of 1 (`true`) if the image was saved, 0 (`false`) if there was an error writing the file, and `default` if the image object does not contain an image.

Example

```
image roof_photo;

if roof_photo.takePhoto("Take a photo of the household's roof.") then

    // in case the device's camera takes photos with an unnecessarily
    // large resolution, resample the image to a more reasonable size
    roof_photo.resample(maxWidth := 1600, maxHeight := 1200);

    // save the image using the household key...
    string base_filename = path.concat(application, "Roof Photos", key(HH_DICT));

    // ...with 90 quality to prevent the JPEG from being too large
    roof_photo.save(base_filename + ".jpg", quality := 90);

endif;
```

See also: Image Object, Image.Load Function

Image.Width Function

Format

```
i = image_name.width();
```

Description

The `image.width` function returns the width, in pixels, of the image held by the image object.

Return Value

The function returns the width in pixels, or `default` if the image object does not contain an image.

Example

```
// create thumbnails for every image in the Photos directory,
// placing the thumbnails in the Thumbnails directory
list string image_listing;
dirlist(image_listing, "Photos", "*.jpg;*.png");

do numeric counter = 1 while counter <= image_listing.length()

    image thumbnail_image;

    if thumbnail_image.load(image_listing(counter)) then

        // create a thumbnail at 25% of the original image size
        thumbnail_image.resample(thumbnail_image.width() / 4, thumbnail_image.height() / 4);

        // prefix the thumbnail filename with the text "tn"

string thumbnail_filename = path.concat("Thumbnails", "tn" + path.getFilename(image_listing(coun
ter)));

        thumbnail_image.save(thumbnail_filename);

    endif;

enddo;
```

See also: Image Object, Image.Height Function, Image.Resample Function

Image.Height Function

Format

```
i = image_name.height();
```

Description

The `image.height` function returns the height, in pixels, of the image held by the image object.

Return Value

The function returns the height in pixels, or `default` if the image object does not contain an image.

Example

```
// create thumbnails for every image in the Photos directory,
// placing the thumbnails in the Thumbnails directory
list string image_listing;
dirlist(image_listing, "Photos", "*.jpg;*.png");

do numeric counter = 1 while counter <= image_listing.length()

    image thumbnail_image;

    if thumbnail_image.load(image_listing(counter)) then

        // create a thumbnail at 25% of the original image size
        thumbnail_image.resample(thumbnail_image.width() / 4, thumbnail_image.height() / 4);

        // prefix the thumbnail filename with the text "tn"

string thumbnail_filename = path.concat("Thumbnails", "tn" + path.getFilename(image_listing(coun
ter)));

        thumbnail_image.save(thumbnail_filename);

    endif;

enddo;
```

See also: Image Object, Image.Width Function, Image.Resample Function

Image.Resample Function

Format

```
b = image_name.resample(width, height);

b = image_name.resample(width := width,
                        height := height);

b = image_name.resample(maxWidth := max_width,
                        maxHeight := max_height);
```

Description

The `image.resample` function resamples the image held by the image object. Resampling modifies the dimensions of the image by either downsampling or upsampling the image. A loss of image quality may occur during the resampling

process.

In the first version of the function, two numeric expressions, ***width*** and ***height***, specify the new dimensions, in pixels, of the image. If one of the arguments is `notappl`, then the other dimension will be calculated by maintaining the original image's aspect ratio (the ratio of the width to the height). For example, if an image is 800 x 600, each of the following function calls results in a 400 x 300 image:

```
image_name.resample(400, 300);
image_name.resample(400, notappl);
image_name.resample(notappl, 300);
```

The second and third versions of the function, accessed by using named arguments, provide additional ways to control how the image is resampled. Specifying both ***width*** and ***height*** is equivalent to the first version of the function; specifying only one of the arguments is as if `notappl` were specified for the other argument. Providing a ***max_width*** and/or ***max_height*** will conditionally resample an image only if the image's width or height exceeds the provided maximum. Using the 800 x 600 example from above:

```
image_name.resample(maxWidth := 400);
image_name.resample(maxHeight := 1200);
```

The first function call results in a resampling to 400 x 300. The second function call does not result in a resampling because the original height, 600, does not exceed 1200. In this case, the function still returns 1 even though the image was not resampled.

Return Value

The function returns a logical value of 1 (`true`) if the image was resampled, 0 (`false`) if there was an error, and `default` if the image object does not contain an image.

Example 1

```
// create thumbnails for every image in the Photos directory,
// placing the thumbnails in the Thumbnails directory
list string image_listing;
dirlist(image_listing, "Photos", "*.jpg;*.png");

do numeric counter = 1 while counter <= image_listing.length()

    image thumbnail_image;

    if thumbnail_image.load(image_listing(counter)) then

        // create a thumbnail at 25% of the original image size
        thumbnail_image.resample(thumbnail_image.width() / 4, thumbnail_image.height() / 4);

        // prefix the thumbnail filename with the text "tn"

string thumbnail_filename = path.concat("Thumbnails", "tn" + path.getFilename(image_listing(coun
ter)));

        thumbnail_image.save(thumbnail_filename);

    endif;

enddo;
```

Example 2

```
image roof_photo;

if roof_photo.takePhoto("Take a photo of the household's roof.") then

    // in case the device's camera takes photos with an unnecessarily
    // large resolution, resample the image to a more reasonable size
    roof_photo.resample(maxWidth := 1600, maxHeight := 1200);

    // save the image using the household key...
    string base_filename = path.concat(application, "Roof Photos", key(HH_DICT));

    // ...with 90 quality to prevent the JPEG from being too large
    roof_photo.save(base_filename + ".jpg", quality := 90);

endif;
```

See also: Image Object, Image.Width Function, Image.Height Function

Image.TakePhoto Function

Format

```
b = image_name.takePhoto([message]);
```

Description

The `image.takePhoto` function takes a photo using a device's camera and stores the photo in the image object. This function only works on Android devices. The camera UI includes two options:

- **Flip camera icon**: Tap to toggle between the front and rear-facing camera.
- **Camera icon**: Tap to take the photo.

The optional string expression ***message*** will appear on the Android device's screen while the camera is active. The string will not be saved to the image.

Return Value

The function returns a logical value of 1 (`true`) if a photo was successfully taken and 0 (`false`) if there was an error taking a photo or if the operator canceled the operation.

Example

```
image roof_photo;

if roof_photo.takePhoto("Take a photo of the household's roof.") then

    // in case the device's camera takes photos with an unnecessarily
    // large resolution, resample the image to a more reasonable size
    roof_photo.resample(maxWidth := 1600, maxHeight := 1200);

    // save the image using the household key...
    string base_filename = path.concat(application, "Roof Photos", key(HH_DICT));

    // ...with 90 quality to prevent the JPEG from being too large
    roof_photo.save(base_filename + ".jpg", quality := 90);

endif;
```

See also: Image Object, Image.CaptureSignature Function

Image.CaptureSignature Function

Format

```
b = image_name.captureSignature([message]);
```

Description

The `image.captureSignature` function allows the drawing of a signature and stores the captured signature in the image object. This function only works on Android devices. The signature UI includes two options:

- **Save**: Tap to save an image of the displayed signature.
- **Clear**: Tap to to clear the displayed signature.

The optional string expression ***message*** will appear on the Android device's screen while the signature screen is active. The string will not be saved to the image.

Return Value

The function returns a logical value of 1 (`true`) if a signature was successfully captured and 0 (`false`) if there was an error capturing a signature or if the operator canceled the operation.

Example

```
PROC CONSENT

    // have the respondent sign to confirm that they agreed to this interview
    string consent_filename = path.concat("Consent Signatures", key(SURVEY_DICT) + ".png");

    // if the file exists, then they must have already consented on a previous
    // household visit so only take the signature if the file does not exist
    if not fileexist(consent_filename) then

        string consent_message = "Please sign your name, indicating that you consent to be
interviewed for this survey.";
        image consent_signature;

        if not consent_signature.captureSignature(consent_message) then
            errmsg("The respondent must consent before you can continue.");
            reenter;
        endif;

        consent_signature.save(consent_filename);

    endif;
```

See also: Image Object, Image.TakePhoto Function

Image.View Function

Format

```
b = image_name.view();
```

Description

The `image.view` function displays the image held by the image object. The function is equivalent to calling:

```
view(image_name);
```

The image will be show in an embedded window. If you prefer to display the image using the system's default viewer for the image type, you can use the following syntax:

```
view(filename(image_name));
```

This approach will only work if the image has been saved to the disk.

Return Value

The function returns a logical value of 1 (`true`) if the image was displayed, 0 (`false`) if there was an error viewing the image, and `default` if the image object does not contain an image.

Example

```
image tree_photo;

// ... load or take photo

tree_photo.view();

when accept("Are you satisfied with the photo of the tree?", "Yes", "No");
    1 -> tree_photo.save("Tree Photo.jpg");
    2 -> tree_photo.clear();
endwhen;
```

See also: Image Object

Image.Clear Function

Format

```
b = image_name.clear();
```

Description

The `image.clear` function clears the contents of the image object, resulting in the object not holding any image.

Return Value

The function returns a logical value of 1 (`true`).

Example

```
image tree_photo;

// ... load or take photo

tree_photo.view();

when accept("Are you satisfied with the photo of the tree?", "Yes", "No");
    1 -> tree_photo.save("Tree Photo.jpg");
    2 -> tree_photo.clear();
endwhen;
```

See also: Image Object

List Object

List Statement

Format

```
list [list_type] list_name;
```

Description

The `list` statement creates a list with the name ***list_name***. The list name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare lists globally in `PROC GLOBAL` or locally in functions or procedures.

Lists can be `numeric` or `string`. By default a list is numeric, but the type can be modified by specifying the ***list_type***.

The initial elements of a list can be set on definition by listing each value, separated by a comma. A list can also be initialized to the values of another list.

Example

```
// numeric lists
list DaysPerMonthTypical = 31, 28, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31;

list numeric DaysPerMonthLeapYears = DaysPerMonthTypical;
DaysPerMonthLeapYears(2) = 29;

// string lists
list string MonthNames = "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec";
```

See also: List Object, Array Object, HashMap Object, ValueSet Object

List.Add Function

Format

```
i = list_name.add(value || list_name);
```

Description

The `list.add` function adds a single value (the numeric or string expression ***value***) or a list of values (***list_name***) to the end of a list. The type of the added value or list must match the type of the receiving list.

Return Value

The function returns the number of the values added to the list. If the list is read-only, the function returns `default`.

Example

```
list string eligible_head_names;

do numeric counter = 1 while counter <= count(PERSON_REC)
    if AGE(counter) >= 15 and USUAL_MEMBER(counter) = 1 then
        eligible_head_names.add(NAME(counter));
    endif;
enddo;
```

See also: List Object, List.Insert Function

List.Insert Function

Format

```
i = list_name.insert(index, value || list_name);
```

Description

The `list.insert` function inserts a single value (the numeric or string expression ***value***) or a list of values (***list_name***) at the position in the list referenced by the numeric expression ***index***. The type of the inserted value or list must match the type of the receiving list. List index values are one-based, so the index must be between 1 and one past the length of the list. If the index is `1`, the values are inserted at the beginning of the list. If the value is `list.length() + 1`, the values are inserted at the end of the list (as would happen with the `list.add` function).

Return Value

The function returns the number of the values inserted into the list. The function returns 0 if the index is not valid. If the list is read-only, the function returns `default`.

Example

```
list string eligible_head_names;

do numeric counter = 1 while counter <= count(PERSON_REC)

    if AGE(counter) >= 15 and USUAL_MEMBER(counter) = 1 then

        // insert the names in sorted order
        numeric insert_index;

        do insert_index = 1 while insert_index <= eligible_head_names.length()
            and eligible_head_names(insert_index) <= NAME(counter)
        enddo;

        eligible_head_names.insert(insert_index, NAME(counter));

    endif;

enddo;
```

See also: List Object, List.Add Function

List.Seek Function

Format

```
i = list_name.seek(value, [starting_index || @nth_occurrence]);
```

Description

The `list.seek` function returns the one-based index of the first occurrence in the list of ***value***. The type of the value must match the type of the list. If the numeric expression ***starting_index*** is provided, then the function starts searching the for value starting at the index provided. Alternatively, if the @ symbol precedes the numeric expression ***nth_occurrence***, the function searches for the nth occurrence of the value.

Return Value

The function returns the one-based index of the value, if found, and 0 if no such value is in the list.

Example

```
list string keys_to_process;

// ...

if keys_to_process.seek(key(DICT_NAME)) = 0 then
    skip case;
endif;
```

See also: List Object, Seek Function

List.Remove Function

Format

```
b = list_name.remove(index);
```

Description

The `list.remove` function removes the value at the position in the list referenced by the numeric expression ***index***. List index values are one-based, so the index must be between 1 and the length of the list.

Return Value

The function returns a logical value of 1 (true) if the value is successfully removed, and 0 (false) otherwise, which indicates that the index is not valid. If the list is read-only, the function returns `default`.

Example

```
list string eligible_heads;

// ...

numeric eligible_head_index = eligible_heads.show("Select the head");

if eligible_head_index <> 0 and eligible_heads.length() > 1 then

    eligible_heads.remove(eligible_head_index);

    numeric secondary_head_index = eligible_heads.show("Select someone who could also be a
head");

endif;
```

See also: List Object, List.Clear Function, List.RemoveDuplicates Function, List.RemoveIn Function

List.RemoveDuplicates Function

Format

```
i = list_name.removeDuplicates();
```

Description

The `list.removeDuplicates` function removes duplicate values from a list, leaving the first instance of any duplicate in the list.

Return Value

The function returns the number of duplicates removed from the list. If the list is read-only, the function returns `default`.

Example

```
list string women_names;

for POPULATION_RECORD where HH_SEX = 2 and HH_AGE in 12:95 do
    women_names.add(HH_NAME);
enddo;

women_names.show("Women in Household");
// possible results: Maria, Maria, Linda, Maria, Pamela, Linda, Maria

numeric duplicates_removed = women_names.removeDuplicates();
women_names.show(maketext("Women in Household (%d duplicates removed)", duplicates_removed));
// possible results (4 duplicates removed): Maria, Linda, Pamela
```

See also: List Object, List.Clear Function, List.Remove Function, List.RemoveIn Function

List.RemoveIn Function

Format

```
i = list_name.removeIn(in_list);
```

Description

The `list.removeIn` function removes from a list any values that are specified in an in list. The values in ***in_list*** must be numeric for a numeric list and string for a string list. Note that the `in` operator is case sensitive, so when using a string list, specifying `"Cindy"` will not remove `"cindy"`.

Return Value

The function returns the number of values removed from the list. If the list is read-only, the function returns `default`.

Example 1

```
list household_ages;
// ...
// values: 69, 57, 55, 33, 22, 44, 34, 5

household_ages.removeIn(0:15, 50:99);
// result: 33, 22, 44, 34
```

Example 2

```
list string all_case_keys;
keylist(CENSUS_DICT(CaseStatus.All), all_case_keys);

list string not_deleted_case_keys;
keylist(CENSUS_DICT(CaseStatus.NotDeleted), not_deleted_case_keys);

list string deleted_case_keys = all_case_keys;
deleted_case_keys.removeIn(not_deleted_case_keys);
```

See also: List Object, List.Clear Function, List.Remove Function, List.RemoveDuplicates Function

List.Clear Function

Format

```
b = list_name.clear();
```

Description

The `list.clear` function removes all values from a list, setting the length to 0.

Return Value

The function returns a logical value of 1 (true). If the list is read-only, the function returns `default`.

Example

```
PROC GLOBAL

list possible_ages;

PROC AGE

    possible_ages.clear();

    do numeric counter = 1 while counter <= curocc()
        possible_ages.add(AGE);
    enddo;
```

See also: List Object, List.Remove Function, List.RemoveDuplicates Function, List.RemoveIn Function

List.Length Function

Format

```
i = list_name.length();
```

Description

The `list.length` function returns the size of a list, indicating the number of values contained by the list.

Return Value

The function returns the length of the list.

Example

```
list string file_listing;

dirlist(file_listing, pathname(InputFile), "*.csdb");

if file_listing.length() > 0 then
    skip to DATA_ALREADY_ENTERED;
endif;
```

See also: List Object, Length Function

List.Show Function

Format

```
i = list_name.show([heading]);
```

Description

The `list.show` function displays the values from a list and returns the index of the operator's selection. An optional string expression, ***heading***, specifies the title of the window that displays the values. The function is similar to the `accept` function but with the options taken from the list's values.

Return Value

The function returns the index of the item selected: 1 for the first value, 2 for the second value, etc. The value 0 is returned if the escape key (or back button) is pressed and none of the values is chosen.

Example

```
list string drink_options = "Water", "Milk", "Soda", "Coffee", "Tea";

numeric favorite_drink_index = drink_options.show("What is your favorite drink?");

if favorite_drink_index <> 0 then
    errmsg("You like %s the most!", drink_options(favorite_drink_index));
endif;
```

See also: List Object, List.Sort Function, Accept Function, Show Function, ShowArray Function, ValueSet.Show Function

List.Sort Function

Format

```
b = list_name.sort([ascending || descending]);
```

Description

The `list.sort` function sorts the values within a list. An optional argument, `ascending` or `descending`, allows for the specification of the sort order. If not specified, the default sort order is ascending. String values are sorted in case sensitive order, so `"c"` is recognized as different from `"C"`.

Return Value

The function returns a logical value of 1 (true). If the list is read-only, the function returns `default`.

Example

```
list string children_names;

for POPULATION_RECORD where HH_RELATIONSHIP = 3 do
    children_names.add(HH_NAME);
enddo;

children_names.show("Children in Household (before sort)");
// possible results: Betty, Ellen, Charlene, Donna, Danny, Donnie, Cindy, Elaine

children_names.sort();
children_names.show("Children in Household (after ascending sort)");
// possible results: Betty, Charlene, Cindy, Danny, Donna, Donnie, Elaine, Ellen

children_names.sort(descending);
children_names.show("Children in Household (after descending sort)");
// possible results: Ellen, Elaine, Donnie, Donna, Danny, Cindy, Charlene, Betty
```

See also: List Object, List.Show Function

Map Object

Map Statement

Format

```
map map_name;
```

Description

The `map` statement creates a map with the name map_name. The map_name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare maps globally in `PROC GLOBAL` or locally in functions or procedures.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
mymap.addMarker(38.84839, -76.931098);

// Display the map
mymap.show();
```

See also: Map Object

Map.Show Function

Format

```
b = map_name.show();
```

Description

The `map.show` function displays the map named `map_name`. The function call replaces the active view (generally either a question or a different map) and displays the map. The map will continue to be displayed until the user taps on the back button or the `map.hide` function is called from within a callback function.

The `map.show` function will not finish execution until the user has finished interacting with the map. For this reason, you should do all initial map setup (adding markers, buttons, base map, etc.) ***before*** calling `map.show`. If any map functions are given after the call to `map.show`, they will not be executed until after the map has been hidden. You can, however, make modifications to the map while it is showing from logic in a callback function supplied to one of the map functions such as `map.setOnClick`, `map.addTextButton`, `map.addImageButton`, `map.setMarkerOnClick`, `map.setMarkerOnClickInfoWindow`, or `map.setMarkerOnDrag`.

Return Value

The function returns a logical value of 1 (true) if the map was shown successfully and 0 (false) if there was an error.

Example

```
PROC GLOBAL

// User function that will be called on button click
// to close the map
function closeMap(map m)
    m.hide();
end;

PROC SOMEITEM

// Declare a map
map mymap;

// Add a button to the map that closes the map when it is clicked
mymap.addTextButton("close", closeMap(myMap));

// Show the map; this function will keep running until the user clicks
// the close button or uses the back button to exit the map
mymap.show();

// This logic is only run AFTER the call to hide
errmsg("Done showing map");
```

See also: Map Object, Map.Hide Function

Map.Hide Function

Format

```
b = map_name.hide();
```

Description

The `map.hide` function hides the map named `map_name` and returns to data entry. The `map.hide` function should be used in a callback function supplied to one of the map functions such as `map.setOnClick`, `map.addTextButton`, `map.addImageButton`, `map.setMarkerOnClick`, `map.setMarkerOnClickInfoWindow`, or `map.setMarkerOnDrag`. Calling `map.hide` will cause the earlier call to `map.show` to complete and return control to the statement following the call to `map.show`.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL

// User function that will be called on button click
// to close the map
function closeMap(map m)
    m.hide();
end;

PROC SOMEITEM

// Declare a map
map mymap;

// Add a button to the map that closes the map when it is clicked
mymap.addTextButton("close", closeMap(myMap));

// Show the map; this function will keep running until the user clicks
// the close button or uses the back button to exit the map
mymap.show();

// This logic is only run AFTER the call to hide
errmsg("Done showing map");
```

See also: Map Object, Map.Show Function

Map.SetTitle Function

Format

```
b = map_name.setTitle(title);
```

Description

The `map.setTitle` function sets the text that is displayed directly above the map to the value of the string expression ***title***. This can be used to display an instruction to the user such as "Tap on the map at the location of the household." By default the instruction is empty and is not shown. To remove the title call `map.setTitle("")` function with an empty string.

Return Value

The function returns a logical value of 1 (true).

Example

```
// Declare a map
map mymap;

// Set the title text above the map
mymap.setTitle("Tap on the map to place a household");

mymap.setOnClick(placeHousehold);

// Display the map
mymap.show();
```

See also: Map Object

Map.SetBaseMap Function

Format

```
b = map_name.setBaseMap(base_map);
```

Description

The `map.setBaseMap` function changes the base map that is displayed when the map named `map_name` is shown. Maps can use online base maps, which require an Internet connection to load, or can use offline base maps, which requires a map file stored on the device.

To specify an online map, use one of the following names as the ***base_map***:

Name	Description
Normal	Standard road map showing streets and outlines of some buildings.
Satellite	Satellite imagery.
Hybrid	Features from the street map overlaid on satellite imagery.
Terrain	Physical map based on terrain information.
None	Empty base map. Only markers and buttons are shown.

To use an offline map, specify ***base_map*** using a string expression containing the filename of the offline map.

When no base map is set, the default base map will be calculated using defined rules.

Return Value

The function returns a logical value of 1 (true) if the base map was changed successfully and 0 (false) otherwise (for example, if the offline map is not found).

Example 1 (Online Map)

```
// Declare a map
map mymap;

// Change the base map to satellite imagery
mymap.setBaseMap(Satellite);

// Display the map
mymap.show();
```

Example 2 (Offline Map)

```
// Declare a map
map mymap;

// Change the base map to an offline map file
mymap.setBaseMap("Maps/Suitland.mbtiles");

// Display the map
mymap.show();
```

See also: Map Object, Base Map Specification, Offline Maps

Map.ZoomTo Function

Format (Point)

```
b = map_name.zoomTo(latitude, longitude⌈, zoom_level⌉);
```

Format (Rectangle)

```
b = map_name.zoomTo(min_latitude, min_longitude, max_latitude, max_longitude[, padding]);
```

Description

The `map.zoomTo` function pans and zooms a map to center the map on a given point, or to fit a rectangular region of the map to the screen. The `map.zoomTo` function may be used before the map is displayed to set the initial view, or it may be used in a callback function to change the view while the map is showing.

The first format example (point) moves the map so that the point with coordinates (***latitude***, ***longitude***) is at the center of the screen. If the optional ***zoom_level*** is specified, the map is also zoomed to the specified zoom level. ***zoom_level*** is a number representing the level of detail to show. Zoom level one fits the entire world on the screen. Each consecutive zoom level is twice as detailed as the preceding level. Zoom level 15 shows streets, and zoom level 20 shows individual buildings. The minimum and maximum zoom levels supported will depend on the base map being used. If no zoom level is specified, the current zoom level is maintained.

The second format example (rectangle) pans and zooms the map to fit the rectangular region of the map with corners at (***min_latitude***, ***min_longitude***) and (***max_latitude***, ***max_longitude***). If the optional ***padding*** is specified, the map will be zoomed less so that the rectangular region will fit onto the screen with extra padding on each side. This can be used so that the edges of the rectangle do not line up exactly with the edge of the screen. The ***padding*** is specified as a percentage of the width of the screen. A padding of 5 would add padding equal to 5% of the device screen width.

If `map.zoomTo` is not called before the map is shown, then the initial view is determined in the order listed:

- If there are any markers, the view is set to fit all of the markers onto the screen.
- Next, if an offline map file that specifies an extent is used as a base map, the view is set to the extent of the offline map file.
- Last, the initial view is the entire world.

Return Value

The function returns a logical value of 1 (true) if the map is zoomed/panned and 0 (false) if there is an error (for example, if the rectangle is invalid).

Example 1 (Point)

```
// Declare a map
map mymap;

// Set initial view to center on the point 38.84839, -76.931098 at street level
mymap.zoomTo(38.84839, -76.931098, 15);

// Display the map
mymap.show();
```

Example 2 (Rectangle)

```
// Declare a map
map mymap;

// Set initial view to fit a rectangular region with 15% padding
mymap.zoomTo(38.841546, -76.937428, 38.853679, -76.911550, 15);

// Display the map
mymap.show();
```

See also: Map Object

Map.ShowCurrentLocation Function

Format

```
b = map_name.showCurrentLocation(show_condition);
```

Description

The `map.showCurrentLocation` function enables or disables display of the device's current location on the map `map_name`. If ***show_condition*** is non-zero, the current location is displayed on the map as a blue dot. As the user moves around, the blue dot also moves to follow the user's position.

In addition, there is a button on the right side of the map (shown below) that allows the user to zoom to their current location. If ***show_location*** is zero, both the blue dot and associated button are hidden. Display of the current location is enabled by default.

Zoom To Button

Return Value

The function returns a logical value of 1 (true).

Example

```
// Declare a map
map mymap;

// Disable display of current location
mymap.showCurrentLocation(false);

// Display the map
mymap.show();
```

See also: Map Object

Map.SaveSnapshot Function

Format

```
b = map_name.saveSnapshot(filename);
```

Description

The `map.saveSnapshot` function saves the currently displayed map as an image. The image is specified by the string expression ***filename*** and must have an extension associated with either JPEG or PNG formats. The map must be currently displayed when this function is called, so it must be executed via a callback function association with a marker or button.

Return Value

The function returns a logical value of 1 (true) if the map snapshot image was successfully saved and 0 (false) otherwise.

Example

```
PROC GLOBAL

function SaveMap(map m, string image_filename)
    m.saveSnapshot(image_filename);
end;

PROC EXAMPLE

    map capitals_map;

    // ...

    capitals_map.addTextButton("Save Image", SaveMap(capitals_map, "Capitals.png"));
```

See also: Map Object

Map.SetOnClick Function

Format

```
b = map_name.setOnClick(callbackFunction);
```

Description

The `map.setOnClick` function sets the function that is called when the user taps on the base map away from any markers. The ***callbackFunction*** is the name of a user function defined in the global procedure that will be called when the user taps on the map. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.setOnClick` is called rather than at the time the callback is run.

To retreive the coordinates of the map location that was tapped, use the functions `map.getLastClickLatitude` and `map.getLastClickLongitude`.

If the user taps on a marker rather than on the map itself, ***callbackFunction*** is not called. To capture taps on markers use the `map.setMarkerOnClick` function.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Declare id for map marker
numeric markerId;

// This function is called when user taps on map.
// It moves the marker to the location that the user tapped.
function mapClicked()
    numeric lat = mymap.getLastClickLatitude();
    numeric lon = mymap.getLastClickLongitude();
    mymap.setMarkerLocation(markerId, lat, lon);
end;

PROC SHOW_MAP
preproc

// Add a marker to the map at latitude 38.84839, longitude -76.931098
markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called when the user taps on the map to move the marker
// to the new position
mymap.setOnClick(mapClicked());

// Display the map
mymap.show();
```

See also: Map Object, Map.SetMarkerLocation Function, Map.SetMarkerOnClick Function, Map.GetMarkerLatitude Function, Map.GetMarkerLongitude Function

Map.GetLastClickLatitude Function

Format

```
lat = map_name.getLastClickLongitude();
```

Description

The `map.getLastClickLatitude` function retrieves the latitude of the last point on the map that the user tapped. This is often used in the callback function passed to `map.setOnClick` to get the coordinates of the point that was just tapped. The tapped latitude is only recorded if a user taps on the map itself rather than on a marker.

Return Value

The function returns the latitude of the tapped point in degrees or `notappl` if the user has not tapped a point on the map.

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Declare id for map marker
numeric markerId;

// This function is called when user taps on map.
// It moves the marker to the location that the user tapped.
function mapClicked()
    numeric lat = mymap.getLastClickLatitude();
    numeric lon = mymap.getLastClickLongitude();
    mymap.setMarkerLocation(markerId, lat, lon);
end;

PROC SHOW_MAP
preproc

// Add a marker to the map at latitude 38.84839, longitude -76.931098
markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called when the user taps on the map to move the marker
// to the new position
mymap.setOnClick(mapClicked());

// Display the map
mymap.show();
```

See also: Map Object, Map.GetLastClickLongitude Function, Map.GetMarkerLatitude Function, Map.SetMarkerLocation Function, Map.SetMarkerOnDrag Function

Map.GetLastClickLongitude Function

Format

```
long = map_name.getLastClickLongitude();
```

Description

The `map.getLastClickLongitude` function retrieves the longitude of the last point on the map that the user tapped. This is often used in the callback function passed to `map.setOnClick` to get the coordinates of the point that was just tapped. The tapped longitude is only recorded if a user taps on the map itself rather than on a marker.

Return Value

The function returns the longitude of the tapped point in degrees or `notappl` if the user has not tapped a point on the map.

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Declare id for map marker
numeric markerId;

// This function is called when user taps on map.
// It moves the marker to the location that the user tapped.
function mapClicked()
    numeric lat = mymap.getLastClickLatitude();
    numeric lon = mymap.getLastClickLongitude();
    mymap.setMarkerLocation(markerId, lat, lon);
end;

PROC SHOW_MAP
preproc

// Add a marker to the map at latitude 38.84839, longitude -76.931098
markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called when the user taps on the map to move the marker
// to the new position
mymap.setOnClick(mapClicked());

// Display the map
mymap.show();
```

See also: Map Object, Map.GetLastClickLatitude Function, Map.GetMarkerLatitude Function, Map.SetMarkerLocation Function, Map.SetMarkerOnDrag Function

Map.AddMarker Function

Format

```
markerId = map_name.addMarker(latitude, longitude);
```

Description

The `map.addMarker` function creates a new marker at the position defined by ***latitude*** and ***longitude*** and adds the new marker to the map named `map_name`. Initially the marker will have the default red and black marker icon and no description.

You can customize the marker icon using the function `map.setMarkerText` or `map.setMarkerImage`. You can add a description of the marker using `map.setMarkerDescription`. The description will appear in the marker list and in a popup when the user taps on the marker.

Return Value

The function returns the identifier of the new marker which may be used in subsequent calls to `map.removeMarker`, `map.setMarkerText`, `map.setMarkerImage`, `map.setMarkerDescription` `map.setMarkerOnClick`, `map.setMarkerOnClickInfoWindow`, `map.setMarkerOnDrag`, `map.setMarkerLocation`, `map.getMarkerLatitude`, or `map.getMarkerLongitude`.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set marker popup text
mymap.setMarkerDescription(markerId, "CSPro Team at the U.S. Census Bureau");

// Show the map
mymap.show();
```

See also: Map Object, Map.RemoveMarker Function

Map.RemoveMarker Function

Format

```
b = map_name.removeMarker(markerId);
```

Description

The `map.removeMarker` function removes a marker from the map named `map_name`. The ***markerId*** should be the id returned by `map.addMarker` function when the marker was added to the map.

Return Value

The function returns a logical value of 1 (true) if the marker was found and was successfully removed and 0 (false) if there is an error.

Example

```
PROC GLOBAL

// Function that allows the user to modify the marker text or description,
// or to remove the marker from the map
function editMarker(map m, numeric markerId)

    numeric sel = accept("Edit marker", "Text", "Description", "Remove");
    if sel = 1 then
        string newText = prompt("Enter new icon text");
        if newText <> "" then
            m.setMarkerText(markerId, newText);
        endif;
    elseif sel = 2 then
        string newDescription = prompt("Enter new icon description");
        if newDescription <> "" then
            m.setMarkerDescription(markerId, newDescription);
        endif;
    elseif sel = 3 then
        if accept("Confirm marker removal from the map", "Yes", "No") = 1 then
          m.removeMarker(markerId);
        endif;
    endif;
end;

PROC SHOW_MAP
preproc

// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);
mymap.setMarkerDescription(markerId, "U.S. Census Bureau");

// Call edit marker when user taps on popup info window
mymap.setMarkerOnClickInfoWindow(markerId, editMarker(mymap, markerId));

// Display the map
mymap.show();
```

See also: Map Object, Map.AddMarker Function

Map.ClearMarkers Function

Format

```
b = map_name.clearMarkers();
```

Description

The `map.clearMarkers` function removes all the markers currently on the map named `map_name`.

Return Value

The function returns a logical value of 1 (true).

Example

```
// User function that will be called on button click
// to filter households by status
function filterMapHouseholds(map myMap)

    numeric choice = accept("Choose status to show", "Not started", "Partial", "Complete");
    if choice > 0 then

        // Clear current markers
        myMap.clearMarkers();

        // Add markers for households matching status chosen
        forcase HOUSEHOLD_DICT WHERE HOUSEHOLD_STATUS = choice do
            numeric marker = myMap.addMarker(HOUSEHOLD_LATITUDE, HOUSEHOLD_LONGITUDE);
            myMap.setMarkerDescription(marker, maketext("%l -
%s", HOUSEHOLD_NUMBER, HOUSEHOLD_HEAD_NAME));
        endfor;
    endif;
end;
```

See also: Map Object, Map.AddMarker Function, Map.ClearButtons Function

Map.SetMarkerImage Function

Format

```
b = map_name.setMarkerImage(markerId, pathToImage);
```

Description

The `map.setMarkerImage` sets the icon for the marker with id ***markerId*** on map `map_name`. ***markerId*** is the id that was returned by `map.addMarker` when the marker was added to the map. ***pathToImage*** is the path to a file containing the image to display as the marker icon. Standard image files (PNG, JPEG, BMP) are supported.

If the marker icon was previously set with `map.setMarkerText`, calling `map.setMarkerImage` will replace the marker text with the image.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the image was set successfully and 0 (false) if there is an error.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set the icon
mymap.setMarkerImage(markerId, "resources/marker_icon.png");

// Display the map
mymap.show();
```

See also: Map Object, Map.AddMarker Function

Map.SetMarkerText Function

Format

```
b = map_name.setMarkerText(markerId, text[, backgroundColor, textColor]);
```

Description

The `map.setMarkerText` sets the icon for the marker with id ***markerId*** on map `map_name` to an icon with the specified text and color. The icon will contain the text surrounded by a small outline. ***markerId*** is the id that was returned by `map.addMarker` when the marker was added to the map. ***text*** is the phrase to display as the marker icon. You can optionally set the color of the text and the background color of the icon by specifying the ***backgroundColor*** and ***textColor***. Colors can be given by the names of common colors ("blue", "red", etc.) or using an HTML RGB color specification that starts with "#" such as "#404040". If no colors are specified the text will be black on a white background.

`map.setMarkerText` sets the text used in the marker icon. The icon text should be kept short to avoid making the map crowded. To associate a longer description with a marker use `map.setMarkerDescription` instead to display additional text in the marker list and in a popup on the map when the user taps the marker.

The ***text*** may contain the following HTML tags:

- <b> - bold
- <i> - italic
- - underline
- <strike> or <s> - strikethrough
- <sup> - superscript
- <sub> - subscript
- <font> - font (including color)
-
 - line break

If the marker icon was previously set with `map.setMarkerImage`, calling `map.setMarkerText` will replace the marker image with the new text.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the text was set successfully and 0 (false) if there is an error.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set the icon
mymap.setMarkerText(markerId, maketext("%v", HH_NUMBER), "white", "black");

// Display the map
mymap.show();
```

See also: Map Object, Map.AddMarker Function, Map.SetMarkerImage Function

Map.SetMarkerDescription Function

Format

```
b = map_name.setMarkerDescription(markerId, description);
```

Description

The `map.setMarkerDescription` function associates the string expression ***description*** with the given ***markerId***. The description will appear in a popup box next to the ***markerId***'s map location when the user taps the marker. The text box will disappear when the popup box is closed.

The ***description*** may contain the following HTML tags:

- <b> - bold
- <i> - italic
- - underline
- <strike> or <s> - strikethrough
- <sup> - superscript
- <sub> - subscript
- <font> - font (including color)
-
 - line break

Return Value

The function returns a logical value of 1 (true) if the marker was found and the description was set successfully, or 0 (false) if there was an error.

Example

```
// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set the popup text when marker is tapped to
// display name of head of household and number of household
// members
string popupText = maketext("Head: %s %s Males: %d Females %d",
        strip(HEAD_FIRST_NAME),
        strip(HEAD_LAST_NAME),
        MALE_MEMBERS, FEMALE_MEMBERS);
mymap.setMarkerDescription(markerId, popupText);

// Display the map
mymap.show();
```

See also: Map Object, Map.AddMarker Function, Map.SetMarkerText Function

Map.SetMarkerOnClick Function

Format

```
b = map_name.setMarkerOnClick(markerId, callbackFunction);
```

Description

The `map.setMarkerOnClick` function sets the function that is called when the user taps on the marker identified by ***markerId***. When the user taps on the marker the user-defined function ***callbackFunction*** will be called.

The ***callbackFunction*** is the name of a user function defined in the global procedure. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.setMarkerOnClick` is called rather than at the time the callback is run. This allows you to reuse the same callback function for multiple markers and to customize the behavior of the callback by passing different arguments to the callback function for each marker.

If no on-click function is set for a marker then when the marker is clicked, the default behavior is to center the view on the marker and to display the marker's info window. After setting an on-click function with `map.setMarkerOnClick` function, the popup info window is not shown when the marker is clicked. If you wish to still display the popup window and also handle taps on the marker you can use the function `map.setMarkerOnClickInfoWindow()`.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the callback function was set successfully and 0 (false) if there is an error.

Example 1

```
PROC GLOBAL

function clickedOnMarker()
    errmsg("You clicked on a marker");
end;

PROC SOMEITEM

// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called on marker click
mymap.setMarkerOnClick(markerId, clickedOnMarker());

// Display the map
mymap.show();
```

Example 2

```
PROC GLOBAL

function launchHouseholdQuestionnaire(string caseid)

    pff household_pff;

    household_pff.setproperty("AppType", "Entry");

    household_pff.setproperty("Key", caseid);

    household_pff.setproperty("Application", "Household.ent");
    household_pff.setproperty("InputData", "Household.csdb");

    household_pff.setproperty("OnExit", "Menu.pff");

    household_pff.exec();

end;

PROC SHOW_HOUSEHOLDS_ON_MAP
preproc

// Declare a map
map mymap;

// Loop through all households in the listing dictionary
forcase LIST_DICT do

    // Create a marker for household at latitude and longitude from listing file
    numeric markerId = mymap.addMarker(LI_LATITUDE, LI_LONGITUDE);

    // Set the marker text to the household number
    mymap.setMarkerText(markerId, maketext("%v", LI_HOUSEHOLD_NUMBER));

    // When marker is clicked launch household questionnaire application
    // and prefill the ID items
    string householdCode = key(LIST_DICT);
    mymap.setMarkerOnClick(markerId, launchHouseholdQuestionnaire(householdCode));

endfor;

// Show the map
mymap.show();
```

See also: Map Object, Map.AddMarker Function, Map.SetMarkerOnClickInfoWindow Function

Map.SetMarkerOnClickInfoWindow Function

Format

```
b = map_name.setMarkerOnClickInfoWindow(markerId, callbackFunction);
```

Description

The `map.setMarkerOnClick` function sets the function that is called when the user taps on the popup info window for the marker identified by ***markerId***. The popup info window is shown when the user taps on the marker and it displays the marker description set by calling `map.setMarkerDescription`. After tapping on the marker to display the info window, when the user then taps on the popup, the user-defined function ***callbackFunction*** will be called.

The ***callbackFunction*** is the name of a user function defined in the global procedure. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.setMarkerOnClickInfoWindow` is called rather than at the time the callback is run. This allows you to reuse the same callback function for multiple markers and to customize the behavior of the callback by passing different arguments to the callback function for each marker.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the callback function was set successfully and 0 (false) if there is an error.

Example 1

```
PROC GLOBAL

function clickedOnMarkerInfoWindow()
    errmsg("You clicked on a marker popup");
end;

PROC SOMEITEM

// Declare a map
map mymap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called on marker info window click
mymap.setMarkerOnClick(markerId, clickedOnMarkerInfoWindow());

// Display the map
mymap.show();
```

Example 2

```
PROC GLOBAL

function launchHouseholdQuestionnaire(string caseid)

    pff household_pff;

    household_pff.setproperty("AppType", "Entry");

    household_pff.setproperty("Key", caseid);

    household_pff.setproperty("Application", "Household.ent");
    household_pff.setproperty("InputData", "Household.csdb");

    household_pff.setproperty("OnExit", "Menu.pff");

    household_pff.exec();

end;

PROC SHOW_HOUSEHOLDS_ON_MAP
preproc

// Declare a map
map mymap;

// Loop through all households in the listing dictionary
forcase LIST_DICT do

    // Create a marker for household at latitude and longitude from listing file
    numeric markerId = mymap.addMarker(LI_LATITUDE, LI_LONGITUDE);

    // Set the marker description to the household head name.
    // This will be displayed in the popup info window when the user taps the marker
    mymap.setMarkerDescription(markerId, strip(LI_HOUSEHOLD_HEAD_NAME));

    // When the info window is clicked launch household questionnaire application
    // and prefill the ID items
    string householdCode = key(LIST_DICT);
    mymap.setMarkerOnClickInfoWindow(markerId, launchHouseholdQuestionnaire(householdCode));

endfor;
```

See also: Map Object, Map.AddMarker Function, Map.SetMarkerOnClick Function, Map.SetMarkerDescription Function

Map.SetMarkerOnDrag Function

Format

```
b = map_name.setMarkerOnDrag(markerId, callbackFunction);
```

Description

The `map.setMarkerOnDrag` function sets the function that is called when the user drags the marker on the map marker identified by ***markerId***. Markers may be dragged by first long pressing on them and then sliding them to a new position. The ***callbackFunction*** is called at the end of the drag when the user lifts up their finger to release the marker.

The ***callbackFunction*** is the name of a user function defined in the global procedure. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.setMarkerOnDrag` is called rather than at the time the callback is run. This allows you to reuse the same callback function for multiple markers and to customize the behavior of the callback by passing different arguments to the callback function for each marker.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the callback function was set successfully and 0 (false) if there is an error.

Example

```
PROC GLOBAL

function onDragMarker(map dragMap, numeric markerId)
    numeric lat = dragMap.getMarkerLatitude(markerId);
    numeric lon = dragMap.getMarkerLongitude(markerId);
    errmsg("Marker was moved to: %f, %f", lat, lon);
end;

PROC SHOW_MAP
preproc

// Declare a map
map dragMap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = dragMap.addMarker(38.84839, -76.931098);

// Set function that is called on marker drag
dragMap.setMarkerOnDrag(markerId, onDragMarker(dragMap, markerId));

// Display the map
dragMap.show();
```

See also: Map Object, Map.AddMarker Function, Map.SetMarkerOnClick Function, Map.GetMarkerLatitude Function, Map.GetMarkerLongitude Function

Map.SetMarkerLocation Function

Format

```
b = map_name.setMarkerLocation(markerId, latitude, longitude);
```

Description

The `map.setMarkerLocation` function moves the marker identified by ***markerId*** to a location on the map with coordinates ***latitude*** and ***longitude***. ***latitude*** and ***longitude*** should be given in degrees.

Return Value

The function returns a logical value of 1 (true) if the marker was found and the location was set successfully and 0 (false) if there is an error.

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Declare id for map marker
numeric markerId;

// This function is called when the user taps on the map.
// It moves the marker to the location that the user tapped.
function mapClicked()

mymap.setMarkerLocation(markerId, mymap.getLastClickLatitude(), mymap.getLastClickLongitude());
end;

PROC SHOW_MAP
preproc

// Add a marker to the map at latitude 38.84839, longitude -76.931098
markerId = mymap.addMarker(38.84839, -76.931098);

// Set function that is called when the user taps on the map to move the marker
// to the new position
mymap.setOnClick(mapClicked());

// Display the map
mymap.show();
```

See also: Map Object, Map.SetMarkerOnClick Function, Map.GetMarkerLatitude Function, Map.GetMarkerLatitude Function

Map.GetMarkerLatitude Function

Format

```
lat = map_name.getMarkerLatitude(markerId);
```

Description

The `map.getMarkerLatitude` function retrieves the latitude of the marker identified by ***markerId***.

Return Value

The function returns the latitude of the marker in degrees if the marker was found and returns `default` if the marker id is invalid.

Example

```
PROC GLOBAL

function onDragMarker(map dragMap, numeric markerId)
    numeric lat = dragMap.getMarkerLatitude(markerId);
    numeric lon = dragMap.getMarkerLongitude(markerId);
    errmsg("Marker was moved to: %f, %f", lat, lon);
end;

PROC SHOW_MAP
preproc

// Declare a map
map dragMap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = dragMap.addMarker(38.84839, -76.931098);

// Set function that is called on marker drag
dragMap.setMarkerOnDrag(markerId, onDragMarker(dragMap, markerId));

// Display the map
dragMap.show();
```

See also: Map Object, Map.GetMarkerLongitude Function, Map.SetMarkerLocation Function, Map.SetMarkerOnDrag Function

Map.GetMarkerLongitude Function

Format

```
long = map_name.getMarkerLongitude(markerId);
```

Description

The `map.getMarkerLongitude` function retrieves the longitude of the marker identified by ***markerId***.

Return Value

The function returns the longitude of the marker in degrees if the marker was found and returns `default` if the marker id is invalid.

Example

```
PROC GLOBAL

function onDragMarker(map dragMap, numeric markerId)
    numeric lat = dragMap.getMarkerLatitude(markerId);
    numeric lon = dragMap.getMarkerLongitude(markerId);
    errmsg("Marker was moved to: %f, %f", lat, lon);
end;

PROC SHOW_MAP
preproc

// Declare a map
map dragMap;

// Add a marker to the map at latitude 38.84839, longitude -76.931098
numeric markerId = dragMap.addMarker(38.84839, -76.931098);

// Set function that is called on marker drag
dragMap.setMarkerOnDrag(markerId, onDragMarker(dragMap, markerId));

// Display the map
dragMap.show();
```

See also: Map Object, Map.GetMarkerLatitude Function, Map.SetMarkerLocation Function, Map.SetMarkerOnDrag Function

Map.AddGeometry Function

Format

```
geometryId = map_name.addGeometry(geometry_name);
```

Description

The `map.addGeometry` function adds the vector data (points, lines, and polygons) contained in ***geometry_name*** to the map named `map_name`. The vector data will be drawn on the map in a layer on top of the base map. Points are rendered as markers while lines and polygons are drawn as lines and polygons. The vector data may be loaded from a GeoJSON file using the function `geometry.load` or traced by the interviewer using the `geometry.tracePolygon` or `geometry.walkPolygon` functions.

Loading very large GeoJSON files can be slow and may exceed the memory of the device. In some cases it may be necessary to break up large GeoJSON files into smaller files. For example, rather than using a single file containing geometry for an entire country, create smaller files for the geometry in each district and then load only the file for the district that the interviewer is currently working in. Alternatively, rather than using geometry you can save the vector data in a custom base map.

Return Value

The function returns the identifier of the new geometry which may be used in subsequent calls to `map.removeGeometry` to delete the geometry from the map.

Example

```
// Declare a map
map mymap;

// Load geometry from a file
geometry boundary;
boundary.load("boundary.geojson");

// Add geometry to the map
numeric geometryId = mymap.addGeometry(boundary);

// Show the map
mymap.show();
```

See also: Map Object, Geometry Object, Map.RemoveGeometry Function, Map.ClearGeometry Function

Map.RemoveGeometry Function

Format

```
b = map_name.removeGeometry(geometryId);
```

Description

The `map.removeGeometry` function removes geometry from the map `map_name`. ***geometryId*** should be the id returned by `map.addGeometry` when the geometry was added to the map.

Return Value

The function returns a logical value of 1 (true) if the geometry was found and was successfully removed and 0 (false) if there is an error.

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Declare a geometry to hold boundary polygon
geometry boundary;

// Declare a variable to hold the id of the boundary on the map
numeric boundaryGeometryId;

function showBoundary()
    // Add boundary polygon to the map and save the id
    boundaryGeometryId = mymap.addGeometry(boundary);
end;

function hideBoundary()
    // Remove the boundary polygon from the map
    mymap.removeGeometry(boundaryGeometryId);
    boundaryGeometryId = 0;
end;

function toggleBoundary()
    if boundaryGeometryId = 0 then
        showBoundary();
    else
        hideBoundary();
    endif;
end;

PROC SHOW_MAP
preproc

// Load the boundary polygon from a file
boundary.load("boundary.geojson");

// Add a button to show/hide the boundary
mymap.addTextButton("Boundary", toggleBoundary());

// Display the map
mymap.show();
```

See also: Map Object, Geometry Object, Map.AddGeometry Function, Map.ClearGeometry Function

Map.ClearGeometry Function

Format

```
b = map_name.clearGeometry();
```

Description

The `map.clearGeometry` function removes all the geometry currently on the map named `map_name`. Note that this clears only geometry added using the `map.addGeometry` function. Markers added to the map using `map.addMarker` should be cleared using the function `map.clearMarkers`.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL

function editOutline(map m, geometry outline)
    // Remove existing outline
    m.clearGeometry();

    // Edit the outline
    if outline.tracePolygon(m) then
        outline.save("outline.geojson");
    endif;

    // Add modified outline to the map
    m.addGeometry(outline);
end;

PROC MAP_OUTLINE

// Declare a map to draw the outline on
map m;

// Load the outline from a GeoJSON file
geometry outline;
outline.load("outline.geojson");

// Add the geometry to the map
m.addGeometry(outline);

// Add a button to allow editing the outline
m.addTextButton("Edit outline", editOutline(m, outline));

// Display the map
m.show();
```

See also: Map Object, Geometry Object, Map.AddGeometry Function, Map.RemoveGeometry Function

Map.AddTextButton Function

Format

```
buttonId = map_name.addTextButton(text, callbackFunction);
```

Description

The `map.addTextButton` function adds a new button to the map with the label ***text***. Map buttons are displayed on the right-hand side of the map. The new button will be added below any existing buttons. When the user taps on the button the user-defined function ***callbackFunction*** will be called.

The ***callbackFunction*** is the name of a user function defined in the global procedure. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.addTextButton` is called rather than at the time the callback is run. This allows you to reuse the same callback function for multiple buttons and to customize the behavior of the callback by passing different arguments to the callback function for each button.

Return Value

The function returns the identifier of the new button which may be used in subsequent calls to `map.removeButton`.

Example

```
PROC GLOBAL

// User function that will be called on button click
// to close the map
function closeMap(map m)
    m.hide();
end;

PROC SOMEITEM
preproc

// Declare a map
map mymap;

// Add a button to the map that closes the map when it is clicked
mymap.addTextButton("close", closeMap(mymap));

// Show the map; this function will keep running until the user clicks
// the close button or uses the back button to exit the map
mymap.show();
```

See also: Map Object, Map.RemoveButton Function, Map.AddImageButton Function

Map.AddImageButton Function

Format

```
buttonId = map_name.addImageButton(pathToImage, callbackFunction);
```

Description

The `map.addImageButton` function adds a new button to the map, using the image in the ***pathToImage*** file as the button's icon. The ***pathToImage*** argument is the path to a file containing the image to display on the button. Standard image files (PNG, JPEG, BMP) are supported. Map buttons are displayed on the right hand-side of the map. The new button will be added below any existing buttons. When the user taps on the button, the user-defined function ***callbackFunction*** will be called.

The ***callbackFunction*** is the name of a user function defined in the global procedure. The function name may optionally be followed by function arguments which will be passed to the function when it is run. These arguments are evaluated at the time that `map.addTextButton` is called rather than at the time the callback is run. This allows you to reuse the same callback function for multiple buttons and to customize the behavior of the callback by passing different arguments to the callback function for each button.

Return Value

The function returns the identifier of the new button which may be used in subsequent calls to `map.removeButton`.

Example

```
PROC GLOBAL

// User function that will be called on button click
// to close the map
function closeMap(map m)
    m.hide();
end;

PROC SOMEITEM

// Declare a map
map mymap;

// Add a button to the map that closes the map when it is clicked
mymap.addImageButton("resources/button_close.png", closeMap(mymap));

// Show the map; this function will keep running until the user clicks
// the close button or uses the back button to exit the map
mymap.show();
```

See also: Map Object, Map.RemoveButton Function, Map.AddTextButton Function

Map.RemoveButton Function

Format

```
b = map_name.removeButton(buttonId);
```

Description

The `map.removeMarker` function removes a marker from the map `map_name`. ***buttonId*** should be the id returned by `map.addTextButton` or `map.addImageButton` when the button was added to the map.

Return Value

The function returns a logical value of 1 (true) if the button was found and was successfully removed and 0 (false) if there is an error.

Example

```
PROC GLOBAL

// Declare a map
map mymap;

// Id of button that will be shown on map
numeric buttonId;

// Function to remove a button from a map
function removeButton()
    mymap.removeButton(buttonId);
end;

PROC SHOW_MAP
preproc

// Add a button that removes itself when it is clicked
buttonId = mymap.addTextButton("remove me", removeButton());

// Display the map
mymap.show();
```

See also: Map Object, Map.AddImageButton Function, Map.AddTextButton Function

Map.ClearButtons Function

Format

```
b = map_name.clearButtons();
```

Description

The `map.clearButtons` function removes all the buttons currently on the map named `map_name`.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL
numeric showAddHousehold = true;
numeric showFilter = true;

// User function that will be called on button click
// to update the buttons
function updateButtons(map myMap)

    // Clear existing buttons
    myMap.clearButtons();

    if showAddHousehold then
        myMap.addTextButton("Add", addHousehold(myMap));
    endif;

    if showFilterButton then
        myMap.addTextButton("Filter", filterHouseholds(myMap));
    endif;

    myMap.addTextButton("Done", closeMap(myMap));
end;
```

See also: Map Object, Map.AddTextButton Function, Map.AddImageButton Function, Map.ClearMarkers Function

Base Map Specification

When using CSPro's mapping features, you can use as a base map an online map, which requires an Internet connection, or an offline map. Online maps are defined using predefined names:

Name	Description
Normal	Standard road map showing streets and outlines of some buildings.
Satellite	Satellite imagery.
Hybrid	Features from the street map overlaid on satellite imagery.
Terrain	Physical map based on terrain information.
None	Empty base map. Only markers and buttons are shown.

When specifying an online base map name, you can use the name directly or use it as a string expression. For example, these statements have the same effect:

```
map my_map;
string desired_base_map = "Terrain";

my_map.setBaseMap(Terrain);
my_map.setBaseMap(desired_base_map);
```

Default Base Map Calculation

When using a map object, or using functionality such as the interactive GPS modes, there are ways to specify the base map (by using the `map.setBaseMap` function or by providing a ***base_map*** argument to the `gps` function call). However, if no base map is specified, the default base map is calculated using the following rules:

1. If the PFF has a BaseMap attribute set, use it.
2. Otherwise use the Default Base Map setting as defined in the Mapping Properties, which defaults to the **Normal** map type.

See also: Mapping, Mapping Properties, Offline Maps, Map.SetBaseMap Function

Offline Maps

The default base map shown is the Google Maps road map which requires an Internet connection. You can instead use an offline map from a file stored on your device. Offline map files store map tiles which are images of pieces of a map at various levels of detail. Rather than downloading map images from Google Maps, CSPro will read the images directly from an offline map file.

Supported Files

CSPro supports the following formats for offline map files:

MBTiles	An open format orginally developed by Mapbox but now widely used by many other systems. MBTiles can be exported from QGIS. You can also download MBTiles files created from Open Street Map data from the Humanitarian Open Street Maps project at https://www.hotosm.org. Only MBTiles containing image tiles (PNG or JPEG) are supported. Vector tiles are not supported.
ArcGIS Tile Package	Tile packages exported from ArcGIS. See the ArcGIS Help for details. Tile packages (.tpk files) must be created using the *ArcGIS Online/Bing Maps/Google Maps* tiling scheme to be compatible with CSPro.

Putting Maps on Devices

Once you have a map file you can copy it to your device via a USB connection. You can also download map files to your device using the `syncfile` command in your logic; however, since map files tend to be large, this may be slow. Map files can be placed anywhere on the device although it is most common to place them within a subfolder of your application's folder.

Specifying the Map to Use in the Case Listing

Use the BaseMap attribute in the **[Files]** section of the PFF file to provide the path to your device's offline map file. See Display the Case Listing on a Map for more info.

Specifying the Map to Use With the Map Object

When using the `map` object in logic, use the `map.setBaseMap` function to set the offline map.

File Size

The level of detail of an offline map is a big component of the file size. Levels of detail in offline maps start at zero, where the entire world is covered by a single image tile. Each successive level contains four times the number of images as the previous level. This means:

Level 1 contains 4 tiles
Level 2 contains 16 tiles
Level 3 contains 64 tiles, etc.

Level 20 is detailed enough to see individual buildings. At level 20 the entire world requires over a trillion tiles. By removing the last level of detail from your map, you reduce the size of the file by close to two-thirds.

Since offline map files may contain hundreds or thousands of images, they can become very large. The file size will depend on the extents of the map and the level of detail. It is not possible to store a map of the entire world, or even of an entire country, at a high level of detail on most devices. When creating your map files, you will need to adjust the extents and level of detail to ensure that the map files will not exceed the available storage space on your device. If your survey will be conducted over a large geographic area, you may need to provide separate maps for each interviewer or each team that only covers their assigned area.

See also: Map Object, Base Map Specification

Path

Path Namespace

The path namespace is a set of functions used to perform operations on a path or to get information from a path specification.

Functionality

Function	Description
path.concat	Combines multiple strings into one string representing a file path.
path.getDirectoryName	Extracts the directory name from a file path.
path.getExtension	Extracts the file extension from a file path.
path.getFileName	Extracts the file name from a file path.
path.getFileNameWithoutExtension	Extracts the file name without its extension from a file path.
path.getRelativePath	Returns a relative path based on a path's location relative to a directory.
path.selectFile	Shows the operator a visual listing of the file system and returns the path of the file the operator selected.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: PathName Function

Path.Concat Function

Format

```
s = path.concat(path_string1[, ..., path_stringN]);
```

Description

The `path.concat` function concatenates the values of multiple string expressions (***path_string1*** + ... + ***path_stringN***) into one string representing a file path. The first argument can be either a string expression or one of the path types documented in the `pathname` function. The combined paths are returned as an absolute, not relative, path.

Return Value

The function returns the concatenated path string.

Example

```
string image_filename = path.concat(Application, "../Images", maketext("%v%v.jpg", CLUSTER, HHNO
));
```

See also: Path Namespace, PathName Function, Concat Function

Path.GetDirectoryName Function

Format

```
s = path.getDirectoryName(path_string);
```

Description

The `path.getDirectoryName` function returns the directory name for the path specified in the string expression ***path_string***.

Return Value

The function returns a string with the directory name extracted from the full path.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: Path Namespace, Path.GetFileName Function

Path.GetExtension Function

Format

```
s = path.getExtension(path_string);
```

Description

The `path.getExtension` function returns the file extension for the path specified in the string expression ***path_string***. The extension will begin with a `"."`. If the file does not have an extension, a blank string is returned.

Return Value

The function returns the extension extracted from the full path.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: Path Namespace, Path.GetFileName Function, Path.GetFileNameWithoutExtension Function

Path.GetFileName Function

Format

```
s = path.getFileName(path_string);
```

Description

The `path.getFileName` function returns the file name for the path specified in the string expression ***path_string***. The file name will include the extension of the file; to get the file name without the extension, use the `path.getFileNameWithoutExtension` function.

Return Value

The function returns a string with the directory name removed from the full path.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: Path Namespace, Path.GetDirectoryName Function, Path.GetExtension Function, Path.GetFileNameWithoutExtension Function

Path.GetFileNameWithoutExtension Function

Format

```
s = path.getFileNameWithoutExtension(path_string);
```

Description

The `path.getFileNameWithoutExtension` function returns the file name for the path specified in the string expression ***path_string***. The file name will not include the extension of the file; to get the file name with the extension, use the `path.getFileName` function.

Return Value

The function returns a string with the directory name and extension removed from the full path.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: Path Namespace, Path.GetFileName Function, Path.GetExtension Function

Path.GetRelativePath Function

Format

```
s = path.getRelativePath(relative_to, path);
```

Description

The `path.getRelativePath` function returns a relative path based on the string expression ***path***'s location relative to a directory specified by ***relative_to***. The ***relative_to*** argument can be either a string expression or one of the path types documented in the `pathname` function.

Return Value

The function returns a relative path.

Example

```
string data_directory = "C:\CSPro Data Files";

string survey_data_path = path.concat(data_directory, "Region 1", "COVID-19 Survey.csdb");
// result: C:\CSPro Data Files\Region 1\COVID-19 Survey.csdb

string region_data_directory = path.getDirectoryName(survey_data_path);
// result: C:\CSPro Data Files\Region 1\

string survey_data_filename = path.getFilename(survey_data_path);
// result: COVID-19 Survey.csdb

string survey_data_extension = path.getExtension(survey_data_path);
// result: .csdb

string survey_data_filename_without_extension = path.getFilenameWithoutExtension(survey_data_pat
h);
// result: COVID-19 Survey

string survey_data_path_relative_to_data_directory = path.getRelativePath(data_directory, survey
_data_path);
// result: Region 1\COVID-19 Survey.csdb
```

See also: Path Namespace

Path.SelectFile Function

Format

```
s = path.selectFile(⌠message⌡);

s = path.selectFile(⌠title := message,⌡
                    ⌠filter := wildcard,⌡
                    ⌠startDirectory := start_directory_path,⌡
                    ⌠rootDirectory := root_directory_path,⌡
                    ⌠showDirectories := show_directories_flag⌡);
```

Description

The `path.selectFile` function shows the operator a visual listing of the file system and returns the path of the file the operator selected. Hidden and system files are not shown in the visual listing.

In the first version, an optional string expression ***message*** specifies the title of the dialog that shows the file listing. The operator will initially see files starting in the directory where the currently running application is located, and will have access to other files on that drive.

The second version, accessed by using named arguments, gives you control over the location and kind of files that the operator can choose. The string expression ***message*** specifies the title of the dialog that shows the file listing.

You can specify a string expression, ***wildcard***, that is applied to filter the files shown in the visual listing. For example, `"*.csdb"` would show only CSPro DB files. You can specify multiple wildcard filters by separating each filter with a semicolon. The filter `"*.jpg;*.jpeg;*.bmp"` would show both JPEG and BMP files. Instead of using a string expression, you can also use a predefined wildcard expression:

- `FileType.Audio`: Files that the audio object can read and write.
- `FileType.Geometry`: Files that the geometry object can read and write.
- `FileType.Image`: Files that the image object can read and write.

To specify the initial directory shown to the operator, you can specify a ***start_directory_path***. You can also specify a ***root_directory_path***. Specifying a root directory prevents the operator from moving to a location above the root directory. If neither are provided, the operator will initially see files starting in the directory where the currently running application is located, and will have access to other files on that drive. If a root directory is provided without a start directory, the start directory will be set to the root directory. If both a start and root directory are provided, the root directory must be equal to or above the start directory. Both directory paths can be one of the following:

- A string expression specifying a directory.
- One of the path types documented in the `pathname` function.
- A media type, which will show the files of the specified media type on Android. These files can reside in multiple directories.

By default, both files and directories are shown to the operator. To hide directories, set the numeric expression ***show_directories_flag*** to `false`.

Return Value

The function returns a string with the path of the selected file, or a blank string is no file was selected.

Example 1

```
string pff_filename = path.selectFile(title := "Select an Application to Execute",
                                      filter := "*.pff");

if pff_filename <> "" then
    execpff(pff_filename);
endif;
```

Example 2

```
string data_directory = path.concat(application, "../Data");
string current_region_data_directory = path.concat(data_directory, maketext("%v", REGION));

string data_filename = path.selectFile(title := "Select a Data File To Process",
                                       filter := "*.csdb",
                                       rootDirectory := data_directory,
                                       startDirectory := current_region_data_directory);
```

Example 3

```
string image_filename = path.selectFile(title := "Select Your Favorite Image",
                                        rootDirectory := Media.Images);
```

See also: Path Namespace, DirList Function

Pff Object

Pff Statement

Format

```
pff pff_name;
```

Description

The `pff` statement creates a dynamic pff with the name ***pff_name***. The pff name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare pffs globally in `PROC GLOBAL` or locally in functions or procedures.

Example

```
PROC INTERVIEWER_MENU

    // run the listing program in the interviewer's assigned cluster
    if INTERVIEWER_MENU = 1 then
        pff listing_pff;
        listing_pff.load("Listing.pff");
        listing_pff.setproperty("Key", maketext("%v", CLUSTER));
        listing_pff.exec();
    endif;
```

See also: Pff Object

Pff.Load Function

Format

```
b = pff_name.load(pff_filename || application_name);
```

Description

The `pff.load` function loads the contents from an existing PFF file into a pff. If supplying a string expression, ***pff_filename***, the properties from that PFF will be loaded. If supplying an ***application_name*** (the name of your application, which generally ends with _FF), then the PFF used to run the current application will be loaded.

Return Value

The function returns a logical value of 1 (true) if the PFF file exists and was successfully loaded and 0 (false) otherwise.

Example 1

```
// run the household data entry application but change the name
// of the data file to the current cluster
pff household_pff;
household_pff.load("Household.pff");
household_pff.setproperty("InputData", maketext("%v.csdb", CLUSTER));
household_pff.exec();
```

Example 2

```
function DisplayMinorMessage(string message_text)

    // do not display minor messages when running this batch
    // application and only showing summary messages
    pff this_application_pff;
    this_application_pff.load(CENSUS_FF);

    if this_application_pff.getproperty("ErrmsgOverride") <> "Summary" then
        errmsg("%s", message_text);
    endif;

end;
```

See also: Pff Object, Pff.Save Function

Pff.Save Function

Format

```
b = pff_name.save(pff_filename);
```

Description

The `pff.save` function saves the contents of the pff object to the disk to a PFF file with the name supplied by the string expression ***pff_filename***. Because pff objects can be executed without saving them, this function is not necessary, but may be useful if you want a copy of your PFF file on the disk.

Return Value

The function returns a logical value of 1 (true) if the PFF was successfully saved and 0 (false) otherwise.

Example

```
// create PFFs for running the listing application for each province
pff listing_pff;
listing_pff.load("Listing.pff");

do numeric counter = 1 while counter <= PROVINCE_VS.length()

    listing_pff.setproperty("Description", maketext("List in %s", PROVINCE_VS.labels(counter)));
    listing_pff.setproperty("Key", maketext("%02d", PROVINCE_VS.codes(counter)));

    string new_pff_filename = maketext("Listing%02d.pff", PROVINCE_VS.codes(counter));
    listing_pff.save(new_pff_filename);

enddo;
```

See also: Pff Object, Pff.Load Function

Pff.GetProperty Function

Format

```
s = pff_name.getproperty(property_name⌈, string_list⌋);
```

Description

The `pff.getproperty` function returns a string containing the current value associated with the pff's property specified by the string expression ***property_name***. PFF properties all have default values, so even if a property was not explicitly defined in the PFF file, it may have a default value. Some properties, such as input data filenames for batch applications, can have multiple values, and if you want to retrieve all of these values, you must supply a ***string_list***, which will contain zero, one, or more strings with the values.

Properties names can be prefixed with the section name, as explained in `pff.setproperty`, but it is generally not necessary to do so when using this function.

The function can be used similarly to `sysparm` to retrieve properties defined in a PFF's **[Parameters]** section.

Return Value

The function returns a string with the property information. If the property is invalid, a blank string is returned.

Example

```
// show our custom report unless the PFF specifies ViewListing=Never
pff this_application_pff;
this_application_pff.load(CENSUS_FF);

if this_application_pff.getproperty("ViewListing") <> "Never" then
    view("Disability Statistics.csv");
endif;
```

See also: Pff Object, Pff.SetProperty Function, SysParm Function

Pff.SetProperty Function

Format

```
b = pff_name.setproperty(property_name, property_value || string_list);
```

Description

The `pff.setproperty` function modifies the value associated with the pff's property specified by the string expression ***property_name***. The property will be modified to the value given in the numeric or string expression ***property_value***. If the ***property_value*** is blank, the property will be reset to its default value. Alternatively, a ***string_list*** can be supplied containing values for properties that have multiple values (such as input data filenames).

Property names can be prefixed with the section name. For example, if you load an existing PFF that has a persistent ID value defined for the item `PROVINCE`, you can code:

```
pff_name.setproperty("PROVINCE", 50);
```

However, if no such property was defined in the PFF, the `pff.setproperty` function will not know whether you are modifying a value in the **[DataEntryIds]** section, the **[ExternalFiles]** section, or another section. To avoid this ambiguity, when setting new custom properties, you can specify the section name when setting the property:

```
pff_name.setproperty("DataEntryIds.PROVINCE", 50);
```

Linking to Embedded Dictionaries

When running a data entry application from a .pen file, such as on an Android device, you may not have access to the dictionary file (.dcf) that may be necessary when running certain programs (such as the Compare Data tool). A special version of `pff.setproperty`, when used with the InputDict and OutputDict properties, accepts a dictionary name as an argument. If such an argument is used, certain tools (described in the `execpff` help) will be launched using the dictionary that is embedded in the .pen file.

```
pff_name.setproperty("InputDict", LISTING_DICT);
```

Return Value

The function returns a logical value of 1 (true) because it will always successfully set a property. If the property name is unknown, then the value is assigned to a new parameter (defined in a PFF's **[Parameters]** section).

Example 1

```
PROC INTERVIEWER_MENU

    // run the listing program in the interviewer's assigned cluster
    if INTERVIEWER_MENU = 1 then
        pff listing_pff;
        listing_pff.load("Listing.pff");
        listing_pff.setproperty("Key", maketext("%v", CLUSTER));
        listing_pff.exec();
    endif;
```

Example 2

```
PROC REPORT_SELECTION

    // run the batch application to generate a report on...
    pff report_pff;
    report_pff.load("..\Report\Report.pff");

    // ...the data file for the selected province and region
    if REPORT_SELECTION = 1 then
        report_pff.setproperty("InputData", maketext("..\Data\%v%v.csdb", PROVINCE, REGION));

    // ...the data files for all the regions in the selected province
    elseif REPORT_SELECTION = 2 then
        report_pff.setproperty("InputData", maketext("..\Data\%v*.csdb", PROVINCE));

    // ...the data files for the entire country
    else
        list string data_files_listing;
        dirlist(data_files_listing, "..\Data", filter := "*.csdb", recursive := true);
        report_pff.setproperty("InputData", data_files_listing);

    endif;

    report_pff.exec();
```

See also: Pff Object, Pff.GetProperty Function

Pff.Exec Function

Format

```
b = pff_name.exec();
```

Description

The `pff.exec` function starts another CSPro application using the properties defined in the pff object. If you modified any of the pff object's properties, it is not necessary to save the file to disk when running it, as this function will save a temporary file to the disk with the defined properties.

For data entry PFFs, the function executes the application and immediately closes the current application (without saving any data). This is the same as running `execpff` with the `stop` argument. For other PFFs, the function executes

the application using the `wait` argument. That is, this function is equivalent to:

```
pff my_pff;
// ...

if my_pff.getproperty("AppType") = "Entry" then
    execpff(filename(my_pff), stop);
else
    execpff(filename(my_pff), wait);
endif;
```

If you want to execute a data entry PFF without closing the current application, you can code:

```
execpff(filename(my_pff), wait);
```

Tools Running in the Same Process

The `execpff` help page includes information about how some tools launched via `pff.exec` and `execpff` are run in the same process as the calling application.

Return Value

The function returns a logical value of 1 (true) if the new application was started successfully and 0 (false) otherwise.

Example

```
PROC INTERVIEW_SELECTION_MENU

    pff application_pff;

    // conduct the main household interview
    if INTERVIEW_SELECTION_MENU = 1 then
        application_pff.load("..\Household\Household.pff");

    // conduct the agriculture interview
    else
        application_pff.load("..\Agriculture\Agriculture.pff");

    endif;

    // set the key to the household the user selected
    application_pff.setproperty("Key", SELECTED_HOUSEHOLD_KEY);

    // run the program
    application_pff.exec();
```

See also: Pff Object, ExecPFF Function

SystemApp Object

SystemApp Statement

Format

```
SystemApp system_app_name;
```

Description

The `SystemApp` statement creates a SystemApp object with the name ***system_app_name***. The SystemApp name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare SystemApp objects globally in `PROC GLOBAL` or locally in functions or procedures.

Example

```
// play the training video in Windows Media Player
SystemApp windows_media_player;
windows_media_player.setArgument(pathconcat("../Videos/Census Training.mp4"));
windows_media_player.exec("wmplayer.exe");
```

See also: SystemApp Object

SystemApp.SetArgument Function

Format

```
b = system_app_name.setArgument(argument[, value]);
```

Description

The `SystemApp.setArgument` function adds an argument to a SystemApp object. The argument will later be used when the system application is executed. The string expression ***argument*** and the optional string or numeric expression ***value*** are processed differently on Android and Windows platforms.

Return Value

The function returns a logical value of 1 (true).

Android Example

On Android, a system application is typically an Android activity. Each argument will be part of the "extras" passed in the Intent Bundle that opens the activity. If ***value*** is a string expression, it will be put as a String extra associated with the name ***argument***. If ***value*** is a numeric expression, it will be put as a double extra associated with the name ***argument***. If ***value*** is not specified, a null String extra will be used. There is special processing for three ***argument*** types:

- `"action"`: The ***value*** is used as the action when calling Intent.setAction.
- `"data"`: The ***value*** (which should be URI-encoded) is used as the data when calling Intent.setData.
- `"type"`: The ***value*** is used as the MIME data type when calling Intent.setType.

```
// get directions to the U.S. Census Bureau using Google Maps
SystemApp google_maps_navigation;
google_maps_navigation.setArgument("action", "android.intent.action.VIEW");
google_maps_navigation.setArgument("data", "google.navigation:q=U.S.+Census+Bureau");
google_maps_navigation.exec("com.google.android.apps.maps");
```

Windows Example

On Windows, a system application is an executable program. On Windows, every ***argument*** is turned into a command line argument passed to the executable. If an ***argument*** is provided without a corresponding ***value*** and the ***argument*** contains spaces and is not wrapped in quotes, the argument will be surrounded by quotes so that it is processed as a single argument by the executable. If a ***value*** is provided, then the argument and value are concatenated and will not be surrounded by quotes. If ***value*** is numeric, it will be converted to a string.

```
// plays the song in Windows Media Player, becoming:
// wmplayer.exe "Quick Escape.mp3"
SystemApp windows_media_player;
windows_media_player.setArgument("Quick Escape.mp3");
windows_media_player.exec("wmplayer.exe");

// opens the Census Bureau website in Chrome's Incognito mode, becoming:
// chrome.exe --args --incognito https://www.census.gov
SystemApp chrome;
chrome.setArgument("", "--args --incognito");
chrome.setArgument("https://www.census.gov");
chrome.exec("chrome.exe");
```

See also: SystemApp Object, SystemApp.Exec Function

SystemApp.Exec Function

Format

```
// Android format
b = system_app_name.exec(package_name, activity_name);

// Windows format
b = system_app_name.exec(executable_name);
```

Description

The `SystemApp.exec` function executes a system application associated with the SystemApp object. Any arguments set using the `SystemApp.setArgument` function are passed to the system application. On Android, a system application can return results which can be queried with the `SystemApp.getResult` function.

Return Value

The function returns a logical value of 1 (true) if the system application is started successfully and 0 (false) otherwise.

Android Example

On Android, the optional string expression ***package_name*** specifies the name of the Android application (package) that will be executed. An optional string expression ***activity_name*** specifies the name of the activity within the package that should be opened. If no arguments are provided, then the arguments specified using `SystemApp.setArgument` will be processed by the Android operating system and, if there is a way to handle these arguments, then an appropriate application will be opened.

```
// get directions to the U.S. Census Bureau using Google Maps
SystemApp google_maps_navigation;
google_maps_navigation.setArgument("action", "android.intent.action.VIEW");
google_maps_navigation.setArgument("data", "google.navigation:q=U.S.+Census+Bureau");
google_maps_navigation.exec("com.google.android.apps.maps");
```

Windows Example

On Windows, a system application is an executable specified by the string expression ***executable_name***. Typically the name is the full path of the executable, but for applications that have registered their name in the Windows Registry, you only need to provide the executable file name.

```
// play the training video in Windows Media Player
SystemApp windows_media_player;
windows_media_player.setArgument(pathconcat("../Videos/Census Training.mp4"));
windows_media_player.exec("wmplayer.exe");
```

See also: SystemApp Object, SystemApp.SetArgument Function, SystemApp.GetResult Function

SystemApp.GetResult Function

Format

```
s = system_app_name.getResult(argument);
```

Description

The `SystemApp.getResult` function returns a result from the system application that was executed using the SystemApp object. The function only works for system applications executed on Android devices.

Return Value

The function returns a string value with the result associated with the string expression ***argument***. If no result exists, then the function returns a blank string.

Android Example

```
SystemApp fingerprint_reader;
fingerprint_reader.setArgument("prompt", maketext("Please take the fingerprints for
%s", strip(NAME)));
fingerprint_reader.exec("com.example.entry_add_ons", "com.example.entry_add_ons.FingerPrintReade
rActivity");
FINGERPRINT_FILENAME = fingerprint_reader.getResult("filename");
```

See also: SystemApp Object, SystemApp.Exec Function

SystemApp.Clear Function

Format

```
b = system_app_name.clear();
```

Description

The `SystemApp.clear` function clears any stored arguments and results associated with a SystemApp object.

Return Value

The function returns a logical value of 1 (true).

Example

```
PROC GLOBAL

SystemApp google_maps;

PROC LOCATION

    // clear any previously used arguments
    google_maps.clear();

    // show barber shops near the location
    google_maps.setArgument("action", "android.intent.action.VIEW");
    google_maps.setArgument("data", maketext("geo:%f,%f?q=barber+shop", LATITUDE, LONGITUDE));
    google_maps.exec("com.google.android.apps.maps");
```

See also: SystemApp Object

ValueSet Object

ValueSet Statement

Format

```
valueset [valueset_type] valueset_name;
```

Description

The `valueset` statement creates a dynamic value set with the name ***valueset_name***. The value set name must be unique and must contain only letters, numbers, or the underscore character. The name must begin with a letter. You can declare value sets globally in `PROC GLOBAL` or locally in functions or procedures.

Value sets can be `numeric` or `string`. By default a value set is numeric, but the type can be modified by specifying the ***valueset_type***. When associating a value set with an item using `setvalueset`, the type must match the item type (numeric for numeric items, and string for alphanumeric items).

Example

```
PROC MENU

onfocus

    valueset menu_valueset;

    // interviewer options
    if USER_ROLE = 1 then
        menu_valueset.add("Add New Household", 1);
        menu_valueset.add("View Existing Households", 2);

    // supervisor options
    else
        menu_valueset.add("View Household Report", 3);

    endif;

    menu_valueset.add("Quit", 99);

    setvalueset(MENU, menu_valueset);
```

See also: ValueSet Object, List Object

ValueSet.Add Function

Format

```
i = valueset_name.add(label, from_code[, to_code || special_value]
                      [, image := image_filename]
                      [, textColor := color]);

i = valueset_name.add(valueset_name[, from_code][, to_code]);
```

Description

The `valueset.add` function adds one or more values to the end of a value set. The type of the added code or value set must match the type of the receiving value set.

In the first version, you provide a string expression ***label*** and either a numeric or string expression ***from_code***, which must match the type of the value set. For numeric value sets, it is possible to specify a numeric expression ***to_code***, which creates a range, or a ***special_value***, which is then associated with the ***from_code***. Using named arguments, you can provide a string expression, ***image_filename***, to specify the filename of an image to be shown as a value set image. You can also specify the text color of the value by providing a string expression ***color*** that contains a HTML color name (like `"red"`) or a hex color code (like `"#ff0000"`).

In the second version, you provide a ***valueset_name***. This adds all of the values from that value set to the receiving value set. If you specify a ***from_code*** or ***to_code***, then only the values in that range will be added to the receiving value set.

Return Value

The function returns the number of the values added to the value set. If the value set is read-only, the function returns `default`.

Example 1

```
PROC SUPERVISOR_MENU

onfocus

    valueset string menu_valueset = SUPERVISOR_MENU_VS;

    if CAN_SYNC = 1 then
        menu_valueset.add(SYNC_MENU_VS);
    endif;

    forcase HOUSEHOLD_DICT do
        menu_valueset.add("Review " + getcaselabel(HOUSEHOLD_DICT), key(HOUSEHOLD_DICT));
    endfor;

    setvalueset(SUPERVISOR_MENU, menu_valueset);
```

Example 2

```
PROC OCCUPATION_MINOR

onfocus

    numeric first_occupation_code = OCCUPATION_MAJOR * 100;
    numeric last_occupation_code = first_occupation_code + 99;

    valueset occupation_minor_valueset;

occupation_minor_valueset.add(OCCUPATION_MINOR_VS1, first_occupation_code, last_occupation_code)
;
    setvalueset(OCCUPATION_MINOR, occupation_minor_valueset);
```

See also: ValueSet Object

ValueSet.Remove Function

Format

```
i = valueset_remove.remove(code);
```

Description

The `valueset.remove` function removes all values from a value set that have a code matching the specified numeric or string expression ***code***. The type of the code must match the type of the value set.

When using a numeric value set with ranges—both **from** and **to** codes are defined—or defined special values, the function will remove a value only if it is the **from** value or if it is the special value.

Return Value

The function returns the number of values with the matching code that were removed from the value set. If the value set is read-only, the function returns `default`.

Example

```
PROC SECOND_FAVORITE_DRINK

preproc

    valueset drink_valueset = FAVORITE_DRINK_VS;

    drink_valueset.remove(FAVORITE_DRINK);

    setvalueset(SECOND_FAVORITE_DRINK, drink_valueset);
```

See also: ValueSet Object, ValueSet.Clear Function

ValueSet.Clear Function

Format

```
b = valueset_name.clear();
```

Description

The `valueset.clear` function removes all values from a value set.

Return Value

The function returns a logical value of 1 (true). If the value set is read-only, the function returns `default`.

Example

```
PROC GLOBAL

valueset name_valueset;

PROC LINE_NUMBER_OF_MOTHER

onfocus

    name_valueset.clear();

    // construct value set...
```

See also: ValueSet Object, ValueSet.Remove Function

ValueSet.Length Function

Format

```
i = valueset_name.length();
```

Description

The `valueset.length` function removes the number of values defined in the value set. When multiple values codes are linked to a single label, each multiple set counts as a single value.

Return Value

The function returns the length of the value set.

Example

```
PROC HOUSEHOLD_SELECTION

onfocus

    valueset households_vs;

    // ...

    if households_vs.length() = 0 then
        errmsg("There are no households to select");
        reenter MAIN_MENU;
    endif;
```

See also: ValueSet Object

ValueSet.Randomize Function

Format

```
b = valueset_name.randomize([exclude(exclude_code1, ..., exclude_codeN)]);
```

Description

The `valueset.randomize` function scrambles the order of values in the value set. The function is useful when using capture types in a data entry application. Some survey applications choose to display the possible response categories in a randomized way so as to minimize an enumerator or respondent's selection bias.

An optional exclusion list, ***exclude_code1***, ..., ***exclude_codeN***, allows you to prevent certain values from being given a random order. This is useful for variables like *Don't Know*, which, after the randomization, would remain at the original location in the list of values.

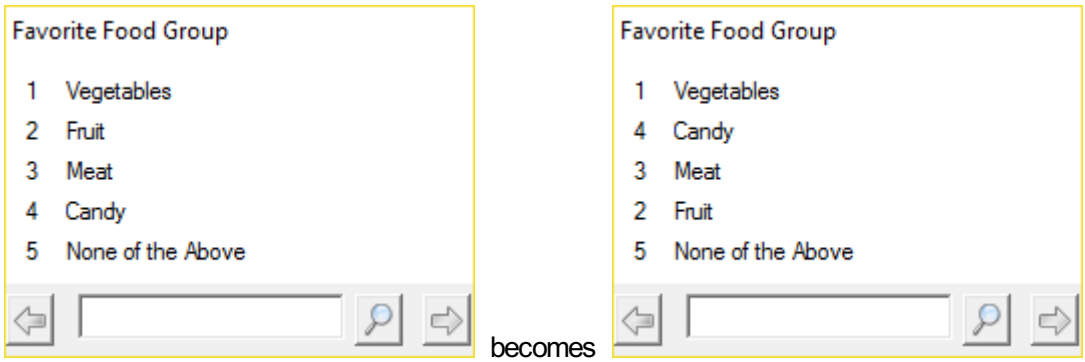
You can use the `seed` function to initialize the random number generation.

Return Value

The function returns a logical value of 1 (true).

Example

```
FAVORITE_FOOD_GROUP_VS1.randomize(exclude(5));
```



See also: ValueSet Object, Random Function, RandomIn Function, Seed Function, ValueSet.Sort Function

ValueSet.Show Function

Format

```
d = valueset_name.show([heading]);
```

Description

The `valueset.show` function displays the labels from a value set and returns the code (or index) of the operator's selection. An optional string expression, ***heading***, specifies the title of the window that displays the values. The function is similar to the `accept` function but with the options taken from the value set's labels. For numeric value sets, the returned value corresponds with the selected label's code. For a string value set, the returned value is the index of the selected label.

Return Value

The function returns the code (or index) of the item selected. If the selected value contains a special value, then that value is returned. The value 0 is returned if the escape key (or back button) is pressed and none of the values is chosen.

Example

```
valueset respondent_query;

do numeric counter = 1 while counter <= count(NAME)
    respondent_query.add(NAME(counter), LINE_NUMBER(counter));
enddo;

numeric respondent_line_number = respondent_query.show("Who in the household is responding to
questions?");
```

See also: ValueSet Object, Accept Function, Show Function, ShowArray Function, List.Show Function

ValueSet.Sort Function

Format

```
b = valueset_name.sort([ ascending || descending ] [ by code || label ]);
```

Description

The `valueset.sort` function sorts the order of entries in the value set by either label or code. An optional argument, `ascending` or `descending`, allows for the specification of the sort order. Another optional argument, `by code` or `by label` allows you to specify whether to sort the value set's entries by code or label. By default the value set will be

sorted in ascending order by label.

Labels are sorted in case insensitive order, so "c" is considered the same as "C". On the other hand, codes for alphanumeric value sets are sorted in case sensitive order, so "c" is recognized as different from "C".

Return Value

The function returns a logical value of 1 (true).

Example

```
valueset enumerators_vs;

forcase STAFF_DICT where STAFF_ROLE = 1 do
    enumerators_vs.add(STAFF_NAME, STAFF_CODE);
endfor;

enumerators_vs.sort();

setvalueset(ENUMERATOR_LOGIN, enumerators_vs);
```

See also: ValueSet Object, ValueSet.Randomize Function

Program Control Statements

Break Statement

Format

```
break;
```

Description

The `break` statement exits a `do`, `while`, `for`, or `forcase` loop early and continues execution with the first statement after the `enddo`.

Example

```
// find the spouse
numeric ptrSpouse;

for numeric ctr in PERSON do

    if P02_REL = 2 then
        ptrSpouse = ctr;
        break;
    endif;

enddo;
```

See also: Next Statement, Exit Statement

Do Statement

Format:

```
do [[varying] var = expression] while/until condition [by expression]
  statements;
enddo;
```

[ ] indicates that this part is optional.

Description:

The **do** statement executes one or more statements repeatedly, in a loop, either while a logical condition is true, or until a logical condition is no longer true. The **do** and **enddo** keywords are required. You must use a **while** or **until** phrase to terminate the loop. The condition is evaluated on each repetition of the loop before any of the statements within the loop are executed.

When the **while** option is used, it means the statements within the loop (between **do** and **enddo**) are executed **while** the condition remains true. That is, if the condition is true, the statements are executed. If the condition becomes false, execution moves to the first statement following the **enddo** keyword.

When the **until** option is used, the statements within the **do** are executed **until** the condition becomes true. That is, if the condition is false the statements are executed. If the condition becomes **true**, execution moves to the first statement following the **enddo** keyword.

The **by** phase adds the indicated number or numeric expression (expression) to the variable after each repetition of the loop. If the **by** phrase is present, at the end of each repetition of the loop, the expression is evaluated. The result of the expression is added to the numeric variable in the varying clause. If the **by** phrase is omitted, 1 is added to the

variable at the end of each repetition of the loop. For example, if you wanted to process only odd-numbered records, you could increment your loop **by** 2.

In the **varying** clause, the variable must be a numeric variable. The variable assignment is performed once, before the first repetition of the loop. The **varying** keyword has no effect on the command, and so may be omitted. It is possible to declare a numeric variable after the **varying** keyword, in which case the numeric variable's scope is the duration of the do loop.

You can exit the loop early by using `break` and you can continue execution with the next iteration of the loop by using `next`.

Example:

```
HEAD = 0;
do varying i = 1 until HEAD > 0 or i > totocc(PERSON)
  if RELATIONSHIP(i) = 1 then
    HEAD = i;
  endif;
enddo;
```

This same example could be rewritten using the **while** condition as follows:

```
HEAD = 0;
do varying i = 1 while HEAD = 0 and i <= totocc(PERSON)
  if RELATIONSHIP(i) = 1 then
    HEAD = i;
  endif;
enddo;
```

It is purely a matter of preference as to which method should be used.

See also: For Statement, While Statement, If Statement

Exit Statement

Format

```
exit [return_value];
```

Description

The `exit` statement terminates a procedure or user-defined function before normal processing is expected to end. When the statement is executed, processing stops for the current procedure or function, and control is passed to the next procedure or function.

When utilized in a user-defined function, an optional argument, ***return_value***, sets the function's return value before exiting the function. The type of the argument, either a numeric or string expression, must match the function's return type.

Example 1

```
function numeric FirstWoman()

    do numeric ctr = 1 while ctr <= HH_MEMBERS

        if SEX(ctr) = 2 then
            FirstWoman = ctr;
            exit; // exit the function, we've found our first woman!
        endif;

    enddo;

    FirstWoman = 0; // no woman was found

end;
```

Example 2

```
function string QueryForString(string message)

    while 1 do

        string response = prompt(message);

        if response = "" then
            errmsg("You cannot enter a blank value");

        else
            exit response;

        endif;

    enddo;

end;

// ...

errmsg("The non-blank value is %s", QueryForString("Enter a non-blank value"));
```

See also: Universe Statement, Skip Case Statement, Stop Function, EndCase Statement

For Statement

Format

```
for [counter in] multiple_occurring_group [where condition] do
    // statements
endfor;
```

Description

The `for` statement executes one or more statements repeatedly within a loop for each occurrence of a multiply occurring group. The ***multiple_occurring_group*** can be a form, roster, record, item, relation, or even a dictionary (following a `selcase` function call). The multiply occurring group controls how many times the `for` loop is executed. An optional logical ***condition***, referencing items in the group, can be supplied to restrict the cases processed by the loop.

The numeric variable ***counter*** contains the number of the current occurrence being examined. It cannot be changed inside the loop, but it can be referenced. Its starting value is 1 and its ending value is determined by the number of occurrences of the group. It is possible to declare a variable local to the loop by adding `numeric` before the ***counter*** variable.

If the group name is a record, item, or relation, then the appropriate keyword `record`, `item`, or `relation` can be used before the name.

The `for` statement should be coded outside of the group it references. In the example below, note that the code is executed in `PROC QUEST`. It should not be located in `PROC PERSON_REC_EDT` or in a procedure for any of the data items within the person record.

Example

```
PROC QUEST

    numeric pointer_spouse;
    numeric pointer_oldest_child;

    for numeric occurrence_number in PERSON_REC_EDT do

        if RELATIONSHIP = 2 then
            pointer_spouse = occurrence_number;

        elseif RELATIONSHIP = 3 then
            if pointer_oldest_child = 0 or AGE > AGE(pointer_oldest_child) then
                pointer_oldest_child = occurrence_number;
            endif;

        endif;

    endfor;
```

See also: ForCase Statement, For (Dictionary) Statement, Do Statement, While Statement

ForCase Statement

Format

```
forcase dictionary_name [where condition] do
    // statements
endfor;
```

Description

The `forcase` statement executes one or more statements repeatedly within a loop for each case in a file opened as external dictionary ***dictionary_name***. An optional logical ***condition***, referencing items in the external dictionary, can be supplied to restrict the cases processed by the loop.

It is possible, by specifying dictionary access parameters after the dictionary name, to limit the cases that the `forcase` loop processes.

Example

```
numeric incomplete_households;

forcase CENSUS_DICT where INTERVIEW_STATUS = 3 do
    inc(incomplete_households);
endfor;
```

See also: For Statement, For (Dictionary) Statement, CountCases Function, LoadCase Function

For (Dictionary) Statement

Format

```
for dictionary_name[(mark_type)] do
    // statements
endfor;
```

Description

The `for` (dictionary) statement executes one or more statements repeatedly within a loop for each selection made during a `selcase` function call. This allows an operator to select some number of cases and then allows you to load each case and perform some analysis on the selected cases. Unlike the `forcase` loop, which operates on all cases in a data file, this loop only operates on selected cases.

The ***dictionary_name*** must be supplied and refers to an external dictionary associated with your application. An optional argument, ***mark_type***, indicates what kinds of cases to process, and can be one of the following values:

mark_type	Description
marked	Process only the cases selected by the operator during the `selcase` function call.
unmarked	Process the cases **not** selected by the operator.
all	Process all of the cases that could have been selected by the operator.

If not specified, the ***mark_type*** defaults to **marked**.

Example

```
selcase(OCCUPATION_DICT,"Plantation") multiple;

numeric maxEducation;

for OCCUPATION_DICT do
    maxEducation = high(maxEducation,MAXIMUM_EDUCATION);
endfor;
```

See also: For Statement, ForCase Statement, NMembers Function

If Statement

Format:

```
if condition then
  statements;
[elseif condition then
  statements;]
[else
  statements;]
endif;
```

[ ] indicates that this part is optional.

Description:

The **if** statement executes different statements based on the value of "condition". The condition following the **if** command is evaluated. If the condition is **true**, then the statements following it are executed and execution moves to the first statement after the **endif** keyword. If the condition is **false**, execution moves to the first **elseif** keyword or the **else** keyword (if there are no **elseif** keywords).

The **elseif** blocks are evaluated in the same way as the first **if** block. When CSPro finds a condition that is **true** it executes the statements following it and moves to the first statement after the **endif** keyword. If all the conditions are **false**, the statements following the **else** keyword are executed. If none of the conditions are true and there is no **else** keyword, execution moves to the first statement after the **endif** keyword without the execution of any statements within the **if** statement.

Every **if** statement must contain an **endif** keyword. However, if multiple **elseif** keywords are nested within an **if** block, they may be terminated with a single **endif** keyword. The statements within the **if** statement can be any number of

CSPro statements. If a condition contains an inequality (e.g., >, <, >=, <=) and one of the values tested in the inequality is a special value (e.g., MISSING, REFUSED, NOTAPPL, or DEFAULT), the result of the condition is false and execution skips to the statement following the else.

Example:

```
if X = 3 then
  z = 6;
elseif x in 4:5 or y in 7:9,12 then
  z = 7;
else
  z = 8;
endif;
```

See also: When Statement, Recode Statement

Next Statement

Format

```
next;
```

Description

The `next` statement ends a `do`, `while`, `for`, or `forcase` loop early and continues execution with the next iteration of the loop. If the next iteration results in the termination of the loop, then execution will begin with the first statement after the `enddo`.

Note: Prior to CSPro 7.6, if you were within a data entry application's roster and wrote `next` to skip to the next iteration of the roster, it would compile, but did not work. Checking is more stringent now, and an error message will be generated. You must write `skip to next` to accomplish the skip.

Example

```
// fill an array with the occurrence numbers of all spouses
spouseNumber = 0;

for numeric ctr in PERSON do

    if P02_REL <> 2 then
        next;
    endif;

    inc(spouseNumber);
    spouseIndices(spouseNumber) = ctr;

enddo;
```

See also: Break Statement, Exit Statement

Universe Statement

Format

```
universe condition [case];
```

Description

The `universe` statement determines whether to allow normal execution of a procedure or function based on the value of

condition. The ***condition*** following the universe command is evaluated. If the condition is true, then the statements following it are executed. If the condition is false, processing stops for the current procedure or user-defined function, and control is passed to the next procedure or user-defined function.

In batch editing mode, if the optional `case` keyword is specified and the condition is false, the program will stop processing the case but will still write it to the output file.

Example 1

```
PROC FERTILITY

    universe SEX = 2 and AGE in 12:49;
```

Example 2

```
PROC ROOF_TYPE

    universe HHTYPE = 1 case;

    // the universe is equivalent to:
    if not HHTYPE = 1 then
        endcase;
    endif;
```

See also: Exit Statement, EndCase Statement, If Statement

When Statement

Format

```
when expression1 [:: expression2 :: expressionN];
     [value1]    [:: value2      :: valueN]       -> statement;
                                                 [-> else_statement;]
endwhen;
```

Description

The `when` program control statement executes a statement based on the value of one or more other variables, combining `if` statements with the power of `recode`. The statement is similar to statements in other programming languages (like **switch** in C or **when** in Kotlin).

One or more expressions, ***expression1*** to ***expressionN***, are given, with each expression separated by two colons **::**. These expressions must evaluate to either a number or string. Based on the values of each expression, CSPro evaluates each line between the `when` and `endwhen`, trying to match a line's values, ***value1*** to ***valueN***, with the evaluated expressions. Once a line matches, a statement, which is given after the arrow **->**, is executed and program control moves to statements following the `endwhen`. Only a single statement can be provided per line.

Each value provided must evaluate to the same type (number or string) as its respective expression. A value omitted is considered a match, and if no values are provided, then the optional ***else_statement*** is executed (assuming no preceding line matched).

Values can be provided in the following ways:

- An individual value. For example: `50`
- Using the syntax of in lists, meaning that multiple values can be separated with commas and ranges be separated by colons. For example: `1,3,5:9`
- An individual value preceded by one of the following operators: <, <=, >, >=, =, <>. If such an operator is used, the expression's value is compared with the value using the operator. For example: `>= 65`

Example 1

```
function SimpleWhenExample(numeric sex, numeric age)

    when sex :: age;
         1   ::        -> errmsg("Male");
         2   ::  0: 11 -> errmsg("Young female");
         2   :: 12: 49 -> errmsg("Fertile female");
         2   :: 50:120 -> errmsg("Older female");
                       -> errmsg("Invalid sex or age");
    endwhen;

end;

SimpleWhenExample(2,  10); // Young female
SimpleWhenExample(2,  38); // Fertile female
SimpleWhenExample(1, 150); // Male
SimpleWhenExample(2, 150); // Invalid sex or age
```

Example 2

```
PROC EMPLOYMENT_STATUS

    when EMPLOYMENT_STATUS;
          1:2 -> skip to UNEMPLOYMENT;
         <= 6 -> skip to INFORMAL_EMPLOYMENT
         >= 7 -> skip to FORMAL_EMPLOYMENT;
    endwhen;
```

See also: If Statement, Recode Statement

While Statement

Format

```
while condition do
    // statements
enddo;
```

Description

The `while` statement executes one or more statements repeatedly, in a loop, while the logical ***condition*** is true. Unlike a `do` loop, a counter is not automatically incremented, so you must ensure the termination of the loop. The condition is evaluated on each repetition of the loop before any of the statements are executed.

You can exit the loop early by using `break` and you can continue execution with the next iteration of the loop by using `next`.

Example

```
setfile(inputFile, "Countries.txt");

string countriesLine;

while fileread(inputFile, countriesLine) do
    // ...
enddo;
```

See also: Do Statement, For Statement

Assignment Statements

Assignment Statement

Format

```
numeric_variable = numeric_expression;

string_variable = string_expression;
```

Description

The assignment statement sets a variable equal to the value of an expression. If the expression is numeric or conditional, then the variable must be numeric. If the expression is a string expression, then the variable must be alphanumeric.

Examples

```
AGE = 10;
Q102 = PREV_AGE;
Y = sqrt(X);

NAME = "John Doe";
NAME = "John " + toupper("Doe");
```

See also: SetValue Function

Recode Statement

(Prior to CSPro 7.4, a different version of the recode/box command function existed. That version should no longer be used as it will be removed in a future release.)

Format

```
recode expression1 [:: expressionN] -> destination_variable1 [:: destination_variableN];
       value1      [:: valueN]      -> result1                [:: resultN];
                                   [-> else_result1           [:: else_resultN];]
endrecode;
```

Description

The `recode` statement assigns a value to one or more output variables based on the value of one or more other input variables. It can be used to rescale variables, to assign values to variables, or to create a composite variable from existing variables. In many instances it is easier to use than writing multiple if statements.

One or more expressions, ***expression1*** to ***expressionN***, are given, with each expression separated by two colons **::**. These expressions must evaluate to either a number or string. Based on the values of each expression, CSPro evaluates each line between the `recode` and `endrecode`, trying to match a line's values, ***value1*** to ***valueN***, with the evaluated expressions. Once a line matches, one or more results, ***result1*** to ***resultN***, which are given after the arrow **->**, are assigned to ***destination_variable1*** to ***destination_variableN***, which can be dictionary items, working variables, array values, list values, or the return values of a user-defined function. A destination variable can also be included among the expressions. It is possible to declare a new `numeric`, `alpha`, or `string` destination variable as part of the recode statement (see Example 2 below).

Each value provided must evaluate to the same type (number or string) as its respective expression, and each result

must be the same type as the destination variable. A value omitted is considered a match, and if no values are provided, then the optional ***else_result1*** to ***else_resultN*** are used in the assignment (assuming no preceding line matched).

Values can be provided in the following ways:

- An individual value. For example: `50`
- Using the syntax of in lists, meaning that multiple values can be separated with commas and ranges be separated by colons. For example: `1, 3, 5:9`
- An individual value preceded by one of the following operators: <, <=, >, >=, =, <>. If such an operator is used, the expression's value is compared with the value using the operator. For example: `>= 65`

Example 1

```
recode AGE    -> AGE_GROUP;
        0:19 -> 1;
       20:29 -> 2;
       30:39 -> 3;
       40:49 -> 4;
       >= 50 -> 5;
             -> 9;
endrecode;
```

Example 2

```
PROC AGE

    recode RELATIONSHIP :: AGE(1) - AGE -> numeric valid_age_difference_with_head;
           3, 4         :: <  12        -> false; // biological and step children must be 12+
years younger
           6            :: > -12        -> false; // parents must be 12+ years older
           7            :: <  24        -> false; // grandchildren must be 24+ years younger
                                        -> true;
    endrecode;
```

Example 3

```
function string GetDisplayName()

    recode NAME -> GetDisplayName;
           ""   -> "(Undefined)";
                -> strip(toupper(NAME));
    endrecode;

end;
```

Example 4

```
recode INDUSTRY_SECTION -> min_division :: max_division;
                    "A" ->            1 ::  3;
                    "B" ->            5 ::  9;
                    "C" ->           10 :: 33;
                    // ...
endrecode;
```

See also: If Statement, When Statement

Recode Statement (Deprecated)

***Feature Upgrade**: Starting with CSPro 7.4, this version of recode has been superseded by a new version, `recode`. This version will be removed in a future version of CSPro. To update to the new version, you must change three things:*

1. *Replace => with ->*
2. *When using multiple variables, replace : with ::*
3. *Modify ranges from using - to using : (as with in lists)*

For example, this shows the differences between using recode prior to CSPro 7.4 and using it starting with CSPro 7.4:

```
// prior to CSPro 7.4
recode SEX :   AGE => ASK_FERTILITY;
         2 : 12-49 => 1;
           :       => 0;
endrecode;

// CSPro 7.4+
recode SEX ::   AGE -> ASK_FERTILITY;
         2 :: 12:49 -> 1;
                    -> 0;
endrecode;
```

The following documentation is for the old recode statement.

Format:

```
recode var-1 [:var-2 [:var-n]]    => var-out;
      [range-1] [:range-2 [:range-n]]   => exp;
      [range-1] [:range-2 [:range-n]]   => exp;
          : : :
                 [: [:]]   => other-exp;
endrecode;
```

[ ] indicates that this part is optional.

Description:

The **recode** statement assigns a value to a variable based on the value of one or more other variables. It is used to rescale variables, to assign values to variables, and to create new variables from existing ones. It works like a multiple if statement but is easier to use. The recode statement evaluates each line within it sequentially, beginning with the first line.

If the values of variables "var-1" to "var-n" lie within the ranges "range-1" to "range-n", respectively, then "var-out" is assigned the value given by the expression on the first line and the **recode** statement is ended. If the values of the variables "var-1" to "var-n" do not all lie within their specified ranges, then the next line of the **recode** statement is evaluated. This process continues until either a value is assigned to "var-out" or the end of the **recode** statement is reached.

A variable in a multiple record or group cannot be used in the **recode** statement except in data entry applications (where it may be specified without an index and the current occurrence of a variable is assumed). Use working variables to refer to or to assign values to variables in multiple sections or groups.

Variables "var-1" through "var-n" are referred to as independent variables and must be separated by colons [**:**]. "Var-out", the variable whose value is assigned by the recode statement, is referred to as the dependent variable. A **recode** statement can have any number of independent variables, but only one dependent variable. The dependent variable can also be included among the independent variables. The dependent variable is separated from the independent variables by **=>**.

The ranges specified in the **recode** statement (i.e., "range-1" through "range-n") can take the following formats:

- A range between two values, e.g., 12-15
- An individual value, e.g., 9
- A comparison with another value, using >, <, >=, <=, or <>, e.g., < 5
- A special value , e.g., NOTAPPL
- Some combination of these formats separated by a comma, e.g., < 5, 9, 12-15, missing

A blank range for an independent variable includes all values. A blank range for all independent variables on the last

line of a **recode** statement acts as a catch-all condition. It ensures that a value is always assigned to "var-out" by the recode statement. If a value is not assigned by the **recode** statement, the value of "var-out" will not change. The number of ranges on each line must equal the number of independent variables.

The expression for the dependent variable must result in a numeric value (if "var-out" is a numeric variable) or a string (if "var-out" is an alphanumeric variable).

(Note to ISSA users: The Recode and Box statements are identical.)

Example 1:

```
recode AGE => AGE_GROUP;
       0-19 => 1;
      20-29 => 2;
      30-39 => 3;
      40-49 => 4;
      >= 50 => 5;
            => 9;
endrecode;
```

is equivalent to the following if statements:

```
if AGE in 0:19 then
  AGE_GROUP = 1;
elseif AGE in 20:29 then
  AGE_GROUP = 2;
elseif AGE in 30:39 then
  AGE_GROUP = 3;
elseif AGE in 40:49 then
  AGE_GROUP = 4;
elseif AGE >= 50 then
  AGE_GROUP = 5;
else
  AGE_GROUP = 9;
endif;
```

Example 2:

```
recode ATTEND : ED_LEVEL => EDUC;
    2,notappl :          => 1;
            1 : 1        => 2;
            1 : 2,3      => 3;
              :          => 9;
endrecode;
```

```
is equivalent to the following if statements:
```

```
if (ATTEND = 2 or ATTEND = notappl) then
  EDUC = 1;
elseif ATTEND = 1 then
  if ED_LEVEL = 1 then
    EDUC = 2;
  elseif ED_LEVEL in 2:3 then
    EDUC = 3;
  endif;
else
  EDUC = 9;
endif;
```

Example 3:

```
recode UNITS : NUMBER  => DAYS;
             : notappl => notappl;
             : missing => missing;
           1 :         => NUMBER;
```

```
            2 :          => NUMBER*7;
            3 :          => NUMBER*30;
            4 :          => NUMBER*365;
              :          => missing;
endrecode;
```

is equivalent to the following if statements:

```
if NUMBER = notappl then DAYS = notappl;
elseif NUMBER = missing then DAYS = missing;
elseif UNITS = 1 then DAYS = NUMBER;
elseif UNITS = 2 then DAYS = NUMBER*7;
elseif UNITS = 3 then DAYS = NUMBER*30;
elseif UNITS = 4 then DAYS = NUMBER*365;
else DAYS = missing;
endif;
```

See also: If Statement

Impute Function

Format

```
d = impute(item_name, new_value)
    [title(frequency_table_title)]
    [valueset(valueset_name)]
    [specific]
    [stat([item_name1, ..., item_nameN])];
```

Description

The `impute` function assigns a new value to an item. The ***item_name*** is a dictionary item, either numeric or alphanumeric, and ***new_value*** is an expression that matches the type of the item. The function is similar to using the assignment operator:

```
item_name = new_value;
```

However, unlike using the assignment operator, the `impute` function keeps track of these assignments and generates a report on the frequency of values used in the imputations. These imputation statistics are useful when cleaning data in a batch application. If your program contains any `impute` statements, the results of this function will be written a frequencies file. The default file extension is .impute_freq.lst, but you can use whatever extension you prefer.

The function has several optional arguments:

Specify a frequency title (`title`): If supplying a string expression as a ***frequency_table_title***, this title will be used when the frequency writer creates the imputation frequencies. If no title is specified, a default title such as "Imputed Item SEX: Sex" will be used.

Specify a frequency value set (`valueset`): By default, when the frequency writer creates the imputation frequencies, it shows each value imputed and it looks at the item's primary value set, if one exists, to find a label that matches the value. If you would prefer to use a different value set when creating the imputation frequencies, you can specify a ***valueset_name*** that belongs to the item.

Create a specific frequency table (`specific`): Typically, if you have multiple `impute` statements for one item (with the same `valueset` setting), only one frequency table will be written, with the frequencies for all imputations combined. Even if differing titles are specified, one table will be written, with the title coming from the last executed imputation. If you would like a frequencies table for a particular imputation statement, you can use the `specific` command to indicate that a frequency table should be created for that imputation.

Impute Stat Data

Create a data file with frequency details (`stat`): If you want more details about imputations beyond the frequencies showing the imputed values, you can use the `stat` command to generate a data file that will contain information about each imputation. For each case in the input data that results in any imputations, the stat data file will contain an entry with the case IDs, the original value of the imputed item, the new value used in the imputation, and the line number of the `impute` statement that resulted in the imputation. For example:

```
impute(SEX, 2)
stat();
```

This would result in a data file with a record, `IMPUTE_SEX_REC`, with three items: `IMPUTE_SEX_INITIAL` (the initial value of `SEX`), `IMPUTE_SEX_IMPUTED` (the imputed value; in this case `2`), and `IMPUTE_SEX_LINE_NUMBER` (the line number of the imputation).

If you would like to see the value of other items that might be useful during analysis, you can specify ***item_name1***, ***item_name2***, and so on. The values of these items will be included in the stat data file. For example:

```
impute(EDUCATION, getdeck(educationHotdeckBySexAge))
stat(SEX, AGE);
```

By default, the only entries written to the stat data file are imputations where `stat` is included as part of the `impute` statement. Alternatively, you can specify an override:

```
set impute(stat, on || off || default);
```

If an override is coded, any `impute` statement that follows will either:

- `on`: automatically be included in the stat data file as if `stat()` were coded.
- `off`: any `stat` commands will be ignored.
- `default`: behave using the default behavior, where the stat data file only includes entries for imputations with `stat` commands.

Imputation Files

Applications using the `impute` function can generate up to three files:

- Imputation frequencies (with the default extension .impute_freq.lst)
- Imputation stat dictionary (if using `stat`, with the default extension .impute_stat.dcf)
- Imputation stat data (if using `stat`, with the default extension .impute_stat.csdb)

You can specify the names of these data files in the File Associations dialog or in your application's PFF file.

The frequencies report contains five columns:

Imputed Item SEX: Sex

Categories	Frequency	CumFreq	%	Cum %
1 Male	271	271	52.9	52.9
2 Female	241	512	47.1	100.0
Total	512	512	100.0	100.0

- **Categories**: Lists the values that were assigned during the imputations and a value set label for the value (if applicable). For example: "2 Female."
- **Frequency**: Shows the frequency (that is, the total number of times) each value was assigned. For example: 241 (code 2 assigned 241 times).

- **CumFreq**: Displays the cumulative totals of the Frequency column.
- **%**: Indicates what percentage each imputation represents from the total number of imputations made. For example: 47.1 (code 2 assigned 47.1% of the total number of imputations of `SEX` made).
- **Cum %**: Displays the cumulative totals of the % column.

Return Value

When imputing a numeric item, the function returns the numeric expression ***new_value***. When imputing an alphanumeric item, the function returns 1 (true).

Example

```
PROC SEX

if not invalueset(SEX) then

    // set all heads to men and everyone else to the opposite of the head's sex
    // (note that this is not a good imputation but is just a simple example)
    if curocc() = 1 then
        impute(SEX, 1)
        title("Head's Sex")
        specific;

    else
        impute(SEX, 3 - SEX(1));

    endif;
```

See also: Imputation

SetValue Function

Format

```
b = setvalue(variable_name, value[, occurrence1, ..., occurrence3]);
```

Description

The `setvalue` function executes an assignment when the name of the variable to be assigned to is not known at compilation time (when writing the application). The function searches for a variable with a name stored in the string expression ***variable_name***, and if such a variable is found, the function assigns to it the value stored in ***value***. If the variable is numeric, then ***value*** should be a numeric expression, and if the variable is alphanumeric, then it should be a string expression. Optional arguments, ***occurrence1*** to ***occurrence3***, allow you to pass occurrence numbers to the function. This function is especially useful when combined with a `userbar` button or an `OnKey` character sequence.

Return Value

The function returns a logical value of 1 (true) if the variable name was found and the value was successfully changed. The function returns 0 (false) if the variable was not found, if a string expression was assigned to a numeric variable, or if a numeric expression was assigned to an alphanumeric variable.

Examples

```
// the following statements are equivalent but you would not use setvalue
// because the left-hand side of the assignment is already known

setvalue("NAME", "Amelia");
NAME = "Amelia";

setvalue("AGE", 20);
AGE = 20;

// below, the left-hand side of the assignment is not known at compilation time

setvalue(getsymbol(), missing); // set the current field to missing

setvalue(getsymbol(), "DK", 3); // set the third occurrence of the current field to "DK"
```

See also: GetValue Function, Assignment Statement

GetValue Function

Formats

```
d = getvalue(variable_name⌠, occurrence1, ..., occurrence3⌡);

d = getvaluenumeric(variable_name⌠, occurrence1, ..., occurrence3⌡);

s = getvaluealpha(variable_name⌠, occurrence1, ..., occurrence3⌡);
```

Description

The `getvalue` function returns the value of a variable when the variable name is not known at compilation time (when writing the application). The function searches for a variable with a name stored in the string expression ***variable_name***, and if such a variable is found, the function returns the value of that variable.

The functions `getvalue` and `getvaluenumeric` return the value of numeric variables. The function `getvaluealpha` returns the value of an alphanumeric variable. Optional arguments, ***occurrence1*** to ***occurrence3***, allow you to pass occurrence numbers to the function.

Return Value

The function returns the variable's value if the variable name was found. If a numeric variable was not found, the functions `getvalue` and `getvaluenumeric` return `default`. If an alphanumeric variable was not found, `getvaluealpha` returns a blank string.

Example

```
errmsg("The current field's value is %d", getvalue(getsymbol()));
```

See also: SetValue Function

Data Entry Statements and Functions

Accept Function

Format 1

```
i = accept(heading, choice1[, choice2, ... choiceN]);
```

Format 2

```
i = accept(heading, choices_list || choices_array);
```

Description

The `accept` function displays a menu with a heading and a list of choices. The operator can use the down- or up-arrow keys to select the desired choice and press Enter. The operator can also use the mouse to click on the desired choice.

The ***heading*** text is a string expression. In the first version of the function, each choice (***choice1*** to ***choiceN***) is provided as a string expression. In the second version of the function, the choices can be provided using a string list or a string or alphanumeric array. With an array, the number of choices will be calculated dynamically. Each element of the array will be displayed until a blank string is found in the array.

Return Value

The function returns the index of the choice selected: 1 for the first choice, 2 for the second choice, etc. The value 0 is returned if the Escape key (or back button) is pressed and none of the choices is selected.

Example 1

```
numeric manual_choice = accept("Open which manual?", "Supervisor", "Interviewer");

if manual_choice = 1 then
    view("../Manuals/Supervisor.pdf");

elseif manual_choice = 2 then
    view("../Manuals/Interviewer.pdf");

endif;
```

Example 2

```
list string household_names;

do numeric ctr = 1 while ctr <= count(NAME)
    household_names.add(NAME(ctr));
enddo;

numeric interview_index = accept("Select a Person to Interview", household_names);
```

See also: Prompt Function

Advance Statement

Format

```
advance [to] [field_name];
```

Description

The advance statement moves forward, field by field, to the specified field, executing logic in fields' preproc and postproc events as it proceeds. It acts as though the Enter key were pressed repeatedly until either the specified field appears or one of the procedures executed during the advance goes to a different field.

The optional ***field_name*** is either the name of a field or is a string variable specifying the name of the field. If the ***field_name*** is not specified, the program attempts to advance to the end of the level.

The target field can be located in any record at the same level as the current field, but it cannot be located at a different level. The field must be later on the path than the current field, meaning that it is a field that has not yet been entered. If the field has already been entered, an error message will be displayed during data entry. If you do not know whether the field is earlier in the data path, use the move statement.

Note that the advance statement behaves differently from the skip statement, moving *over* some number of fields, rather than skipping *past* the fields.

If you want to conditionally execute logic based on whether or not the program is advancing, the inadvance function indicates if an advance is in progress.

Example

```
PROC CEB

preproc

    // for girls aged less than 15, prefill in 0 for children ever born, but
    // keep the information on path in case the keyer wants to change it
    if AGE < 15 then
        CEB = 0;
        CEB_M = 0;
        CEM_F = 0;
        advance to MARITAL_AGE;
    endif;
```

See also: InAdvance Function, Ask Statement, Move Statement, NoInput Statement, Reenter Statement, Skip Statement

Ask Statement

Format

```
ask if condition;
```

Description

The ask statement determines whether a field, roster, or form is eligible for data entry. The statement must be coded in a preproc. The ***condition*** is evaluated, and if true, any additional logic is executed until the appropriate field is reached and entered. If the ***condition*** is false, then control moves to the next field, roster, or form.

The ask statement is an inversion of the skip statement and can be written as:

```
if not condition then
    skip to next_field;
endif;
```

Example

```
PROC MARRIAGE_AGE

preproc

    ask if MARITAL_STATUS in 2:5;
```

See also: Advance Statement, Move Statement, Reenter Statement, Skip Statement

ChangeKeyboard Function

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setproperty` function with the argument `"Keyboard"`.*

Format

```
i = changekeyboard(dictionary_symbol[,keyboard_id]);
```

Description

The `changekeyboard` function modifies the keyboard input associated with a field on a form. The value ***dictionary_symbol*** may be an item, group, form, or an entire dictionary. The numeric expression ***keyboard_id*** refers to a keyboard ID (which can be determined using the Change Keyboard Input dialog box). A keyboard ID of 0 will reset the field to the default keyboard for the application.

Calls to this function are ignored when executed on mobile devices.

Return Value

The function returns the number of items whose capture positions were successfully changed. If a keyboard ID is not specified, the function will instead return the current keyboard ID associated with a given field.

Example

```
PROC NAME

onfocus

    // use the Dvorak keyboard to key in the name
    changekeyboard(NAME,66569);
```

See also: Change Keyboard Input

Connection Function

Format

```
b = connection([WiFi || Mobile]);
```

Description

The connection function determines whether a device has access to the Internet. If called with no arguments, then the function determines whether the device can connect to the Internet via Wi-Fi or via a mobile network like 3G, 4G, or GPRS. Optional arguments (***WiFi*** and ***Mobile***) allow you to determine if a given connection is valid on mobile CSEntry. On desktop CSEntry, the optional arguments are ignored and the function returns whether a connection can be made via a LAN, a Wi-Fi connection, or a mobile network connection.

Return Value

The function returns a logical value of 1 (true) if a connection exists and 0 (false) otherwise. The presence of a connection does not guarantee access to the Internet, however. For example, a device could be connected to a Wi-Fi router that does not have an Internet connection. The function would still return 1 in that case.

Example

```
// only sync when connected on Wi-Fi
if connection(WiFi) then
    if syncconnect(Dropbox) then
        syncdata(PUT, HOUSEHOLDQUESTIONNAIRE_DICT);
        syncdisconnect();
    endif;
endif;
```

DeMode Function

Format

```
i = demode();
```

Description

The demode function returns the data entry mode and is often used to limit the execution of certain statements to a specific mode. For example, a variable may need initialization when the operator invokes **add** or **verify** mode, but can be left unaltered for **modify** mode.

Return Value

The function returns an integer but there are also aliased values for the modes. There are three data entry operator modes:

Return Value	Mode
1	**add** (entering new cases)
2	**modify** (modifying cases that have already been entered)
3	**verify** (reentering cases as a check for differences between the first and second entry)

Example

```
if demode() = add then
    INTERVIEW_START_DATE = sysdate("YYYYMMDD");
endif;
```

Display Orientation

Display Orientation

In a data entry application that uses a rich visual interface, you may want to exert programmatic control over the

orientation of the screen, e.g., whether the display is horizontal or vertical. For computers with video cards that support changing the display orientation, which includes most new computers as well as tablet PCs, two functions can be used to control the orientation. Calls to these functions are ignored when executed on mobile devices.

Format:

```
b = getorientation();
```

Description:

The **getorientation** function returns the current display orientation. The function will return one of four values:

- 0: The natural orientation of the display device
- 90: The display orientation is rotated 90 degrees from the natural orientation
- 180: The display orientation is rotated 180 degrees from the natural orientation
- 270: The display orientation is rotated 270 degrees from the natural orientation

SetOrientation Function

Format:

```
b = setorientation(numeric-expression);
```

Description:

The **setorientation** function changes the orientation of the display. The numeric expression must be one of the four values listed above. The function returns 1 if the display orientation was successfully changed, 0 otherwise.

Example:

```
function rotateScreen()

    numeric nextOrientation = getorientation() + 90;

    if nextOrientation = 360 then
        nextOrientation = 0;
    endif;

    setorientation(nextOrientation);

end;
```

EditNote Function

Format

```
s = editnote([dictionary_symbol][,operator_id]);
```

Description

The `editnote` function displays the note entry dialog box for adding or changing the note for a field or other dictionary symbol. You can use this function to force the collection of note text. The operator can always create or edit a note manually by pressing **Ctrl+N** (on desktop CSEntry) or by clicking on the Notes button on mobile CSEntry.

If no arguments are supplied, the note for the current field is displayed for editing. The optional argument ***dictionary_symbol*** allows you to edit the note for a dictionary item, record, or level. If using the dictionary name, the case note is edited. Another optional string argument, ***operator_id***, allows you to edit the note for a particular operator. If no operator ID is supplied, then the note for the current operator will be edited.

Return Value

The function returns a string containing the note text. If there is no note, the length of the string will be 0.

Example

```
PROC COOKING

    if COOKING = 9 then
        COOKING_OTHER = editnote();
    endif;
```

See also: GetNote Function, PutNote Function

EndLevel Statement

Format

```
endlevel;
```

Description

The `endlevel` statement ends data entry for the current level of the current questionnaire. The effect of this statement depends on where it is used. If it is used in a field, roster, or form procedure, all remaining procedures within that level are skipped and control passes to the level **postproc**.

If the statement is executed in a level **preproc** or **postproc**, control passes to the **postproc** of the next-highest level. If it is used in the highest-level **postproc**, control passes to the form file's **postproc** (if there is one), and then data entry is terminated for the current case.

In system-controlled applications, CSPro will continue to add cases at the lowest level of a multiple-level dictionary until it is told to stop by `endlevel`. Therefore, the statement should be used in the **postproc** of the lowest level to end data entry at that level.

Example

```
if MORE_WOMEN = 0 then
    endlevel;
endif;
```

See also: EndGroup Statement

EndGroup Statement

Format

```
endgroup;
```

Description

The `endgroup` statement finishes data entry for the current group (roster or multiply-occurring form) in a data entry application. It can not be used in a batch application. If the statement is used in an item procedure, it causes an automatic skip to the **postproc** of the current group/record. If the statement is executed in the **preproc** of the group/record, the entire group/record is skipped and control passes to the group/record's **postproc**.

This function has superseded the endsect statement. Where endsect exists in an application, it will continue to work,

but users creating new applications should adopt the endgroup instruction.

Example

```
if KIDSBORN = 0 then
    endgroup;
endif;
```

See also: EndLevel Statement

Enter Statement

Format

```
enter form_file_name;
```

Description

The enter statement allows the use of a secondary form file to capture data in a secondary data file.

The ***form_file_name*** is the name of the secondary form file that you want to use. The secondary form file must be part of your data entry application.

The enter statement cannot be used inside a user-defined function.

Example

```
if V108 = 6 then
    enter OTHERS_FF;
endif;
```

GetCaptureType Function

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the getproperty function with the argument "CaptureType".*

Format

```
i = getcapturetype(field_name);
```

Description

The getcapturetype function returns the capture type currently associated with a field on a form. The ***field_name*** must be located on one of the application's forms.

Return Value

The function returns the capture type, which is one of the following values:

Code	Capture Type
0	Text Box
1	Radio Button
2	Check Box
3	Drop Down Box
4	Combo Box
5	Date Picker
6	Number Pad (desktop CSEntry only)
7	Barcode (Android only)
8	Slider (Android only)
9	Toggle Button

Example

```
PROC SEX

onfocus

    // change SEX from a radio button to a drop down box
    if getcapturetype(SEX) = 1 then
        setcapturetype(SEX,3);
    endif;
```

See also: SetCaptureType Function

GetCaseLabel Function

Format

```
s = getcaselabel(dictionary_name);
```

Description

The `getcaselabel` function returns the label for the case currently associated with the dictionary ***dictionary_name***. The dictionary can be either the main input dictionary of a data entry application or an external dictionary.

Return Value

The function returns the label if one has been set, or a blank string if there is no label for the case.

Example

```
PROC LAST_FIELD

    string end_query = maketext("Are you finished entering '%s'?", getcaselabel(CENSUS_DICT));

    if accept(end_query, "Yes", "No") <> 1 then
        reenter;
    endif;
```

See also: SetCaseLabel Function, Key Function

GetDeviceID Function

Format

```
s = getdeviceid();
```

Description

The getdeviceid function returns a string that can be used to uniquely identify a device. As programmed now, it returns a MAC address on Windows and an ANDROID_ID on Android, but this may change in the future. It is theoretically possible that this ID is not unique, but the chance of this happening without deliberate user invention is virtually impossible.

Return Value

The function returns a string with the unique ID.

Example

```
PROC USER_NAME

    if not loadcase(USER_PERMISSIONS_DICT,USER_NAME) then
        errmsg("The entered user name is not in the database of users.");
        reenter;

    elseif USER_ALLOWED_DEVICE <> getdeviceid() then
        errmsg("You, %s, do not have permission to use device
%s.",strip(USER_NAME),getdeviceid());
        reenter;

    endif;
```

See also: GetOperatorId Function, GetUserName Function, GetBluetoothName Function

GetImage Function

Format

```
s = getimage(item_name || valueset_name, value);
```

Description

The getimage function returns the filename of a value set image. Based on the provided item or value set, the image filename associated with the ***value*** will be returned. The ***value*** is either a numeric or string expression, based on the type of the item. If an ***item_name*** is used instead of a ***valueset_name***, then the function will search for the relevant image in the item's current value set.

Return Value

The function returns a string containing the image filename, or a blank string if there is no value set image for the provided value or if the provided value does not exist in the value set.

Example

```
PROC IMAGE_SELECTOR

    string image_filename = getimage(IMAGE_SELECTOR_VS1, IMAGE_SELECTOR);

    view(image_filename);
```

See also: GetLabel Function, SetValueSet Function, Image.Load Function

GetNote Function

Format

```
s = getnote([dictionary_symbol][,operator_id]);
```

Description

The `getnote` function returns a string containing the note for a field or other dictionary symbol. If no arguments are supplied, the note for the current field is returned. The optional argument ***dictionary_symbol*** allows you to query the note for a dictionary item, record, or level. If using the dictionary name, the case note is returned. Another optional string argument, ***operator_id***, allows you to query the note for a particular operator. If no operator ID is supplied, then the note for the current operator will be returned.

Return Value

The function returns a string containing the note text. If there is no note, the length of the string will be 0.

Example

```
string caseInformation = getcaselabel(CENSUS_DICT);

if length(getnote(CENSUS_DICT)) > 0 then
    // add the case note to the case label for a full description of the case
    caseInformation = caseInformation + " - " + getnote(CENSUS_DICT);
endif;
```

See also: PutNote Function, EditNote Function

GetOperatorId Function

Format

```
s = getoperatorid();
```

Description

The `getoperatorid` function returns the operator ID for the current operator. The operator ID may have been entered by the operator or passed as a parameter in the PFF file for the run.

Return Value

The function returns a string containing the operator ID assigned to or entered by the current operator. In batch mode it returns a blank string.

Example

```
LF_USER_ID = getoperatorid();

if not loadcase(LOGINS_DICT, LF_USER_ID) then
    errmsg("You do not have access to this system");
    stop(1);
endif;
```

See also: SetOperatorId Function, GetDeviceID Function, GetUserName Function, GetBluetoothName Function

GetOS Function

Format

```
i = getos([detailed_information]);
```

Description

The getos function returns a code that indicates what operating system is running the program. The codes are:

Code	Operating System
10	Windows
20	Android

It is also possible to get more detailed information about the operating system. If the optional argument ***detailed_information*** is a string variable, the string will be filled with the operating system name and the version number. Alternatively, if passed a one-dimensional string hashmap, the hashmap will be filled with several values:

Key	Value
"name"	Name (e.g., "Windows")
"version"	Version number (e.g., "10.0")
"build"	Build number (Windows only; e.g., "22000")

Windows version numbers do not necessarily match the official release numbers. For example, Windows 8.1 has a version number of 6.3.

Return Value

The function returns the operating system code and, optionally, more details about the operating system.

Example

```
string operating_system_name;
getos(operating_system_name);

errmsg("Code = %d, Text = %s", getos(), operating_system_name);

// the above code might display, for example:
// Code = 10, Text = Windows;6.1
// Code = 20, Text = Android;4.2.2
```

See also: GetOperatorId Function, GetUserName Function, GetBluetoothName Function

GetRecord Function

Format

```
s = getrecord(item_name_string);
```

Description

The getrecord function returns the name of the record that contains the item identified by the string expression ***item_name_string***.

Return Value

The function returns a string with the name of the record. If the item does not exist, the function returns a blank string.

Example

```
errmsg("SEX belongs to %s",getrecord("SEX"));                    // SEX belongs to POPULATION
errmsg("WATERSOURCE belongs to %s",getrecord("WATERSOURCE"));    // WATERSOURCE belongs to
HOUSING
```

See also: GetSymbol Function

GetUserName Function

Format

```
s = getusername();
```

Description

This `getusername` function returns the login of the current user. On Windows this is the name of the user currently logged in, whereas on Android this is the first account registered on the device.

Return Value

The function returns a string containing the user name.

Example

```
PROC USER_NAME

preproc

    USER_NAME = getusername();
    noinput;
```

See also: GetDeviceID Function, GetOperatorId Function, GetOS Function, GetBluetoothName Function

GPS Function

It is possible to take advantage of the functionality of a Global Positioning System (GPS) receiver when designing an application for use on either a laptop or tablet with a GPS receiver. Manipulating the GPS receiver is done from within the program's logic or by using one of the interactive GPS modes.

Basic Example

```
gps(open); // Android

gps(open, 3, 4800); // Windows (COM3, 4800 baud)

if gps(read, 15) then // attempt to read for up to 15 seconds
    errmsg("Latitude is %f, longitude is %f", gps(latitude), gps(longitude));

else
    errmsg("GPS signal could not be acquired");

endif;

gps(close);
```

Opening and Closing the GPS Receiver

```
b = gps(open);
b = gps(close);
```

Before using the device's GPS receiver, it is necessary to open a connection to the GPS unit. After making all necessary GPS readings, close the connection. The function returns 1 if successful, 0 otherwise. If using CSPro on a Windows laptop or tablet, it is necessary to specify the hardware settings of the GPS unit, specifically the number of the COM port and the baud rate (see the above example).

Obtaining the Last Successful GPS Reading

```
b = gps(readlast);
```

The **readlast** command obtains the last successful GPS reading, a reading that might be very old. If the GPS unit has been turned on for some time and the device is being used outdoors, it is likely that the reading is very fresh (recent), but if, for example, an enumerator walks inside a building to conduct an interview, the reading may be minutes or hours old, from the last time that the enumerator was outside. The function returns 1 if there was a successful previous reading, 0 otherwise.

Obtaining a New GPS Reading

```
b = gps(read);
b = gps(read, duration_in_seconds);
i = gps(read, duration_in_seconds, message);
```

The **read** command obtains a new GPS reading in a specified time period. The time period can be provided by supplying a numeric expression ***duration_in_seconds***. If no time period is specified, the program will pause for up to three seconds to obtain a reading. A reading time of up to ten minutes (600 seconds) may be specified. An optional third argument, the string expression ***message***, displays a message while the program attempts to obtain a GPS reading. The message box has a cancel button and if the user cancels the operation, the function returns -1. Otherwise the function returns 1 if a reading was successful, 0 otherwise. Unlike the **readlast** command, a successful function call with **read** guarantees a fresh GPS reading.

It is possible on Android devices to specify a desired level of accuracy for the reading. When specified, the function will continue to take GPS readings until a reading at or below the level of accuracy is achieved. The function will still return 1 if, though timing out, a successful reading at an accuracy level greater than specified was achieved during the allotted time period. This ensures that a GPS reading will be as accurate as possible, but with the assumption that any reading is better than no reading. Use a second numeric expression to specify the level of the accuracy.

```
b = gps(read, duration_in_seconds, accuracy);
i = gps(read, duration_in_seconds, message, accuracy);
```

Querying a Successful GPS Read

If the **readlast** or **read** commands returned successfully, the GPS system has valid values for latitude and longitude. Further attributes of the reading can be queried, though they are not guaranteed to be valid.

```
d = gps(latitude);
d = gps(longitude);
d = gps(altitude);
d = gps(satellites);
d = gps(accuracy);
d = gps(readtime);
```

Latitude and **longitude** return coordinates in degrees. **Altitude** returns the number of meters above sea level of the reading. **Satellites** returns the number of satellites used to calculate the values in the last reading. Generally, the greater number of satellites, the better the quality of the reading. **Accuracy** is a calculation of the precision of the last reading. On Windows devices, an accuracy value of 1 is the most accurate and 50 is the least accurate reading. On Android devices, the value signifies the accuracy of the reading, measured in meters. **Readtime** returns, in local time, the time of the last successful reading. If the queried value (other than latitude and longitude) is not available, or if the last GPS read was unsuccessful, the function will return `default`.

Calculating Distances

The `gps` function can also calculate great-circle distances between two sets of GPS coordinates. Great-circle calculations give a rough approximation of the distance between two points. The function returns the distance as measured in meters.

```
d = gps(distance, latitude1, longitude1, latitude2, longitude2);
```

For example:

```
numeric meters_to_census_bureau = gps(distance, 38.846261, -76.929445, gps(latitude), gps(longit
ude));
```

See also: Interactive GPS Modes

Interactive GPS Modes

The interactive GPS modes provide three easy-to-use approaches to taking GPS readings. Each mode displays a widget which lessens the need to design and implement a custom interface. Additionally, each interactive GPS mode automatically opens and closes the GPS unit, so there is no need to call `gps(open)` and `gps(close)`. All three interactive GPS modes are only supported on Android.

Satellite Mode

Displays a widget with a satellite image and a message. The satellite image can be tapped to take a GPS reading. The latitude, longitude, and accuracy are displayed in the bottom-left corner. The message is displayed across the top. Exit the widget by pressing the *CLOSE* button.

```
b = gps(readInteractive, message,
        baseMap := None,
        readDuration := duration_in_seconds,
        message := message);
```

The `gps` function must be passed the `readInteractive` command with the named argument `baseMap` set to `None`. The numeric expression ***duration_in_seconds*** may be used to set the read duration If it is not specified, the read duration will default to 15 seconds. A message can be provided as a string expression as the optional second argument, or via the ***message*** named argument.

Map Mode: Current Location

Displays a widget with a map, message, and current location indicator. Press the *CAPTURE CURRENT GPS LOCATION* to take a GPS reading specified by the current location indicator. The latitude, longitude, and accuracy are displayed in the bottom-left corner. In the bottom-right corner there is a center location button that will center the view on the current location. The message is displayed across the top. Exit the widget by pressing the *CLOSE* button.

```
b = gps(readInteractive⌠, message,⌡
        ⌠baseMap := base_map,⌡
        ⌠message := message⌡);
```

The gps function must be passed the readInteractive command. The optional named argument ***base_map*** can be set to one of four map types (Normal, Satellite, Hybrid, and Terrain), or a filename. The filename can specify a TPK or MBTile, so an offline map can be displayed. If no ***base_map*** is specified, then the base map will be calculated using defined rules. A message can be provided as a string expression as the optional second argument, or via the ***message*** named argument.

Map Mode: Select Location

Displays a widget with a map, message, current location indicator, and a reticle. The reticle allows a location to be selected. Press the *CAPTURE SELECTED LOCATION* to take a GPS reading indicated by the reticle. The latitude, longitude, and accuracy are displayed in the bottom-left corner. In the bottom-right corner there is a center location button that will center the view on the current location. The message is displayed across the top. Exit the widget by pressing the *CLOSE* button at the bottom.

```
b = gps(select⌠, message,⌡
        ⌠baseMap := base_map,⌡
        ⌠message := message⌡);
```

The gps function must be passed the select command. The optional named argument ***base_map*** can set to one of five map types (Normal, Satellite, Hybrid, Terrain, and None), or a filename. The filename can specify a TPK or MBTile, so an offline map can be displayed. If no ***base_map*** is specified, then the base map will be calculated using defined rules. A message can be provided as a string expression as the optional second argument, or via the ***message*** named argument.

Return Value

When using these interactive modes, the gps function returns a logical value of 1 (true) if successful and 0 (false) otherwise. If the function is run on Windows, the function returns default.

Examples

```
// Satellite Mode
gps(readInteractive,
    baseMap := None,
    readDuration := 60,
    message := "Click on the satellite to take a GPS reading for up to one minute:");

// Map Mode: Current Location
gps(readInteractive, "Take a GPS reading at your current location:");

// Map Mode: Select Location
gps(select, "Select your current location on the map:",
    baseMap := "Suitland.mbtiles");
```

See also: GPS Function, Base Map Specification

HideOcc Function

Format

```
b = hideocc(roster_name ‖ form_name⌠(occurrence_number)⌡);
```

Description

The hideocc function hides an occurrence of a roster or form from showing in the mobile CSEntry case tree. It will also hide the row from appearing on a roster on desktop CSEntry. If no numeric ***occurrence_number*** is supplied, then the

current occurrence for the roster or form will be hidden.

Return Value

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Example

```
PROC KEEP_PERSON

    if KEEP_PERSON = 0 then
        hideocc(PERSON000);
    endif;
```

See also: ShowOcc Function, SetOccLabel Function, Occurrence Labels

Highlighted Function

Format

```
b = highlighted(field_name);
```

Description

The `highlighted` function is used to determine whether a field, ***field_name***, is on the path (green) in system-controlled mode, or has been passed through (green or yellow) either directly or by skips in operator-controlled mode. This can be used to determine whether an item has been entered or skipped because of logic in system-controlled mode, or whether it is before or after the high water mark in operator-controlled mode.

Return Value

The function returns 1 (true) if the field has been passed (green or yellow) and 0 (false) if the field is yet to be entered (white).

Example

```
if highlighted(HOURS_WORKED) then
    errmsg("Hours Worked = %d",HOURS_WORKED);
else
    errmsg("Skipped or not entered yet!");
endif;
```

See also: VisualValue Function, Operator vs. System Controlled, Color of Fields in Data Entry User's Guide

InAdvance Function

Format

```
b = inadvance();
```

Description

The `inadvance` function indicates if an advance is in progress in a data entry application, allowing for the conditional execution of logic based on whether an enumerator has directly entered a field. Advances generally occur when the `advance` statement is executed in logic, when resuming from a partial save, or when operators move from one field to another using the case tree.

A common requirement is to execute some logic only when **not** advancing. For example, as with the `warning` function, you may want to display a "soft check" error message when the operator directly entered the field, not when resuming from a partial save. These two sets of logic are the same:

```
warning("...");

if not inadvance() then
    errmsg("...");
endif;
```

Return Value

The function returns a logical value of 1 (`true`) if an advance is being evaluted and 0 (`false`) otherwise. The function always returns 0 for batch and tabulation applications.

Example

```
PROC SURVEY_CONFIRMATION

    // always require the enumerator to confirm that the survey is complete
    if inadvance() then
        SURVEY_CONFIRMATION = notappl;
        reenter;
    endif;
```

See also: Advance Statement, Warning Function

IsPartial Function

Format

```
b = ispartial([dictionary_name]);
```

Description

The `ispartial` function determines, for data entry input dictionaries whether a case was opened from a partial case, and for external dictionaries where the case is currently marked as a partial save. By default the function looks at the case loaded by the main data entry input dictionary, but an optional argument, ***dictionary_name***, allows you to determine the partial save status of cases in external dictionaries.

Return Value

The function returns a logical value of 1 (true) if the case was opened from a partial case or if it has been partially saved during the data entry session and 0 (false) otherwise.

Example

```
if ispartial() then
    errmsg("Entering a partially saved case");
endif;
```

See also: SavePartial Function, OnStop Global Function, IsVerified Function

IsVerified Function

Format

```
b = isverified(dictionary_name);
```

Description

The `isverified` function determines whether a case has been verified by double-keying. The function, with argument ***dictionary_name***, can look at cases in external dictionaries or in batch edit input dictionaries.

Return Value

The function returns a logical value of 1 (true) if the case was verified and 0 (false) otherwise.

Example

```
PROC CASE_MENU

    if CASE_MENU = 1 then // verify case

        if isverified(SURVEY_DICT) then
            errmsg("The case has already been verified.");
            reenter;

        else
            // ...

        endif;

    endif;
```

See also: IsPartial Function

Move Statement

Format

```
move [to] field_name [advance || skip];
```

Description

The `move` statement allows movement to a field without regard to whether it is before or after the current field. This is particularly useful in an OnKey or userbar function because you may not know at what point the function will be called. The ***field_name*** is either the name of a field or is a string variable specifying the name of the field.

Movement to a field before the current field acts exactly like a `reenter` statement. The action of move to a field after the current field depends on the optional keywords `advance` or `skip`. If no keyword is specified, or if `skip` is coded, then forward movement acts exactly like a `skip` statement, skipping *past* fields. If `advance` is coded, then forward movement acts exactly like an `advance` statement, moving *over* fields.

Example

```
function numeric OnKey(numeric keystroke)

    if keystroke = 2070 then // Ctrl+F
        // move to the first person
        move to NAME(1) advance;

    elseif keystroke = 2076 then // Ctrl+L
        // move to the last person
        move to NAME(totocc(PERSON_REC000)) advance;

    else
        OnKey = keystroke;

    endif;

end;
```

See also: Advance Statement, Ask Statement, Reenter Statement, Skip Statement

NoInput Statement

Format

```
noinput;
```

Description

The `noinput` statement prevents input of a field during data entry. This command can be coded only in a `preproc` or `onfocus` procedures.

When the statement is executed in a `preproc`, control passes directly from the field's `preproc` to the field's `postproc`, executing the `onfocus` and `killfocus` procedures (if present) and performing the item range check, but not permitting input of the field.

When the statement is executed in an `onfocus`, control passes directly from the field's `onfocus` to the field's `postproc`, executing the `killfocus` procedure if present and performing the item range check, but not permitting input of the field. The field is on the data entry path even though entry is prevented.

The effect of the `noinput` statement is similar, but not identical, to that of a protected field. If a `noinput` statement is used, it is possible to back-tab to the field. It is not possible to back-tab to a field that is protected.

Example

```
PROC MARITAL_STATUS

preproc

    if AGE < 12 then
        MARITAL_STATUS = 1;
        noinput;
    endif;
```

See also: Advance Statement

OnChangeLanguage Global Function

Format:

```
function OnChangeLanguage();
```

Description:

The **onchangelanguage** global function allows you to execute certain code after the user has changed the CAPI language via the CSEntry Options menu. This function must be placed in the Global Procedure.

Returned value:

The return value of the function is ignored.

Example:

```
function OnChangeLanguage()
    setvaluesets(concat("_",getlanguage()));
end;
```

See also: Global Procedure, User Defined Functions

OnChar Global Function

Format

```
function OnChar(numeric key_value)
```

Description

The `OnChar` global function allows you to trap characters in order to perform special actions or to change the action of the character. It can also be used to disable or remap characters. This function must be placed in `PROC GLOBAL`.

If an `OnChar` function is defined, every character the operator types is sent to the function for processing. If the function returns a value, then the return value is processed by the field as the character. If a statement in the function causes movement to another field within the case, then the movement is executed. If no function is defined, then characters are unmodified.

The parameter ***key_value*** is a numeric code identifying what character was typed using the keyboard. Its value can be used within the function.

You can use the OnKey Character Map to determine the value of characters.

Calls to this function are ignored when executed on mobile devices.

Differences Between `OnKey` and `OnChar`

The `OnChar` function differs from the `OnKey` function. A keystroke contains information about the key pressed on the keyboard, regardless of what this keystroke eventually maps to. A character refers to the final representation of one or more keystrokes. This is important when using non-Latin languages that require multiple keystrokes to create one character. For example, creating the Chinese character '马' using a Pinyin input system requires two keystrokes: 'm' and 'a.' The code for this character is 39532. If typing such a keystroke, `OnKey` will be called several times before `OnChar` is called. If both `OnKey` and `OnChar` functions exist, `OnKey` will always be called before `OnChar` is called.

The `OnChar` function also returns values different from the `OnKey` function for some Latin keystrokes. For example, with Caps Lock off, if a keyer holds down Shift and types 'M,' `OnKey` will return 1077 (1000 for the shift, 77 for 'm'). `OnChar`, on the other hand, will return 77, the character code for 'M.' For a lowercase 'm,' `OnKey` returns 77 and `OnChar` returns 109, the character code for 'm.'

The `OnChar` function does not return any information about whether any of the Shift, Ctrl, or Alt keys were held down when the character was typed.

Return Value

The `OnChar` function must return an integer value. The value should be either the value of the character pressed (the

same as the value passed to the function), a substituted character value (remapping the character), or zero (0) to indicate that the character is to be ignored.

Example

```
function OnChar(numeric keystroke)

    if keystroke = 24110 then // 帮 (bang)
        move to HELP_FORM;
    else
        OnChar = keystroke;
    endif;

end;
```

See also: OnKey Global Function, OnKey Character Map, User-Defined Functions

OnKey Global Function

Format:

```
function OnKey(key-value);
```

Description:

The **onkey** global function allows you to trap keystrokes in order to perform special actions or to change the action of the key. It also can be used to disable or remap keys. This function must be placed in the Global Procedure.

If an onkey global function is coded, every keystroke the operator types is sent to the onkey function for processing. If the onkey function returns a value, then the return value is processed by the field as the keystroke. If a statement in the onkey function causes movement to another field within the case, then the movement is executed. If no onkey function is coded, then keystrokes are unmodified.

The **key value** is a number code identifying what key was pressed on the keyboard. Its value can be used within the function. See detailed description below.

You can use the OnKey Character Map to determine the value of characters.

Calls to this function are ignored when executed on mobile devices.

Returned value:

Like any global function, the onkey global function returns an integer value. The value should be either the value of the key pressed (the same as the value passed to the function), a substituted key value (remapping the key), or zero (0) to indicate that the key is to be ignored.

Example:

```
function OnKey(numeric x)
  if x in 114:116 then { F3, F4, F5 }
    { don't allow these keys to work, eat the key }
    OnKey = 0;
  elseif x = 2069 then { Ctrl+E go to END_FIELD }
    move to END_FIELD;
  else
    OnKey = x; { return rest of keys }
  endif;
end;
```

Keyboard Key Values

The key values passed into and out of the **onkey** function are given below.

Single key values are given below. When Shift, Ctrl, and/or Alt are held down while pressing another key, the following values are added to the code of the other key.

Keys	Add
Shift	1000
Ctrl	2000
Shift+Ctrl	3000
Alt	4000
Alt+Shift	5000
Alt+Ctrl	6000
Alt+Shift+Ctrl	7000

For example, if Shift, Ctrl, and/or Alt are held down when A is pressed

Keys	Code
A	65
Shift+A	1065
Ctrl+A	2065
Shift+Ctrl+A	3065
Alt+A	4065
Alt+Shift+A	5065
Alt+Ctrl+A	6065
Alt+Shift+Ctrl+A	7065

Numbers

For the number keys across the top of the keyboard, or on the numeric keypad when the "NumLock" button is depressed, the following keyboard key value will be returned:

Key	Code
0	48
1	49

2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Letters

For the letters a-z and A-Z, the following keyboard key values will be returned. Please note that the status of the CapsLock key does not effect the code. However, if the Ctrl, Shift, and/or Alt keys are being held down, they will change the code returned. See above.

Key	Code
a	65
b	66
c	67
d	68
e	69
f	70
g	71
h	72
i	73
j	74
k	75
l	76
m	77

n	78
o	79
p	80
q	81
r	82
s	83
t	84
u	85
v	86
w	87
x	88
y	89
z	90

Function Keys

Key	Code
F1	112
F2	113
F3	114
F4	115
F5	116
F6	117
F7	118
F8	119

F9	120
F10	121
F11	122
F12	123

Non-numeric Numpad Keys:

Key	Code
Num Lock	144
/	111 (with or without num lock)
*	106 (with or without num lock)
-	109 (with or without num lock)
+	107 (with or without num lock)
.	110 (with num lock—46 without)

Miscellaneous Keys:

Key	Code
SysReq	no code returned
Bksp	8
Tab	9
Enter	13 (with or without num lock)
Break	19
Caps	20
Escape	27

Space	32
Page Up	33
Page Down	34
End	35
Home	36
Left Arrow	37
Up Arrow	38
Right Arrow	39
Down Arrow	40
Insert	45
Delete	46
Wnd	91 (flying window)
Scroll Lock	145
;	186
=	187
,	188 (comma)
-	189
.	190 (period)
/	191
`	192 (accent)
[	219
]	221
\	220

```
'        222
```

Control Keys:

Key	Code
Shift	1016
Ctrl	2017
Alt	4018

See also: Global Procedure, User Defined Functions, OnKey Character Map, OnKey Character Map

OnStop Global Function

Format

```
function OnStop();
```

Description

`OnStop` is a special global function. It has no return value and must be placed in the `PROC GLOBAL` section just like any other user-defined functions.

When defined, it provides control over what happens when the data entry operator tries to stop data entry using the ESC key, the ■ Stop button, pressing Ctrl+S, or attempting to exit data entry. When any of the above events occur, the `OnStop` function is executed and no stop dialog (discard, save, cancel) occurs.

If an `OnStop` function has been coded in a data entry application, then when resuming a partial case, no resume dialog ("Do you want to go to last...") occurs. If special actions are required when entering a partial case, check whether a partial case has been entered using the `ispartial` function and program the appropriate action. You can retrieve the name and occurrence number of the last field entered (on a one-level application) by calling `getsymbol(savepartial)`.

The `OnStop` function is not executed when the `stop` function is executed.

The `OnStop` function can be used to keep the operator from stopping data entry (see Example 1 below) or to allow stopping only under certain conditions (see Example 2 below).

Example 1

```
PROC GLOBAL

function OnStop()
    reenter;
end;
```

Example 2

```
PROC GLOBAL

function OnStop()
    if getsymbol() in "FIRST_NAME", "LAST_NAME" then
        reenter;
    else
        savepartial();
        stop(1);
    endif;
end;
```

See also: User-Defined Functions, Function Statement, Stop Function, SavePartial Function, IsPartial Function, EndLevel Statement

OnSystemMessage Global Function

Format

```
function numeric OnSystemMessage([numeric message_number][, numeric message_type][, string messa
ge_text]);
```

Description

`OnSystemMessage` is a special global function. It is a function that gets called when a system message is issued. This is similar to the `OnKey` or `OnStop` functions, which get called automatically by the running CSPro application, not because of a call to the function in your logic. A system message is a message such as "Invalid subscript," not a user message that comes from an `errmsg` statement.

The function must return a numeric value and you can provide from one to three parameters. If one numeric parameter is provided, then it will receive the message number. If two numeric parameters are provided, the second numeric parameter will get the message type (1 = error, 2 = warning). If a string parameter is provided, then it will receive the message text.

Return Value

Return 0 (false) to suppress the message. Returning anything other than 0 means that the message will be issued (displayed to the enumerator in a data entry application or written to the listing file in a batch application).

Example 1

```
function OnSystemMessage(numeric message_number, numeric message_type)

    // suppress all warning messages
    recode message_type -> OnSystemMessage;
           1            -> true;
           2            -> false;
    endrecode;

end;
```

Example 2

```
function OnSystemMessage(numeric message_number, string message_text)

    // modify the displayed message for invalid subscript errors
    if message_number = 1008 then
        errmsg("Subscript Error (%d). Most likely cause is no children in household.
%s", message_number, message_text);

        // suppress the system message
        exit false;

    endif;

    // issue the message in all other cases
    exit true;

end;
```

See also: User-Defined Functions, Function Statement

Prompt Function

Format

```
s = prompt(message⌠, initial_value⌡⌠, prompt_type⌡);
```

Description

The `prompt` function displays a dialog box showing the string expression ***message*** and provides a text box for an operator to enter a string value. An optional string expression ***initial_value*** determines the starting value displayed to the operator. Another optional argument, ***prompt_type***, provides a way to customize the kind of entry permitted. Types include:

- **Password**: The entered characters will be masked with asterisks.
- **Uppercase**: Alphabetical characters will automatically be converted to their uppercase equivalents.
- **Numeric**: Only numeric characters can be entered.
- **Multiline**: The text box will support multiple lines of data entry and the Enter key on the keyboard will insert carriage returns, which will be converted to `"\n"` in the returned value.

The **password** and **multiline** options are not compatible so when using one of these options, the other option cannot be used.

Return Value

The function returns a string value containing the entered value. If the operator canceled without entering a string, the function returns a blank string (even if an initial value was provided).

Example 1

```
string survey_password = prompt("Enter your password:", password);
```

Example 2

```
if ROOF_TYPE = 9 then // other (specify)
    ROOF_TYPE_OTHER = prompt("Specify the kind of roof on the dwelling
unit:", ROOF_TYPE_OTHER, multiline);
endif;
```

See also: Accept Function

Protect Function

Format

```
i = protect(symbol, protect_property);
```

Description

The `protect` function modifies the protected property of the field ***symbol***. If ***symbol*** is a block, group, form, or dictionary, the function will try to apply the protected property to all fields belonging to that symbol. The numeric expression ***protect_property*** turns the protected property off (if zero) or on (if nonzero).

Return Value

The function returns the number of items whose protected property was successfully changed.

Example

```
PROC RELATIONSHIP

onfocus

    // automatically set the first person to be the head of the household
    // and protect the field so that it is not shown to the enumerator
    if curocc() = 1 then
        RELATIONSHIP = 1;
        protect(RELATIONSHIP, true);
    else
        protect(RELATIONSHIP, false);
    endif;
```

See also: SetProperty Function

PutNote Function

Format

```
b = putnote(note_text⌠,dictionary_symbol⌡⌠,operator_id⌡);
```

Description

The `putnote` function sets the note for a field or other dictionary symbol, replacing the note with the text found in the string expression ***note_text***. If the note text is an empty string, the note will be removed. The optional argument ***dictionary_symbol*** allows you to set the note for a dictionary item, record, or level. If using the dictionary name, the case note is set. Another optional string argument, ***operator_id***, allows you to set the note for a particular operator. If no operator ID is supplied, then the note for the current operator will be set.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
PROC STATUS

    // set the case note to a description of the interview status
    putnote("Interview Status: " + getlabel(STATUS,STATUS),CENSUS_DICT);
```

See also: GetNote Function, EditNote Function

RandomizeVS Function

***Feature Upgrade**: Starting with CSPro 7.5, you are encouraged to use the `valueset.randomize` function.*

Format

```
i = randomizevs(dictionary_symbol[, exclude(exclude_code1, ..., exclude_codeN)]);
```

Description

The `randomizevs` function scrambles the order of values in a value set. The function is useful when using capture types in a data entry application. Some survey applications choose to display the possible response categories in a randomized way so as to minimize an enumerator or respondent's selection bias.

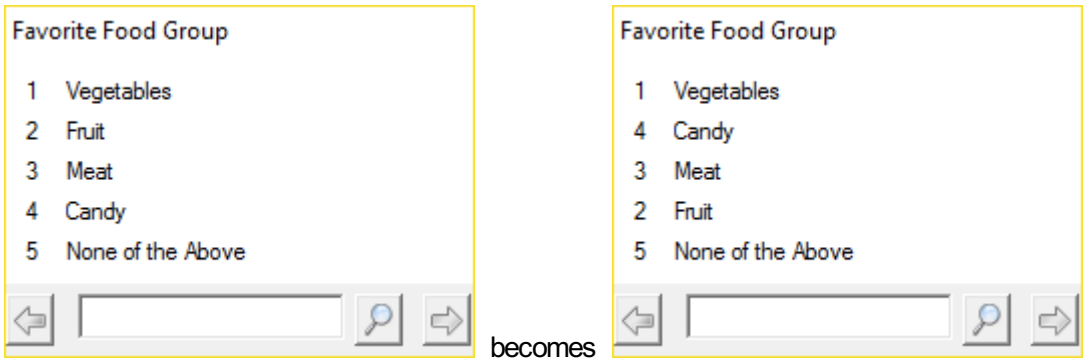
The ***dictionary_symbol*** may be the name of a value set, item, group, form, or an entire dictionary. If randomizing the value set for an item, an optional exclusion list, ***exclude_code1***, ..., ***exclude_codeN***, allows you to prevent certain values from being given a random order. This is useful for variables like *Don't Know*, which, after the randomization, would remain at the bottom of the list of values.

Return Value

The function returns the number of items for which the value sets were successfully randomized.

Example

```
randomizevs(FAVORITE_FOOD_GROUP, exclude(5));
```



See also: Random Function, RandomIn Function, Seed Function, ValueSet.Randomize Function

Reenter Statement

Format

```
reenter [field_name];
```

Description

The `reenter` statement is used to force the entry operator to reenter the contents of the current field or of a field that was entered earlier. The optional ***field_name*** is either the name of a field or is a string variable specifying the name of the field. If the ***field_name*** is not specified, the current field is reentered. The field to be reentered must be earlier on the data path than the current variable. If it is not, an error message will be displayed during data entry. If you do not know whether the field is earlier in the data path, use the `move` statement.

When a reenter statement is executed, the preproc of the reentered field will not be executed. The postproc of the field will be executed normally after the field has been reentered. If the field is on a different form than the current field, that form will be displayed automatically.

Example 1

```
PROC RELATIONSHIP

    if curocc() = 1 and RELATIONSHIP <> 1 then
        errmsg("The head of household must be entered on the first row.");
        reenter;
    endif;
```

Example 2

```
PROC DOB_YEAR

    if DOB_YEAR = CensusYear and DOB_MONTH > CensusMonth then
        errmsg("The date of birth cannot be after the census date. Reenter the month and
year.");
        reenter DOB_MONTH;
    endif;
```

See also: Advance Statement, Ask Statement, Move Statement, Skip Statement

SavePartial Function

Format

```
b = savepartial([clear]);
```

Description

The savepartial function saves the current case as a partially added, modified or verified case. It is useful in a large data entry application to perform intermediate backups. It can also be used to automatically perform a partial save when the keyer stops the case before completing it. You can also have CSEntry automatically partially save cases.

The function can be coded only in the preproc or postproc of a field. The function cannot be used before all ID values for the case have been entered. You can get the field name where the last partial save occurred by coding getsymbol(savepartial).

An optional argument, clear, results in the removal of visual values before the data is saved. This behavior is the same as would happen if the case was completed in a system controlled application. Generally you will want to maintain skipped-over fields in case an interviewer decides to make changes to the case at a later point, so it is rare that you would use this argument.

Return Value

The function returns a logical value of 1 (true) if the case was successfully saved as a partial case and 0 (false) otherwise.

Example

```
PROC LINE_NUMBER

preproc

    savepartial();
```

See also: IsPartial Function, Stop Function

SelCase Function

Format:

```
b = selcase([heading,]ext-dict-name,alphanumeric-expression[,offset])
     [include(dictionary-item])] [where logical-expression];
```

[ ] indicates that this part is optional.

Description:

The **selcase** function allows a data entry operator to select and load a case from an external file. This function can only be used in data entry applications. It searches the index of the external file named by "*ext-dict-name*" for all cases whose keys match the criterion specified by "*alphanumeric-expression*." If two or more matching keys are found, they will be presented to the entry operator in a display box. Using a highlight bar, the operator can select one of the keys. The case identified by that key is then read into memory.

The "*offset*" tells CSPro the number of characters, beginning with the first character of the ID items for the external file, that should be suppressed if multiple matches are found.

• *alphanumeric-expression*: can be a literal or a CSPro expression. The matching is case sensitive. If an empty string is passed, all cases in the external file are returned.

• *include*: tells CSPro to list additional items from the specified dictionary in the display box.

• *where*: applies the logical expression to all cases returned by the **selcase** statement. The resulting display box will only show cases in which the logical expression evaluated to true (returning a nonzero value).

It is possible, by specifying dictionary access parameters after the dictionary name, to limit the cases that the `selcase` function processes.

Return value:

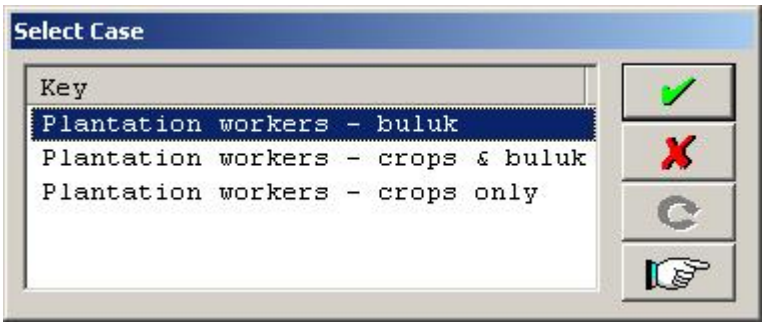
The function returns a logical value of **true** (1) if a case is found or selected by the entry operator and **false** (0) otherwise.

Example 1:

```
OK = selcase(LOOKUP,concat(PROV, DIST));
```

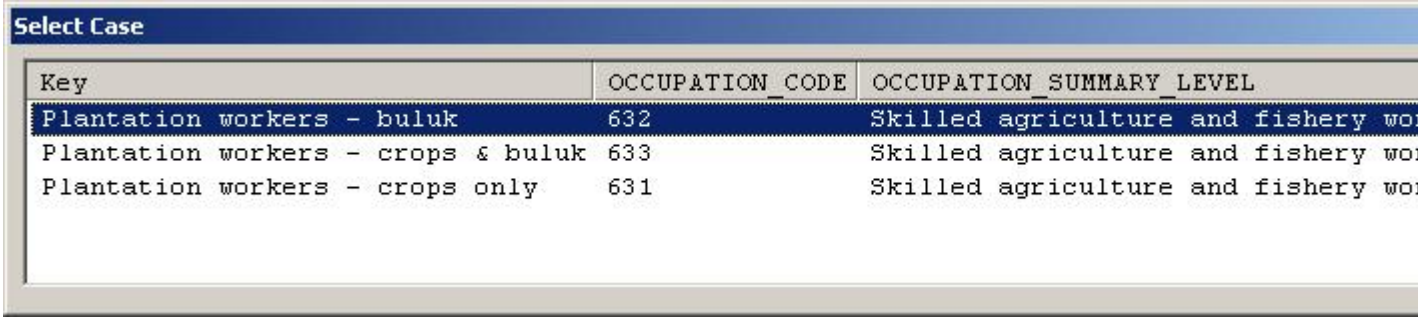
Example 2:

```
OK = selcase(OCCUPATION_DICT, "Plantation");

will return cases whose key begins with "Plantation."
```



Example 3:

```
OK = selcase(OCCUPATION_DICT, "Plantation")
     include(OCCUPATION_CODE, OCCUPATION_SUMMARY_LEVEL);
```



Example 4:

```
OK = selcase(OCCUPATION_DICT, "")
     include(OCCUPATION_CODE, OCCUPATION_SUMMARY_LEVEL)
     where OCCUPATION_CODE >= 631 and OCCUPATION_CODE <= 633;

will result in the same screen as appeared in Example 3.
```

Set Attributes Statement

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use these functions as they may soon be removed from CSPro. To replicate the behavior of the functions, you can use the `setproperty` function with one of the following arguments: `"CaptureType"`, `"Protected"`, or `"UseEnterKey"`.*

Format 1:

```
set attributes (field-1[, field-2, ..., field-N])
    display | visible | autoskip | return | protect | hidden | native;
```

Format 2:

```
set attributes(field-1[, field-2, ..., field-N])
  assisted on|off [(question, responses)];
```

[ ] indicates that this part is optional
| indicates that one of the attributes may be selected

Description of Format 1:

This **set attributes** statement switches the values of various field properties. Field properties can be set statically, via the field properties dialog box, or dynamically at run time via the set attributes command. A dynamically set field property will override any statically set property. Field properties can be set dynamically anywhere in the program **except** in the `PROC GLOBAL` section.

One or more dictionary items can be named in the field list. If the dictionary name is used, all the fields in the dictionary are affected. If a form name is used, all the fields on the form are affected.

In Format 1, only one attribute setting can be used in any single set attributes statement. The options are as follows:

`visible`	If a field is hidden, its value will now be visible; if it was already visible, the setting has no effect
`autoskip`	This is equivalent to leaving the statically-set field property "Use Enter Key" unchecked. If this option is used, the cursor automatically advances to the next field, after the maximum number of characters have been entered. This option will override any statically-set field property settings.
`return`	This is equivalent to checking the statically-set field property "Use Enter Key." If this option is used, the operator must press the <Enter> key to advance from the listed field(s). This option will override any statically-set field property settings.
`protect`	This is identical to the statically-set field property "protected." If a field is set to

	'protect', the operator will not be able to enter it. If the field was already statically set to "protected," the setting has no effect.
hidden	If a field is visible, its value will now be hidden from view; if it was already hidden, the setting has no effect.
native	Regardless of what settings have been made dynamically in the program, if a field is set to native, all field settings will revert to their initial, statically-set properties.

Example of Format 1:

```
set attributes (total_HH_income) protect;
```

Description of Format 2:

The set attributes statement with the assisted keyword switches on or off a popup responses box during data entry. The values in the responses box come from the first value set in the data dictionary for that field. The user can either select a response or type a response. This behavior is true in any CSPro data entry application.

One or more dictionary items can be named in the field list. If the dictionary name is used, all the fields in the dictionary are affected. If a form name is used, all the fields on the form are affected.

By default, the responses are taken from the first value set of the item. You can modify the values and responses using the function setvalueset.

Description of Format 2 in CAPI applications

This statement has further meaning in CAPI data entry applications. Note that when you create a CAPI data entry application, the question text for each item is automatically shown during data entry, but NOT the responses box. There are two ways to make the responses box appear (or disappear):

1. Use this statement in the application's logic. For example:

```
set attributes(MYDICT) assisted on;
Set attributes(MYDICT) assisted off;
set attributes(REL, SEX, EDUC) assisted on;
```

2. Have the operator turn the response box on and off:

   To show or hide responses for **this field**, press **Ctrl+C** or from the **Options** menu, select **Show Responses (This Field)**.

   To show or hide responses for **all fields**, press **Ctrl+K** or from the **Options** menu, select **Show Responses (All Fields).**

   The operator can also move the response box around on the screen using the mouse.

Use the following forms of this command to get the desired behavior in your program logic for CAPI applications:

```
// show both questions and responses
set attributes(MYDICT) assisted on;
set attributes(MYDICT) assisted on (question, responses);

// show question text but not responses
set attributes(MYDICT) assisted on (question);
< no set attributes command >

// show responses but not question text
set attributes(MYDICT) assisted on (responses);
```

See also: Change Field Properties, Change Data Entry Options, Introduction to CAPI

Set Behavior CanEnter Statement

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setproperty` function with the argument `"CanEnterNotAppl"` or `"CanEnterOutOfRange"`.*

Format 1:

```
set behavior([field-1, ..., field-N]) canenter(notappl|outofrange) on
(confirm|noconfirm);
```

Format 2:

```
set behavior([field-2, ..., field-N]) canenter(notappl|outofrange) off;
```

Description:

The **set behavior canenter** statement allows the entry of blanks (notappl) for numeric data items during data entry or to bypass the system 'Out of Range' message during data entry. You may wish to enter blanks when answers are missing from the form. You may wish to bypass the system 'Out of Range' message in order to code your own message.

If no arguments are specified, then the set behavior statement affects all data items from the point where it is executed onward. To limit its scope, it must be turned on and off at appropriate times. It is possible, however, to specify the name of a field, group, record, form, or dictionary to narrow the scope of the behavior to only fields that fall within the specified object.

In operator-controlled applications, notappl defined as a value in the value set for the item usually allows blank to be accepted. In system-controlled applications the set behavior function must be used to allow blanks even if notappl is in the value set.

The keywords **confirm** or **noconfirm** must be coded when **on** is used. Confirm means that a message box is displayed asking if it OK to enter this value. Noconfirm means that no message box is displayed.

Example 1:

```
set behavior() canenter(notappl) on (noconfirm);
```

Example 2:

```
set behavior(AGE,AGE_FIRST_MARRIAGE) canenter(outofrange) on (noconfirm);
set behavior(RELIGION) canenter(notappl) on (noconfirm);
```

See also: Special Values, Invalueset

SetCapturePos Function

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setproperty` function with the argument `"CapturePosX"` or `"CapturePosY"`.*

Format

```
i = setcapturepos(symbol, x_coord, y_coord);
```

Description

The `setcapturepos` function modifies the capture position for the capture types associated with a field on a form. The ***symbol_name*** may refer to a field, group, form, or an entire dictionary. The two numeric expressions ***x_coord*** and ***y_coord*** refer to the x (horizontal) and y (vertical) positions of the top-left corner of the capture type window. This position is relative to the form window, not the whole CSEntry window, with (0,0) referring to the top-left corner of the form. If the position given for a field is greater than the size of the form, CSEntry will ignore the argument when displaying the capture type window. This function is useful for CAPI applications in which a part of the screen is left blank specifically for capture type windows. Calls to this function are ignored when executed on mobile devices.

Return Value

The function returns the number of items whose capture positions were successfully changed.

Example

```
setcapturepos(CAPI_DICT, 700, 20); // draw all windows on the right side of the form
```

See also: SetCaptureType Function, Change Form Properties

SetCaptureType Function

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setproperty` function with the argument `"CaptureType"` or `"CaptureDateFormat"`.*

Format

```
i = setcapturetype(symbol_name, capture_type[, date_format]);
```

Description

The `setcapturetype` function modifies the capture type currently associated with a field on a form. The ***symbol_name*** may refer to a field, group, form, or an entire dictionary. The numeric expression ***capture_type*** refers to a capture type code, listed in the table below. If specifying a date, the format can be specified with the optional string expression ***date_format***.

Code	Capture Type
0	Text Box
1	Radio Button
2	Check Box
3	Drop Down Box
4	Combo Box
5	Date Picker
6	Number Pad (desktop CSEntry only)
7	Barcode (Android only)
8	Slider (Android only)
9	Toggle Button

Return Value

The function returns the number of items that were successfully changed to the specified capture type. An item's capture type can be successfully changed if the current value set associated with the item supports the requested capture type. For instance, if an item's value set contains ranges, the capture type cannot be changed to a radio button. The success of changing a field to a date picker does not depend on the date format, so if the date format is not valid for the field the capture type will be changed to the default date format for the item.

Example

```
setcapturetype(DATE_OF_BIRTH, 5, "YYYYMMDD");

errmsg("%d fields changed to radio buttons on the housing form",setcapturetype(HOUSING_FORM,1));
```

See also: GetCaptureType Function, SetCapturePos Function

SetCaseLabel Function

Format

```
b = setcaselabel(dictionary_name, case_label);
```

Description

The `setcaselabel` function sets the label to ***case_label*** for the case currently associated with the dictionary ***dictionary_name***. The dictionary can be either the main input dictionary of a data entry application or an external dictionary. By default, CSEntry will show the case label instead of the ID items when listing the cases in a data source. (A menu option allows CSEntry to show the case IDs instead.)

Return Value

The function returns 1 (true) if successful or 0 (false) if the function failed.

Example

```
PROC HEAD_HH_NAME

    setcaselabel(CENSUS_DICT, maketext("Household %d headed by
%s", HH_NUMBER, strip(HEAD_HH_NAME)));
```

See also: GetCaseLabel Function

Set ErrMsg Statement

Format

```
set errmsg(message_style[,close_message][,close_keystroke]);
```

Description

CSPro data entry applications have two operating modes: operator- and system-controlled modes. Each mode has a different style error message box. System-controlled mode uses a standard Windows dialog box whereas operator-controlled mode uses a customized yellow box. The `set errmsg` statement allows the user to choose which kind of box to use. The first argument, ***message_style***, is one of the following:

- **default**: Reset the box to the default style for the data entry mode.
- **operator**: Use the operator-controlled box.
- **system**: Use the system-controlled box.

For operator-controlled message boxes, the optional string expression ***close_message*** allows you to modify the text that appears beneath the error message, which by default is "Press F8 to clear." Another optional argument, the numeric expression ***close_keystroke***, modifies the keystroke required to close the box.

Calls to this function are ignored when executed on mobile devices.

Examples

```
set errmsg(system);     // use the system-controlled style
set errmsg(operator);   // use the operator-controlled style
set errmsg(default);    // use the default style for the data entry mode

// defines the error message and defaults to F8 to close the box
set errmsg(operator,"Appuyez sur F8 pour fermer");

// 67 (C) is the key that will close the box
set errmsg(operator,"Appuyez sur C pour fermer",67);

// 67 (C) is the key that will close the box
// but the default error message text is used
set errmsg(operator,67);
```

See also: SetFont Function

SetFont Function

Setting Format

```
b = setfont(ErrMsg || ValueSets || NumberPad || UserBar || Notes || All, font_name, font_size⌈, bold,
 italics⌋);
```

Resetting Format

```
b = setfont(ErrMsg || ValueSets || NumberPad UserBar || Notes || All, default);
```

Description

In a data entry application, the `setfont` function allows you to modify the font that CSPro uses to display text in:

- ***ErrMsg***: The font that appears in boxes generated by the `errmsg` function.
- ***ValueSets***: The font that appears in capture type popup windows (other than the number pad).
- ***NumberPad***: The font that is used by the number pad.
- ***UserBar***: The font that appears in buttons and text strings on the `userbar`.
- ***Notes***: The font that appears when a user edits a field note.
- ***All***: The font is changed for all of the four above entities.

The font is changed to the font indicated by the string expression ***font_name*** and the numeric expression ***font_size***, or if the resetting format is used, the font is restored to CSPro's ***default*** font selection. The user must ensure that the desired font is installed on the machine that will run the data entry application. Optional ***bold*** and ***italics*** markers may be indicated. Calls to this function are ignored when executed on mobile devices.

Return Value

The function returns 1 (true) if the font(s) were changed successfully, and 0 (false) otherwise.

Example

```
function majorError(string message)

    setfont(ErrMsg, "Arial", 24, bold, italics);
    errmsg("%s", message);

end;

function minorError(string message)

    setfont(ErrMsg, "Arial", 12, bold);
    errmsg("%s", message);

end;

function minorWarning(string message)

    setfont(ErrMsg, "Arial", 12, italics);
    errmsg("%s", message);

end;


PROC AGE

    if AGE > 95 then
        minorWarning(maketext("Age (%d) is over 95, set to 95", AGE));
        AGE = 95;

    elseif not AGE in 12:95 then
        minorError(maketext("Age (%d) is invalid for this survey, reenter", AGE));
        reenter;

    endif;
```

See also: Set ErrMsg Statement, SetValueSets Function

SetOperatorId Function

Format

```
b = setoperatorid(operator_id);
```

Description

The `setoperatorid` function sets the operator ID to the string expression ***operator_id*** for the current data entry session. The operator ID is generally entered by the operator or passed as a parameter in the PFF file, but for some programs, such as menu systems, it may be useful to set the operator ID in logic.

Return Value

The function returns 1 (true) if successful or 0 (false) if the function failed or was executed in batch mode.

Example

```
PROC USER_ID

    LF_USER_ID = USER_ID;

    if not loadcase(LOGINS_DICT, LF_USER_ID) then
        errmsg("Invalid login.");
        reenter;
    endif;

    setoperatorid(strip(LF_USER_NAME));
```

See also: GetOperatorId Function

Show Function

Format

```
i = show([heading,]group_name,item_list[,where condition][,title(text_list)]);
```

Description

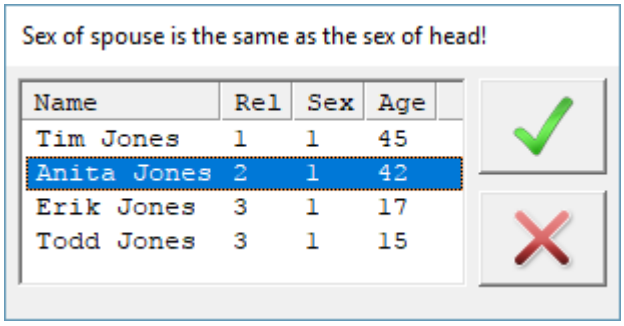
The show function displays items from a roster in the form of a menu that looks like a roster. The ***group_name*** specifies which group contains the items to be displayed, one or more of which are specified in the ***item_list***. This function is similar to the show and accept functions, and is useful as a menu or simply as a way to show roster values in another part of the questionnaire. An optional string expression, ***heading***, defines the title of the grid. An optional where clause, with a specified ***condition***, allows you to only specify that only some occurrences of the roster should be displayed. If you want to override the column headings, you can specify them with string expressions in ***text_title***.

Return Value

The function returns the number of the item selected: 1 for the first item, 2 for the second item, etc. This is the number of the item on the display, not in the roster. The value 0 is returned if the escape key (or back button) is pressed and none of the options is chosen.

Example 1

```
if RELATIONSHIP = 2 and SEX = SEX(ptrHead) then
    show("Sex of spouse is the same as the sex of head!",PERSON_REC,NAME,RELATIONSHIP,SEX,AGE,
        title("Name","Rel","Sex","Age"));
endif;
```



Example 2

```
if ctrHead > 1 then // more than 1 head in the household
    errmsg("More than 1 head of household (count=%d)",ctrHead);
    numeric headToDelete = show(PERSON_REC,NAME,RELATIONSHIP,SEX,AGE,WHERE RELATIONSHIP = 1);
    // ...
endif;
```

See also: Accept Function, ShowArray Function, List.Show Function

ShowArray Function

Format

```
i = showarray(heading, array_name, row_count, column_count, title(text_titles));
```

Description

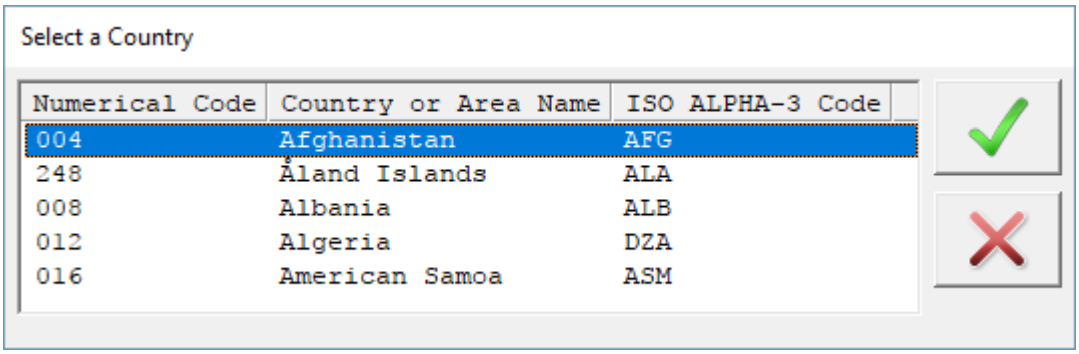
The showarray function, similar to the show and accept functions, displays items from an array in a grid. An optional string expression, ***heading***, defines the title of the grid, which contains the items in ***array_name***. The function is useful as a menu or simply as a way to show values that are relevant to data collection. If the optional numeric expressions ***row_count*** and ***column_count*** are not specified, then the function will parse the array, determining the size of the menu based on where the first blank alphanumeric element is found. If you want to override the column headings, you can specify them in with string expressions in ***text_titles***.

Return Value

The function returns the number of the item selected: 1 for the first item, 2 for the second item, etc. The value 0 is returned if the escape key (or back button) is pressed and none of the options is chosen.

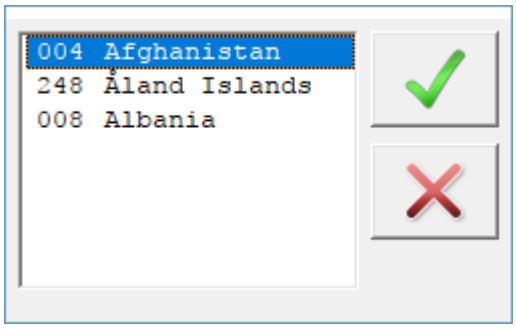
Example 1

```
PROC GLOBAL

array string unitedNationsCountryCodes(5,3) =
    "004",  "Afghanistan",      "AFG",
    "248",  "Åland Islands",    "ALA",
    "008",  "Albania",          "ALB",
    "012",  "Algeria",          "DZA",
    "016",  "American Samoa",   "ASM",
    // ...
    ;

PROC EXAMPLE

numeric countrySelection = showarray("Select a Country", unitedNationsCountryCodes,
    title("Numerical Code", "Country or Area Name", "ISO ALPHA-3 Code"));
```



Example 2

```
numeric countrySelection = showarray(unitedNationsCountryCodes, 3, 2);
```



See also: Accept Function, Show Function, List.Show Function

ShowOcc Function

Format

```
b = showocc(roster_name || form_name[(occurrence_number)][,show_condition]);
```

Description

The showocc function shows an occurrence of a roster or form in the mobile CSEntry case tree. It will also show the row on a roster on desktop CSEntry. If no numeric ***occurrence_number*** is supplied, then the current occurrence for the roster or form will be shown.

By default, all occurrences of a roster and form are shown by CSEntry, but occurrences may be hidden with the hideocc function. Alternatively, an optional numeric argument ***show_condition***, allows the showocc function to conditionally hide occurrences, based on whether the condition is true. The occurrence will be hidden if ***show_condition*** is 0; the occurrence will be shown for any other value.

Return Value

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Example

```
PROC MYSURVEY_QUEST

preproc

    // fill in and potentially hide occurrence labels for the household roster
    do numeric ctr = 1 while ctr <= totocc(PERSON_ROSTER)
        showocc(PERSON_ROSTER(ctr),KEEP_PERSON(ctr));
        setocclabel(PERSON_ROSTER(ctr),strip(NAME(ctr)));
    enddo;
```

See also: HideOcc Function, SetOccLabel Function, Occurrence Labels

Skip Statement

Formats

```
skip [to] field_name;

skip [to] next [multiply_occurring_field_name];

skip;
```

Description

The skip statement skips to the a field located after the current field. The argument ***field_name*** is either the name of a field or is a string variable specifying the name of the field. If the field has multiple occurrences, and if skipping from a field not on the same repeating form or roster, the target occurrence number must be specified to skip to the correct occurrence.

The next keyword skips to the next occurrence of ***multiply_occurring_field_name***. If the target field is on the same repeating form or roster as the current field, control will move to the next occurrence of the target field. If not, control will move to the first occurrence of the target field. Occurrence numbers cannot be used with the next keyword.

When using the next keyword without the optional target field, control passes to the next occurrence of the current repeating form or roster, with the target field as the first field in that form or roster. This is a useful way to skip to the beginning of the next occurrence.

If using skip without specifying a target field, then control passes to the next field in the application. This targetless skip can only occur in the preproc of a field, roster, form, or user-defined function. CSPro will automatically figure out what the target field should be, as it does with an ask statement.

The target field can be located in any record at the same level as the current field, but it cannot be located at a different level. The field must be later on the path than the current field, meaning that it is a field that has not yet been entered. If the field has already been entered, an error message will be displayed during data entry. If you do not know whether the field is earlier in the data path, use the move statement.

When a skip statement is executed, the preproc of the target field will be executed but none of the statements between the skip statement and the preproc of the target field will be executed. Skipped fields are assigned the special value of notappl.

Note that the skip statement behaves differently from the advance statement, skipping *past* some number of fields, rather than moving *over* the fields.

Example 1

```
PROC ATTENDED_SCHOOL_YN

    if ATTENDED_SCHOOL_YN = 2 then
        skip to REASON_NEVER_ATTEND;
    endif;
```

Example 2

```
PROC CHILDREN_EVER_BORN

preproc

    if SEX = 1 or AGE < 12 then
        skip to next;
    endif;
```

Example 3

```
PROC MARRIAGE_AGE

preproc

    if MARITAL_STATUS = 1 then
        skip;
    endif;
```

See also: Advance Statement, Ask Statement, Move Statement, Reenter Statement

Userbar Function

It is possible to add a bar at the top of a data entry application to facilitate certain kinds of communication between a user and the intentions of the application programmer. For instance, a button can be added that allows the user to jump to a certain section of a questionnaire, or the bar can display reference text useful for the user. This "userbar" can display text, buttons, and text fields.

General Userbar Commands

```
b = userbar(show | hide | clear);
```

Show adds a userbar to the application, or if one has been added and is hidden, the show command displays it again. The **hide** command removes the userbar from the data entry window but keeps the userbar's contents in case it is shown again later. The **clear** command removes the userbar from the data entry window and deletes the contents of the userbar. The function returns 1 if successful, 0 otherwise. If the first command to a userbar is an add command, the userbar will be automatically displayed. If you want to add items to the userbar without displaying it, call hide or clear before adding the items.

Adding Items to the Userbar

```
i = userbar(add text | button | field | spacing,[...]);
```

All of the add commands, detailed below, return a resource identifier, an integer that uniquely points to the added object. If you plan to modify or remove objects added to the userbar, it is necessary to maintain this resource identifier as a way to identify on what object the modify or remove command should work. Objects are added to the userbar in left-to-right format. There is no way to add an item in between two existing items.

Adding Text to the Userbar

```
i = userbar(add text,alpha-expression);
```

The *alpha-expression* is a string of text that will be displayed on the userbar. The color of the text can be modified using the set color command.

Adding Buttons to the Userbar

```
i = userbar(add button,alpha-expression[,function-name]);
```

The button text comes from the *alpha-expression*. An optional argument specifies a function that is to be called when the button is pressed. If the function has parameters, it is necessary to specify the values of the arguments. The values of these arguments are evaluated when the button is activated by a user, not at the point that the button is added to the userbar. Several existing functions (mimicking options available on the CSEntry menu) can be specified using the do command:

```
i = userbar(add button,alpha-expression,do(alpha-expression));
```

The *alpha-expression* within the **do** command can be one of the following values:

"NextField", "PreviousField", "AdvanceToEnd", "EditNote", "ChangeLanguage", "PartialSave", "FieldHelp", "InsertLevelOcc", "AddLevelOcc", "DeleteLevelOcc", "InsertGroupOcc", "InsertGroupOccAfter", "DeleteGroupOcc", "SortGroupOcc", "PreviousScreen", "NextScreen", "EndGroupOcc", "EndGroup", "EndLevelOcc", "EndLevel", "FullScreen", "ToggleResponses", or "ToggleAllResponses"

Adding Fields to the Userbar

```
i = userbar(add field,alpha-expression[,function-name]);
```

The field's initial text comes from the *alpha-expression*. As with adding buttons, an optional argument specifies a function that is called when the user presses enter while typing text in the field. The width of the field depends on the initial text, so if you want a large field that starts with a blank value, use a long but blank string to initialize the field. Modifying the text of the field will not affect its size.

Adding Spacing to the Userbar

```
i = userbar(add spacing,numeric-expression);
```

If you want to space out the items on the userbar, specify in *numeric-expression* the number of pixels to leave blank

between the last-to-be and next-to-be added items.

Modifying Items on the Userbar

```
b = userbar(modify,resource-identifier,alpha-expression); // for text, buttons, and fields
b = userbar(modify,resource-identifier,function-name); // for buttons and fields
b = userbar(modify,resource-identifier,alpha-expression,function-name); // for buttons and fields
b = userbar(modify,resource-identifier,numeric-expression); // for spacing
```

To modify an item on the userbar, use the resource identifier that the function returned when the item was added to the userbar and specify the changes to the item. For buttons and fields, the text of the item and/or the function can be modified. The function returns 1 if successful, 0 otherwise.

Removing Items from the Userbar

```
b = userbar(remove,resource-identifier);
```

The remove command takes the item pointed to by the resource identifier off the userbar. The function returns 1 if successful, 0 otherwise.

Changing the Color of Items

```
b = userbar(set color[,resource-identifier],red-value,green-value,blue-value);
```

The background color of the userbar is changed by specifying the RGB (0 to 255) values of the desired color. Alternatively, if a resource identifier of a text item is passed, the color of the text item is changed. The function returns 1 if successful, 0 otherwise.

Identifying the Source of an Action

```
i = userbar(get);
```

The get command returns the resource identifier of the last clicked button or the last userbar field in which data was entered. If no event has occurred, the function returns 0.

Obtaining the Contents of a Field

```
b = userbar(get,resource-identifier,alpha-variable);
```

In this format, the get command returns the contents of a userbar field identified by the resource identifier. *Alpha-variable* must be an alphanumeric variable declared in the PROC GLOBAL section of the code. The function returns 1 if successful, 0 otherwise.

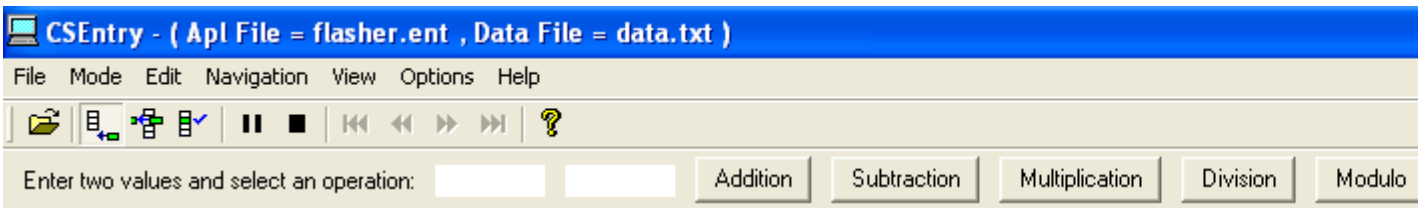
Userbar on Mobile Devices

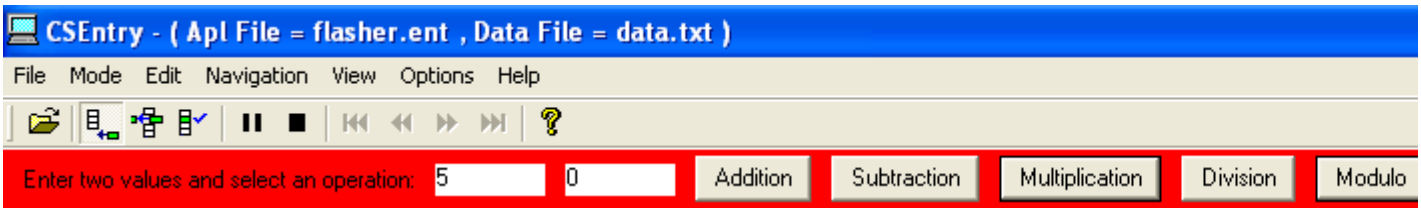
The userbar shows up as an icon to the left of the field note icon. Only buttons are displayed on the userbar so any color changes or additions of fields or spacing will be ignored. Options that mimic the CSEntry menu, added via the do(...) command, will also be ignored.

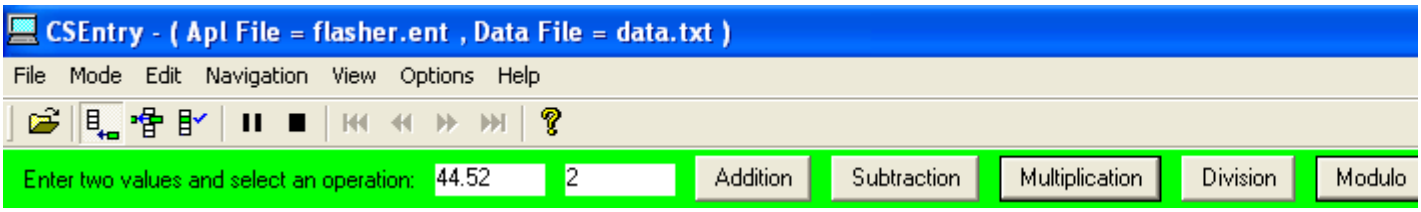
Example

```
PROC GLOBAL

numeric ADD_OP = 1, SUBTRACT_OP = 2, MULT_OP = 3, DIVIDE_OP = 4, MOD_OP = 5, EXP_OP = 6;

numeric leftOperatorRID,rightOperatorRID,resultsRID;

alpha (30) leftOperator,rightOperator;
alpha (1) operatorTypeString;

function performMathOperation(numeric operationType)

    userbar(get,leftOperatorRID,leftOperator);
    userbar(get,rightOperatorRID,rightOperator);
```

```
    numeric leftNumber = tonumber(leftOperator);
    numeric rightNumber = tonumber(rightOperator);

    if countnonspecial(leftNumber,rightNumber) <> 2 or leftOperator = "" or
rightOperator = "" then
        userbar(modify,resultsRID,"The supplied values are invalid!");
        userbar(set color,255,0,0);

    elseif rightNumber = 0 and operationType in DIVIDE_OP,MOD_OP then
        userbar(modify,resultsRID,"0 cannot be a denominator in a divide or modulo
expression!");
        userbar(set color,255,0,0);

    else
        numeric result;

        if      operationType = ADD_OP      then result = leftNumber + rightNumber;
operatorTypeString = '+';
        elseif  operationType = SUBTRACT_OP then result = leftNumber - rightNumber;
operatorTypeString = '-';
        elseif  operationType = MULT_OP     then result = leftNumber * rightNumber;
operatorTypeString = '*';
        elseif  operationType = DIVIDE_OP   then result = leftNumber / rightNumber;
operatorTypeString = '/';
        elseif  operationType = MOD_OP      then result = leftNumber % rightNumber;
operatorTypeString = '%';
        elseif  operationType = EXP_OP      then result = leftNumber ^ rightNumber;
operatorTypeString = '^';
        endif;

        userbar(modify,resultsRID,maketext("%0.2f %s %0.2f =
%0.2f",leftNumber,operatorTypeString,rightNumber,result));
        userbar(set color,0,255,0);

    endif;

end;


PROC USERBAR_FF

preproc

    userbar(clear);
    userbar(add text,"Enter two values and select an operation:");
    leftOperatorRID = userbar(add field,"                    ");
    rightOperatorRID = userbar(add field,"                    ");
    userbar(add button,"Addition",performMathOperation(ADD_OP));
    userbar(add button,"Subtraction",performMathOperation(SUBTRACT_OP));
    userbar(add button,"Multiplication",performMathOperation(MULT_OP));
    userbar(add button,"Division",performMathOperation(DIVIDE_OP));
    userbar(add button,"Modulo",performMathOperation(MOD_OP));
    userbar(add button,"Exponentiation",performMathOperation(EXP_OP));
    userbar(add spacing,50);
    resultsRID = userbar(add text," ");
    userbar(show);
```







See also: Capture Types

VisualValue Function

Format

```
d = visualvalue(field_name);
```

Description

The `visualvalue` function is used to access the contents of a data item before the data item has been entered. When an item is on a form as a field, CSPro will give its value in logic as ***notappl*** until the item has been entered. If you know that the field has been previously filled, perhaps because the operator is in modify mode or is resuming from a partial save, you can use this function to get the value stored in the item that is on the form and represented as ***field_name***.

This function is for use only with numeric fields. Refering to alphanumeric fields will always result in the evaluation of the field's visual value.

Return Value

The function returns the numeric value of the field.

Example

```
PROC INTERVIEW_START_DATE

preproc

    // do not overwrite the start date if modifying the case
    if visualvalue(INTERVIEW_START_DATE) = notappl then
        INTERVIEW_START_DATE = sysdate("YYYYMMDD");
    endif;
```

See also: Highlighted Function

Batch Edit Statements

EndCase Statement

Format

```
endcase;
```

Description

The endcase statement ends batch editing for the current questionnaire (case). All remaining procedures beyond where the statement is executed will be skipped. The statement is similar to the skip case command, but in the latter command the case is not saved to the output file. When using endcase the case will be saved to the output file.

Example

```
if HHTYPE = 2 then
    endcase;
endif;
```

See also: Skip Case Statement, EndLevel Statement

Export Statement

Format:

```
Export to file-name
    [rec_name(rec-name | alpha-exp)]
    [rec_type(rec-name | alpha-exp)]
    [case_id ([item-list])]
    rec-item-list
```

Description:

The **export** statement writes a record to an export file. Export statements can only be coded in level procedures.

In the **to** phase, the *file-name* is a name declared in the **file** statement in PROC GLOBAL.

The **rec_name**, **rec_type**, and **case_id** phrases can each be coded only once but can be coded in any order. They all must be coded before the *rec-item-list*. The order in which **rec_type** and **case_id** are coded determines the order of output of the record type and case ids in the exported record.

The **rec_name** phrase is optional and only used when data are exported in CSPro format. When coded the **rec_name** phase is used to give a label and name to the record type to the CSPro data dictionary created by the export statement. If a *rec-name* is coded, then the label and name from that record name in the input data dictionary is used for the label and name of the record type created in the exported data dictionary. If an *alpha-exp* is coded, then the label of the record type in the exported data dictionary is the result of the alphanumeric expression and the name is derived from the label. If **rec_name** is not coded, the labels and record names in CSPro will be RECORD001, RECORD002, etc.

The **rec_type** phrase is optional. When coded it is used to place a record type on the exported data record. If a *rec-name* is coded, then the record type value from the record name in the input data dictionary is placed on the exported data file. If an *alpha-exp* is coded, then the value of the expression is placed on the exported data file.

The **case_id** phrase is optional. When coded it is used to place case id items on the exported data record. If **case_id()** is coded then ALL the case ids from the level in which the export statement is coded are placed on the

exported data record. If no **case_id** phrase is coded and export format is CSPro, the ALL case ids from ALL levels will be output.

The *rec-item-list* specifies the contents of the exported data record. This can be any combination of record names or item names.

Where possible users are encouraged to use the **Export Data** tool instead of the **export** statement.

Example:

```
{Export fertility data for women 15 to 54 years old}
PROC GLOBAL
numeric i;
file SPSS_EXPORT;

PROC CENSUS_2000_DICTIONARY_FF

PROC QUEST {export done at LEVEL procedure}
  set behavior() export ( SPSS , ItemOnly );

  do I = 1 until I > totocc(PERSON)
    if P03_SEX (i) = 2 and P04_AGE(i) in 15:54 then
      export to SPSS_EXPORT case_id (PROVINCE,DISTRICT,EA,HU,HH)
          LINE(i), P18_BORN(i), P19_LIVING(i), P20_BORN12(i);
      errmsg ("Record exported for female, 15 to 54") summary;
    endif;
  enddo;
```

See also: Set Behavior Export Statement, File Statement, SetFile Function, Export Data Tool

Freq Statement (Unnamed)

Format

```
freq
include(variables_to_tabulate)
[exclude(variables_not_to_tabulate)]
[disjoint]
[breakdown(length)]
[universe(condition)]
[weight(weight_value)]
[formatting_options]
;
```

Description

The `freq` statement is used to define a frequency table that will be written to the frequencies file. Unlike named frequencies, which allow for control over when frequencies are tallied, frequencies generated using the `freq` statement are tallied at the location where they are defined. Due to this, the statement cannot be located in `PROC GLOBAL`, in user-defined functions, or in application procedures.

The Tabulate Frequencies tool generates `freq` statements automatically, so one way to learn about how to use the statement is to select items to tabulate in that tool and then use the **View** -> **Batch Logic** option to see the commands used to generate the specified frequencies.

Specifying Variables to Tabulate

The `include` command is used to specify what variables should be tabulated. At least one frequency table will be created for every variable specified in the `include` list. The ***variables_to_tabulate*** is a list of variables, separated by commas, that can include:

- **Dictionary names**: dictionaries, records, and items
- **Form names**: forms, groups, and blocks
- **Logic names**: `numeric` and `string` variables

The optional `exclude` command is used to specify variables that should be removed from the inclusion list. The ***variables_not_to_tabulate*** is a list of variables as defined above. The `exclude` command is particularly useful when including records. For example, if you want to tabulate most items on a record with a couple exceptions, you might code:

```
include(PERSON_REC)
exclude(P25_RELIGION, P26_TRIBE)
```

When including or excluding a name that may contain more than one variable—dictionaries, records, forms, groups, and blocks—CSPro uses a rule to determine whether items contained in that grouping should be included in the list of variables to tabulate:

Default rule: Include the item if it has a value set; if no value set is defined, then include the item if it has length 1 - 4.

You can override the default selection by adding one or more of these flags to the `include`/`exclude` list:

Flag	Include/Exclude All...
`numeric`	numeric items
`float`	numeric items with decimals
`integer`	numeric items without decimals
`long`	numeric items without decimals and length 3 - 15
`short`	numeric items without decimals and length 1 - 2
`alpha`	alphanumeric items

These six flags will never include items that have subitems. You can use a combination of flags; for example, this would include all items from `PERSON_REC` with the exception of items that have subitems (though the subitems would be included).

```
include(numeric, alpha, PERSON_REC)
```

Leaving the `include` list empty is as if you coded the primary dictionary name. For example:

```
include()
// may be translated to:
include(CENSUS_DICT)
```

If you do not need to exclude any variables, the `include` command is optional. For example, these unnamed frequency statements are the same:

```
freq include(P03_SEX);
freq(P03_SEX);
```

Handling Variable Occurrences

By default, when frequencies are tallied for a multiply-occurring item, all of the occurrences of the item are tallied. For example, assuming that `P03_SEX` is on a record with 50 occurrences, coding this will tally all defined occurrences of `P03_SEX`:

```
freq include(P03_SEX);
```

If the first house is vacant, then no sex values are tallied; if the second house has three people, then three sex values are tallied; and so on.

If you would like to tally a specific occurrence, you can specify the occurrence in the `include`/`exclude` list. For

example, this would create two tables, one for all sex occurrences, and one for the head's sex (assuming that the head is the first occurrence):

```
freq include(P03_SEX, P03_SEX(1));
```

You can specify occurrence values when using items or records. `PERSON_REC(1)`, for example, would create tables for the first occurrence of the items in `PERSON_REC`. If you specify an occurrence, the value will be tallied regardless of whether the occurrence exists. For example, while `P03_SEX` would not tally vacant households, `P03_SEX(1)` will include tallies of blank values for vacant households.

An optional command, `disjoint`, is a shortcut way of indicating that a frequency table should be created for every occurrence of a variable. For example, this code would create a table for each of the occurrences of `P03_SEX`, resulting in 50 tables (for occurrence 1, occurrence 2, and so on until occurrence 50):

```
freq include(P03_SEX)
     disjoint;
```

When using `disjoint`, you can use `(*)` as an occurrence to specify that you would like to ignore the disjoint setting. For example, this code would create 49 tables (for the combined occurrences, for occurrence 3, occurrence 4, and so on until occurrence 50).

```
freq include(P03_SEX(*), P03_SEX)
     exclude(P03_SEX(1), P03_SEX(2))
     disjoint;
```

Alphanumeric Breakdown

The optional command `breakdown` allows you to control how alphanumeric items and string variables are tallied. A positive numeric constant, ***length***, specifies a number used to split these values before tallying. This can be useful when creating frequencies for data collected using checkboxes. For example, assuming `CHECKBOX_FIELD` occurs twice, first as "AB" and then as "BC":

```
freq include(CHECKBOX_FIELD); // results in: "AB" (1)
                              //             "BC" (1)

freq include(CHECKBOX_FIELD)  // results in:  "A" (1)
     breakdown(1);            //              "B" (2)
                              //              "C" (1)
```

Additional Commands

The optional command `universe` allows you to specify a condition under which the frequency should be tallied. The values will be tallied when the ***condition*** evaluates to true.

The optional command `weight` allows you to weight each tally. The ***weight_value*** can be a constant number (like `10`), a dictionary item (like `HH_WEIGHT`) or any other numeric expression. If no weight is provided, a weight of 1 is used during the tallying.

Optional formatting options allow you to control how the frequency tables are generated. The formatting options include the following commands: `valueset`, `distinct`, `vset`, `heading`, `stat`, `percentiles`, `nofreq`, `decimals`, `sort`, `nonetpercents`, and `pagelength`.

Example 1

```
PROC QUEST

    // create frequency tables for all numeric items in the
    // HOUSING_REC record with the exception of H13_PERSONS

    freq
    include(numeric, HOUSING_REC)
    exclude(H13_PERSONS);
```

Example 2

```
PROC QUEST

    // create frequency tables for the default selection of items in
    // the PERSON_REC record, tallying only the first record occurrence

    // universe is used to make sure that we do not tally empty households

    // weight is 20 because we are creating frequencies on a 5% sample file

    // distinct means that the frequency tables will show all values,
    // not combining values if the value sets have ranges (such as age 0-4)

    freq
    include(PERSON_REC(1))
    universe(totocc(PERSON_REC) > 0)
    weight(20)
    distinct;
```

Example 3

```
PROC QUEST

    // create a frequency table for the item H15_ASSETS, which
    // was collected using a checkbox with each value of length 2

    freq(H15_ASSETS)
    breakdown(2);
```

See also: Freq Object, Freq Statement (Named), Frequency Formatting Options

GetDeck Function

Format:

```
f = getdeck(array-name[,override-dim1,override-dim2,override-dim3]);
```

Description:

The **getdeck** function returns the value in the DeckArray hotdeck using the current values in the items identified by the value sets used in the declaration of the DeckArray. The function automatically recodes the values and accesses the proper cell in the hotdeck. If any of the dimensions of the DeckArray are not value set dimensions, you must specify the numeric index when calling the function.

Return value:

The function returns the value in the hotdeck or DEFAULT in the case that the values supplied are not valid entries in the value sets and thus could not be recoded to a proper cell in the hotdeck.

Example:

```
array education_HD_SexAge(SEX_VS,AGE_FOR_EDUCATION_HD_VS) save;

...

PROC EDUCATION

    EDUCATION = getdeck(education_HD_SexAge); // use current values for sex and age
    EDUCATION = getdeck(education_HD_SexAge,1); // override sex only
    EDUCATION = getdeck(education_HD_SexAge,,28); // override age only
    EDUCATION = getdeck(education_HD_SexAge,1,28); // override both value and age
```

See also: DeckArrays, DeckArray Leftover Rows, Putdeck Function

PutDeck Function

Format:

```
d = putdeck(array-name,numeric-expression[,override-dim1,override-dim2,override-dim3]);
```

Description:

The **putdeck** function updates the value in the DeckArray hotdeck using the current values in the items identified by the value sets used in the declaration of the DeckArray. The function automatically recodes the values and accesses the proper cell in the hotdeck, where it places the value of the *numeric-expression*. If any of the dimensions of the DeckArray are not value set dimensions, you must specify the numeric index when calling the function.

If a (+) is specified after the *array-name*, the "leftover" rows for the hotdeck will also be updated with the value. See the leftover rows page for more information.

Return value:

The function returns 1 if successful or DEFAULT in the case that the values supplied are not valid entries in the value sets and thus could not be recoded to a proper cell in the hotdeck.

Example:

```
array education_HD_SexAge(SEX_VS,AGE_FOR_EDUCATION_HD_VS) save;

...

PROC EDUCATION

    putdeck(education_HD_SexAge,EDUCATION); // use current values for sex and age
    putdeck(education_HD_SexAge,EDUCATION,1); // override sex only
    putdeck(education_HD_SexAge,EDUCATION,,28); // override age only
    putdeck(education_HD_SexAge,EDUCATION,1,28); // override both value and age
```

See also: DeckArrays, DeckArray Leftover Rows, Getdeck Function

Set Behavior Export Statement

Format:

```
set behavior() export (model[, item-type, text-encoding, decimal-mark]);
```

Description:

The **set behavior export** statement is coded before the first **export** statement.

The *model* is required. It is one of the keywords **SPSS**, **SAS**, **Stata**, **R**, **All**, **CSPro**, **TabDelim**, **CommaDelim**, or **SemiColonDelim**, indicating the type of file being exported.

Item-type is optional. It is one of the keywords **ItemOnly**, **SubitemOnly**, or **ItemSubitem**, indicating how subitems and their parent item are handled when entire records are exported. If not coded, **SubitemOnly** is assumed.

Text-encoding is optional. It is either **ANSI** or **Unicode**, and specifies the text encoding of the exported data file and any description or script files. If not coded, **ANSI** is assumed. The **Unicode** option will output UTF-8 files.

Decimal-mark is optional. If specified with the keyword **CommaDecimal**, decimal marks will use commas instead of periods in the exported data file.

See also: Export Statement

SetOutput Function

Format

```
b = setoutput(file_name);
```

Description

The `setoutput` function redirects the output cases of a batch application to the data file specified in the string expression ***file_name***. All data will be appended to (added to the end of) the file. If the file exists prior to the application run, you may, depending on the circumstances, want to use the `filedelete` function to remove the file.

You can use a single file name as an argument or you can specify a string list. Using a list allows you to specify multiple output files. Cases will be output to each of the output files specified.

This function cannot be used in data entry applications.

Return Value

The function returns a logical value of 1 (true) if at least one physical file is successfully assigned and 0 (false) otherwise.

Example

```
PROC PROVINCE

    // this will split the contents of the input data file into
    // one file for each province code
    setoutput(maketext(".\split\%02d.dat", PROVINCE));
```

See also: SetFile Function, Skip Case Statement

Skip Case Statement

Format

```
skip case;
```

Description

The `skip case` statement ends batch edit processing of the current case and skips to the next case in the input file. If an output file is specified, the skipped case is not saved in the output file.

Example

```
// remove any vacant households
if totocc(PERSON_REC) = 0 then
    skip case;
endif;
```

See also: EndCase Statement, Stop Function

Numeric Functions

ABS Function

Format

```
d = abs(numeric value);
```

Description

The `abs` function returns the absolute value of the numeric expression ***value***.

Return Value

The function returns the absolute value. If the value of the numeric expression is a special value, the function returns the special value.

Example

```
if abs(X - target) < abs(Y - target) then
    // ...
endif;
```

CMCode Function

Format

```
i = cmcode(month, year);
```

Description

The `cmcode` function returns the *century month code* (CMC) of the given date using the ***month*** and ***year*** numeric expressions. The CMC is the number of months since January 1900. (The CMC for January 1900 is 1.) It is calculated by multiplying the number of years between the argument ***year*** and 1900 by twelve, then adding the value of argument ***month***.

The function returns the value 9999 if the month is less than one or greater than 12, or if either the month or year are equal to a special value. The function accepts either 2- or 4-digit years. If a 2-digit year is used, the function assumes that the year is in the 20th century (i.e., 19xx). Four-digit years can be used for years in the 20th or 21st century.

Return Value

The function returns the CMC of the date.

Example 1

```
XMONTH = 6;
XYEAR = 82;
DATE = cmcode(XMONTH, XYEAR);
```

The value of DATE for the given arguments (June 1982) would be (82 * 12) + 6 = 990.

Example 2

```
XMONTH = 2;
XYEAR = 2000;
DATE = cmcode(XMONTH, XYEAR);
```

The value of DATE in this example would be ((2000 - 1900) * 12) + 2, or 1202.

See also: DateAdd Function, DateDiff Function, DateValid Function, SysDate Function

CountNonSpecial Function

Format

```
i = countnonspecial(item_name ‖ record_name ‖ array_name ‖ numeric_expression⌈,...⌉);
```

Description

The function `countnonspecial` counts the number of non-special values within the passed arguments. Special arguments include `missing`, `refused`, `notappl`, and `default`. The function can receive multiple numeric arguments, including items, records, arrays, and numeric expressions. This function can greatly simplify some programming tasks; for example, the following two lines of code are the same:

```
if countnonspecial(RELATIONSHIP,SEX,AGE) = 3 then

if not special(RELATIONSHIP) and not special(SEX) and not special(AGE) then
```

If an array is passed, the function will count the non-special values among all the values in the array. For a multiply occurring item or record, to count the non-special values for all the items or records instead of just the current occurrence, include (*) after the item or record name.

`CountValid` is a synonym for this function.

Example 1

```
PROC GLOBAL

    numeric tempVal = notappl;
    array tempArray(2,3) = 1 2 3 notappl missing default;

PROC EXAMPLE

    countnonspecial(5,tempVal,default,28 + 31,3 / 0); // returns 2

    tempVal = 0.123456789;
    countnonspecial(5,tempVal,default,28 + 31,3 / 0); // returns 3

    countnonspecial(5,tempVal,default,28 + 31,3 / 0,tempArray); // returns 6

    tempArray(2,1) = 0;
    countnonspecial(5,tempVal,default,28 + 31,3 / 0,tempArray); // returns 7
```

Example 2

```
// in this example SEX is not on the FERTILITY record
PROC SEX

if SEX <> 2 and countnonspecial(FERTILITY(curocc())) > 3 then
    errmsg("Sex (%d) is not female but 4+ fertility entries defined so sex changed to
female",SEX);
    impute(SEX,2);
endif;
```

Example 3

```
numeric numDefinedValues = countnonspecial(SEX(*),AGE(*));
numeric numPossibleValues = 2 * totocc(POPULATION);

errmsg("%d% of sex and age values are missing",100 * ( 1 - ( numDefinedValues /
numPossibleValues ) ));
```

See also: Special Function, Special Values

Exp Function

Format

```
d = exp(numeric_expression);
```

Description

The `exp` function raises the value of e (2.7182818...) to the power given by ***numeric_expression***. This value e is called Napier's constant or Euler's number and is the basis of natural logarithms.

Return Value

The function returns a decimal number. If the value of ***numeric_expression*** is a special value, the function returns that value.

Example

```
X = exp(Y);
```

High Function

Format

```
d = high(number1[,number2...numberN]);
```

Description

The `high` function returns the maximum (highest) value in a group of numbers represented by numeric expressions ***number1*** ... ***numberN***. The function ignores special values.

Return Value

The function returns the highest value, or `default` in the case that no valid values were passed.

Example

```
errmsg("Lowest value is %f",low(50,-123.45,1982.0605,20)); // displays -123.45
errmsg("Highest value is %f",high(50,-123.45,1982.0605,20)); // displays 1982.0605
```

See also: Low Function, MaxValue Function

Inc Function

Format

```
i = inc(numeric_item[,increment_value]);
```

Description

The `inc` function increments ***numeric_item***, which is either a dictionary item or a numeric variable. If the optional numeric expression ***increment_value*** is present, then the value of that expression is added to ***numeric_item***. If no expression is present, 1 is added to ***numeric_item***. The increment value can be negative or nonnegative. `inc(X)` is essentially shorthand for `X = X + 1;`

Return Value

The function returns the sum of the numeric item and the increment value.

Example

```
X = 5;

inc(X);
// X is 6 after X = X + 1;

inc(X,4);
// X is 10 after X = X + 4;

inc(X,-5);
// X is 5 after X = X + (-5);

X = 5 + inc(X,inc(X));
// X is 17 after X = X + 1; X = X + X; X = 5 + X;
```

Int Function

Format

```
i = int(numeric_expression);
```

Description

The `int` function returns the integer portion of the decimal value ***numeric_expression***.

Return Value

The function returns an integer value. If the value of ***numeric_expression*** is a special value, the function returns that value.

Example

```
X = int(5 / 3); // X will be 1
```

See also: Round Function

Log Function

Format

```
d = log(numeric_expression);
```

Description

The `log` function calculates the base-10 logarithm of ***numeric_expression***. The number ***numeric_expression*** should be a positive value.

Return Value

The function returns a decimal number logarithm. If the value of the number is a special value, the function returns the special value given. If the value of the number is negative, the function returns `default`.

Example

```
numeric incomeLog = log(INCOME);
```

Low Function

Format

```
d = low(number1[,number2...numberN]);
```

Description

The `low` function returns the minimum (lowest) value in a group of numbers represented by numeric expressions ***number1*** ... ***numberN***. The function ignores special values.

Return Value

The function returns the lowest value, or `default` in the case that no valid values were passed.

Example

```
errmsg("Lowest value is %f",low(50,-123.45,1982.0605,20)); // displays -123.45
errmsg("Highest value is %f",high(50,-123.45,1982.0605,20)); // displays 1982.0605
```

See also: High Function, MinValue Function

Random Function

Format

```
i = random(min_value, max_value);
```

Description

The `random` function returns a uniformly distributed random number between the number expressions ***min_value*** and

max_value inclusive.

By default the sequence of random numbers generated by repeated calls to random will always be the same. You can use the `seed` function to generate a different sequence.

Return Value

The function returns an integer random value. The function will return a value of `default` if ***min_value*** is greater than ***max_value*** or if either value is equal to one of the special values.

Example

```
NUMBER = random(1, 100);
```

See also: RandomIn Function, ValueSet.Randomize Function, Seed Function, UUID Function

RandomIn Function

Format

```
d = randomin(in_list);
```

Description

The `randomin` function returns a random number from a grouping of numeric values expressed as an in list. You can use the `seed` function to initialize the random number generation. A non-integer can appear as part of the in list, but if it is part of a range, both the low and high values of the range will be converted to integers and the function will return a uniformly distributed random integer in that range.

Return Value

The function returns a random value from the values of the in list. The function will return `default` if there were no applicable values to construct a group of valid numbers from which to pick a random number. If a value appears more than once in the in list, it will have a higher probability of being selected by the function.

Examples

```
errmsg("Random tribe code: %d", randomin(TRIBE_VS1));

errmsg("Non-continuous random number: %d", randomin(-100:-50, 50:100, 999));

errmsg("After many calls, 1.23 will be selected 75% of the time:
%d", randomin(1.23, 1.23, 1.23, 8));

errmsg("Random month: %d", randomin(1:12)); // same as random(1, 12)
```

See also: Random Function, ValueSet.Randomize Function, Seed Function

Round Function

Format

```
i = round(numeric_expression);
```

Description

The round function returns the rounded value of ***numeric_expression***, rounding the number to the integer that is closest to the number.

Return Value

The function returns the rounded value. If the value of the numeric expression is a special value, the function returns the special value.

Example

```
// round a price to the nearest nickel (five cents)
numeric roundedPrice = round(PRICE * 20) / 20;
```

See also: Int Function

Seed Function

Format

```
b = seed(numeric_expression);
```

Description

The seed function is used to determine the first value generated by the random function. If you want to reproduce a series of random numbers, then for best results, the ***numeric_expression*** should be set to a prime number, such as 1009. However, if you want different random numbers for each run of your program, the systime and timestamp functions return good seed values.

Versions of CSPro 6.0 and greater will return different random numbers, given the same seed value, than earlier versions of CSPro. If you expect a certain sequence to the random numbers and want to match a previously recorded sequence, then you will want to use an old version of CSPro to run your application.

Return Value

The function returns a logical value of 1 (true) if the seeding is successful, and 0 (false) otherwise.

Example

```
seed(systime());

numeric randomAge = random(1,99);
```

Sqrt Function

Format

```
d = sqrt(numeric_expression);
```

Description

The sqrt function returns the square root of ***numeric_expression***. The number ***numeric_expression*** should be a positive value.

Return Value

The function returns a decimal value of the square root of the expression. If the value of the number is a special value, the function returns the special value given. If the value of the number is negative, the function returns `default`.

Example

```
numeric incomeSqrt = sqrt(INCOME);
```

Set Behavior SpecialValues Statement

***Feature Upgrade**: Starting with CSPro 7.1, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setproperty` function with the argument `"SpecialValuesZero"`.*

Format:

```
set behavior() specialvalues(zero) on | off;
```

Description:

The **set behavior specialvalues** statement allows special values to be treated as zero (0) values during arithmetic operations, including addition, subtraction, multiplication, division, and modulo. By default this behavior is disabled. The set behavior statement affects all numeric data items from the point where it is executed onward. To limit its scope, it must be turned on and off at appropriate times.

Example:

```
PROC GLOBAL

   numeric var1,var2,var3;
   numeric result;

PROC SUMMATION

   var1 = 5;
   var2 = 10;
   var3 = default;

   set behavior() specialvalues(zero) on;
   result = var1 + var2 + var3; { result will be 15 }

   set behavior() specialvalues(zero) off;
   result = var1 + var2 + var3; { result will be DEFAULT }
```

See also: Special Values

String Functions

Compare Function

Format

```
i = compare(string1,string2);
```

Description

The `compare` function compares the two strings expressions ***string1*** and ***string2*** character by character to determine the alphabetical (collating sequence) order of the strings. If ***string1*** and ***string2*** are of different lengths, the function will pad the shorter string with blanks to carry out the comparison.

Return Value

The function returns an integer value of:

-1	if ***string1*** would be listed alphabetically before ***string2.***
0	if the strings are identical.
1	if ***string1*** would be listed alphabetically after ***string2***.

Example

```
if compare(thisCaseKey,previousCaseKey) = -1 then
    // ...
endif;
```

Direct string comparisons can also be made. For example, the following code is permissible:

```
if thisCaseKey < previousCaseKey then
    // ...
endif;
```

Concat Function

Format

```
s = concat(string1, string2[, ..., stringN]);
```

Description

The `concat` function concatenates the values of two or more string expressions (***string1*** + ***string2*** + ... + ***stringN***). The strings can be alphanumeric items, text strings, or functions that return strings. You can also use the + operator to concatenate strings.

Return Value

The function returns the concatenated string.

Example

```
FIRST_NAME = "John";
LAST_NAME = "Henry";

string full_name = concat(FIRST_NAME, " ", LAST_NAME); // full_name is: John Henry
```

See also: String Concatenation Operator, PathConcat Function

String Concatenation Operator

In addition to serving as the addition operator for numeric values, the + operator can be used to concatenate strings, similar to the `concat` function. For example, both lines of code below will give the same result:

```
string  text1 = "Hello",
        text2 = "World!";

string messagePlus = text1 + ", " + text2;

string messageConcat = concat(text1,", ",text2);
```

The + operator is a convenient way to build strings and can be used along with other functions that return alphanumeric values.

Example

```
string message;

do numeric counter = 1 while counter <= 13 by 2
    message = message + edit("99 ",counter);
enddo;

// message is now: "01 03 05 07 09 11 13"
```

Edit Function

Format:

```
s = edit(edit-pattern,numeric-expression);
```

Description:

The edit function converts a number to a character string defined by the given "edit pattern". The "edit pattern" is a string containing "Z"s or "9"s (i.e., "9999" or "ZZ9.99"). Both "9" and "Z" represent a digit.

9 display a digit
Z display a digit, but if it is a leading zero, display a blank
. display the decimal character
, display the thousands separator character

Any other character will be displayed as itself.

Return value:

The function returns a string derived from the "numeric-expression" argument.

Example 1:

```
X = 87;
A1 = edit("ZZZ9",X); yields A1 = " 87"
A2 = edit("9999",X); yields A2 = "0087"
A3 = edit("Z999",X); yields A3 = " 087"
```

Example 2:

```
Y = 0;
```

```
A4 = edit("ZZ9",Y); yields A4 = " 0"
A5 = edit("999",Y); yields A5 = "000"
A6 = edit("ZZZ",Y); yields A6 = " "
```

Example 3:

```
A = edit("99:99:99",sysdate());
```

Example 4:

```
A = edit("99/99/99",sysdate("DDMMYY"));
```

Example 5:

```
A = edit("ZZZ,ZZZ,ZZ9",INCOME);
```

See also: Tonumber Function , Sysdate Function

GetBuffer Function

Format

```
s = getbuffer(item_name);
```

Description

The `getbuffer` function returns a string containing a data item's contents. The ***item_name*** can refer to either a numeric or alphanumeric item.

The function is especially useful when a numeric data item in a data file contains a non-numeric value, such as "*", "-", or "a". You cannot test the contents of the numeric data item for alphanumeric values because CSPro stores `default` as the value of any numeric data item which contains non-numeric values. Therefore, to find out what non-numeric value(s) exist in a data item, you would use this function to return its contents in the form of a string.

Return Value

The function returns a string containing the item's contents.

Example

```
if special(AGE) then
    errmsg("Person's age is invalid, AGE = %s",getbuffer(AGE));
endif;
```

Length Function

Format

```
i = length(string_expression);
```

Description

The function `length` calculates the length of an alphanumeric ***string_expression***. If the string is a dictionary item, the value returned is the length of the item. If it the string is the result of a function, the value returned is the length of the string returned by the function.

Return Value

The function returns the length of the string as an integer value. If using an `alpha` variable or a dictionary item, you may

want to strip the string before calculating its length.

Example

```
alpha (30) NAME = "John Henry";
length(NAME);                    // returns 30
length(strip(NAME));             // returns 10
```

Uses on Lists and Arrays

The length function also operates on lists and arrays. When applied to a list, it returns the number of elements in the list. When applied on an array, it returns the size of an array dimension.

Format

```
i = length(list_name);
i = length(array_name[, dimension]);
```

Examples

```
array string usmca_countries(3) = "Canada", "Mexico", "United States";
length(usmca_countries); // returns 3;

// ...

array AgeSexDeckArray(AGE_VS, SEX_VS);

do numeric age_counter = 1 while age_counter <= length(AgeSexDeckArray, 1)
    do numeric sex_counter = 1 while sex_counter <= length(AgeSexDeckArray, 2)
        // ...
    enddo;
enddo;
```

See also: Array.Length Function

MakeText Function

Format 1

```
s = maketext(string_exp[, argument1, ..., argumentN]);
```

Format 2

```
s = maketext(msg_num[, argument1, ..., argumentN]);
```

Description

The maketext function formats a text string with inserted values. Each argument is sequentially inserted into the text string. Arguments can be numeric or alphanumeric expressions, but the type of the argument must match the type of the receiving field in the string expression.

In the string expression:

%**[n]d** = Insert a number and display it as an integer
%**[n.d]f** = Insert a number and display it as a decimal value
%**[n.d]s** = Insert a text string

where "**n**" is the size of the field and "d" is the number of decimal places to show for a number. *For a complete list of*

possible message formatters, see Message Formatting Options.

Numbers are never truncated. Text strings are truncated only if "**.d**" is used. If "**n**" is positive, the insert is right-justified in the size of the field. If "**n**" is negative, the insert is left-justified in the size of the field. If "**n**" is a positive number with a leading zero, the insert is right-justified in the size of the field and zero-filled to the left. When inserting a number, if "**n**" is preceded by a "+", the sign of the number is always displayed.

Return Value

The function returns the formatted string.

Example 1

```
TEXT = maketext("Sex= %d", SEX);
```

Example 2

```
numeric integerValue = 23456;

errmsg( "%d"       , integerValue);      //  23456
errmsg( "%-10d"    , integerValue);      //  23456
errmsg( "%10d"     , integerValue);      //       23456
errmsg( "%+10d"    , integerValue);      //      +23456
errmsg( "%+010d"   , integerValue);      //  +000023456
errmsg( "%-010d"   , integerValue);      //  0000023456
errmsg( "%f"       , integerValue);      //  23456.000000  Note the usage of %f


numeric decimalValue = 12.567;

errmsg( "%f"        , decimalValue);    //   12.567000
errmsg( "%-10.3f"   , decimalValue);    //   12.567
errmsg( "%d"        , decimalValue);    //   12           Note the usage of %d
errmsg( "%10.2f"    , decimalValue);    //        12.57
errmsg( "%10.3f"    , decimalValue);    //       12.567
errmsg( "%+10.3f"   , decimalValue);    //      +12.567
errmsg( "%+010.3f"  , decimalValue);    //   +00012.567
errmsg( "%010.3f"   , decimalValue);    //   000012.567
errmsg( "%10.5f"    , decimalValue);    //     12.56700


string stringValue = "abcdef";

errmsg( "%s"        , stringValue);     //   abcdef
errmsg( "%-10s"     , stringValue);     //   abcdef
errmsg( "%-10.3s"   , stringValue);     //   abc
errmsg( "%10s"      , stringValue);     //       abcdef
errmsg( "%10.3s"    , stringValue);     //          abc
```

See also: Message Formatting Options, Message File (.mgf)

Message Formatting Options

Format

```
function_name(message || message_number⌡, argument1, ..., argumentN⌠);
```

Description

Several functions* take an argument that consists of a message and then an optional number of arguments that are dynamically inserted into the text to create a final message *(*the complete list of functions can be found at the bottom of this page)*. If ***message_number*** is used, it refers to the the user-numbered messages created in the .mgf file.

The message can contain optional arguments, ***argument1*** to ***argumentN***, that will be inserted into the message. Each argument, denoted in the message text with a percent sign, is sequentially inserted into the message. Arguments can be numeric or string expressions, but the type of the argument must match the type of the receiving field in the message text. There are seven formatters:

Formatter	Result
"%d"	Inserts a number that is displayed as an integer.
"%f"	Inserts a number that is displayed as a decimal value (with six decimal places shown by default).
"%s"	Inserts a text string.
"%v"	Inserts a variable formatted intelligently (see below).
"%l"	Inserts a variable's value label (see below).
"%p"	Inserts the name of the current procedure, which can be useful for debugging. No argument should be supplied if using this option.
"%c"	Inserts the first character of a text string, or no characters if the string is blank.

Variable Intelligent Formatting

The "%v" formatter works with either numeric or string variables, whether a dictionary item or a temporary variable. If the formatter is used on an alphanumeric dictionary item or a string variable, then it displays it in the same way as would "%s". However, if a numeric dictionary item is used, then it formats the number in the way that it would be saved, obeying the length, zero fill, and decimal attributes of the field. If a numeric temporary variable is used, then the number is displayed as an integer ("%d") if applicable, or as a decimal value ("%f") with up to six decimal places otherwise.

The "%l" formatter looks up the value set label associated with the variable's current value. If found, it displays that label (in the same way that the getvaluelabel function does). If no value set label is associated with the value, then the "%l" formatter behaves as if it were the "%v" formatter.

Additional Formatting Options

Numeric and string arguments have a couple additional options that can be specified after the percent sign: "%[n]d", "%[n.d]f", and "%[n.d]s". In these cases, "n" controls the length of the replacement text and "d" controls either the number of decimal places shown or the length of the string. There are some rules:

- If the number is positive, as in "%+5d", the text is right justified to the size of the field (e.g., "   9").
- If the number is negative, as in "%-5d", the text is left justified to the size of the field (e.g., "9   ").
- For numbers, if a leading zero proceeds the number, as in "%05d", the text is right justified to the size of the field but instead of being padded with spaces, it is padded with zeros (e.g., "00009").
- For numbers, if the number is preceded by a plus sign, as in "%+d", the sign of the number is always displayed (e.g., "+9").
- Numbers are never truncated. Text strings are not truncated unless used as in "%0.5s", where the second part, ".5", indicates the maximum number of characters.

Example

```
numeric integerValue = 23456;

errmsg( "%d"       , integerValue);      //  23456
errmsg( "%-10d"    , integerValue);      //  23456
errmsg( "%10d"     , integerValue);      //       23456
errmsg( "%+10d"    , integerValue);      //      +23456
errmsg( "%+010d"   , integerValue);      //  +000023456
errmsg( "%-010d"   , integerValue);      //  0000023456
errmsg( "%f"       , integerValue);      //  23456.000000  Note the usage of %f


numeric decimalValue = 12.567;

errmsg( "%f"        , decimalValue);    //   12.567000
errmsg( "%-10.3f"   , decimalValue);    //   12.567
errmsg( "%d"        , decimalValue);    //   12            Note the usage of %d
errmsg( "%10.2f"    , decimalValue);    //        12.57
errmsg( "%10.3f"    , decimalValue);    //       12.567
errmsg( "%+10.3f"   , decimalValue);    //      +12.567
errmsg( "%+010.3f"  , decimalValue);    //   +00012.567
errmsg( "%010.3f"   , decimalValue);    //   000012.567
errmsg( "%10.5f"    , decimalValue);    //     12.56700


string stringValue = "abcdef";

errmsg( "%s"        , stringValue);     //   abcdef
errmsg( "%-10s"     , stringValue);     //   abcdef
errmsg( "%-10.3s"   , stringValue);     //   abc
errmsg( "%10s"      , stringValue);     //       abcdef
errmsg( "%10.3s"    , stringValue);     //          abc
```

See also: ErrMsg Function, FileWrite Function, File.Write Function, LogText Function, MakeText Function, Trace Function, Warning Function, Write Function

Pos Function

Format

```
i = pos(substring, source_string);
```

Description

The `pos` function searches for a grouping of characters, represented in a ***substring***, within a ***source_string***. The function returns the beginning position of the first occurrence of the substring. Both arguments are string expressions, and they are case sensitive, meaning that `"children"` is recognized as different from `"CHILDREN"`.

If using an `alpha` string or dictionary item, be aware that searching for a space character can lead to the function finding one at the end of your string. You may want to `strip` alphanumeric variables before searching for spaces.

Return Value

The function returns the position of the first occurrence of the substring, or if the substring is not found, the function returns 0.

Examples

```
VALUE = pos("L", "FOR THE CHILDREN");   // VALUE will be 12; this is where the
                                        // substring (the letter "L") occurs in the
                                        // source string ("FOR THE CHILDREN")

VALUE = pos("DRE", "CHILDREN");         // VALUE will be 5; this is where the
                                        // substring ("DRE") occurs in the source string

VALUE = pos("DCN", "CHILDREN");         // VALUE will be 0; the substring ("DCN") does
                                        // not occur in the source string
```

See also: PosChar Function, StartsWith Function, Replace Function, IsChecked Function

PosChar Function

Format

```
i = poschar(pattern_string, source_string);
```

Description

The `poschar` function searches for a collection of characters, represented in a ***pattern_string***, within a ***source_string***. The function returns the beginning position of the first occurrence of the pattern string in the source string. Both arguments are string expressions, and they are case sensitive, meaning that `"c"` is recognized as different from `"C"`.

Return Value

The function returns the position of the first occurrence of the pattern string, or if no characters from the pattern string are found, the function returns 0.

Examples

```
VALUE = poschar("L", "CHILDREN");   // VALUE will be 4; this is where the
                                    // pattern string (the letter "L")
                                    // occurs in the source string ("CHILDREN")

VALUE = poschar("LCN", "CHILDREN"); // VALUE will be 1; of the characters in the
                                    // pattern string, "C" is the first character
                                    // encountered in the source string, and it is
                                    // found at position 1
```

See also: Pos Function, StartsWith Function, Replace Function, IsChecked Function

RegExMatch Function

Format

```
i = regexmatch(target_string,regex_string);
```

Description

The `regexmatch` function tests whether the ***target_string*** matches the ***regex_string***.

Return Value

The function returns an integer value of:

1 if ***target_string*** matches ***regex_string.***

0 if ***target_string*** does not match ***regex_string.***

Writing and Testing Regular Expressions

Regular expressions have their own syntax separate from CSPro logic. Therefore, it is recommended to write and test your regular expressions in a regular expression editor. Use any regular expression editor that supports ECMAScript (JavaScript) engine (or flavor). One such editor is regex101, just make sure you change the flavor to ECMAScript.

Example

```
// Matches format xxx-xxx-xxxx
if regexmatch(TELEPHONE_NUMBER, "^([0-9]{3}-){2}[0-9]{4}$") then
    // ...
endif;

// Basic validation for email
if regexmatch(EMAIL, "^[^@\s]+@[^@\s]+\.[^@\s]+$") then
    // ...
endif;
```

Replace Function

Format

```
s = replace(source_text, old_text, new_text);
```

Description

The `replace` function looks at ***source_text*** and replaces one or more instances of the substring ***old_text*** with the value in ***new_text***. All arguments are string expressions and the replacement is case sensitive, meaning that `"c"` is recognized as different from `"C"`.

Return Value

The function returns a new string with the replaced text.

Example 1

```
string example = "Robert Smith's son's name is Robert, Jr.";

example = replace(example, "Robert", "Bob");

// result: Bob Smith's son's name is Bob, Jr.
```

Example 2

```
function string GetFileUri(string local_filename)

    // get the full path in case the filename is relative
    local_filename = path.concat(local_filename);

    // convert all slashes to forward slashes
    local_filename = replace(local_filename, "\", "/");

    // encode the filename as a URI
    exit "file:///" + encode(URI, local_filename);

end;

// ...

string reports_css_uri = GetFileUri("../CSS/Reports.css");
html_file.write("<link rel='stylesheet' type='text/css' href='%s'>", reports_css_uri);
```

See also: Encode Function, Pos Function

StartsWith Function

Format

```
b = startswith(prefix, source);
```

Description

The `startswith` function determines whether the string expression ***source*** begins with the string expression ***prefix***. It is the same as checking if:

```
pos(prefix, source) = 1
```

In addition to standalone use as a function, `startswith` can be used when specifying dictionary access parameters.

Return Value

The function returns a logical value of 1 (true) if the source string starts with the prefix and 0 (false) otherwise.

Examples

```
if startswith("9999", MENU_GEOCODE_SELECTION) then
    errmsg("Are you sure that you want to continue in training mode?")
    select("Yes", continue, "No", reenter);
endif;
```

See also: Pos Function, PosChar Function, IsChecked Function

Strip Function

Format

```
s = strip(string_expression);
```

Description

The `strip` function removes trailing blanks from the ***string_expression***. The result of the function is often used as an argument to other functions such as the `length` and `concat` functions.

You will generally not have to use the `strip` function when working with variable-length strings since they do not have spaces automatically added to the end of the text to match a fixed width.

Return Value

The function returns a string with no trailing blanks.

Example

```
PROC GLOBAL

alpha (30) firstName, lastName, fullName;

PROC EXAMPLE

    firstName = "John";
    lastName = "Henry";

    fullName = concat(strip(firstName), " ", strip(lastName));
    numeric nameLength = length(strip(fullName));

    // fullName is "John Henry "
    // nameLength is 10
```

See also: Alpha Statement, Concat Function, Length Function

ToLower Function

Format

```
s = tolower(string_expression);
```

Description

The `tolower` function scans the given ***string_expression*** and converts any uppercase letters to lowercase letters.

Return Value

The function returns a string with all uppercase letters converted to lowercase letters.

Example

```
// each function call will return "hello james!"
X = tolower("hello james!");
Y = tolower("Hello JaMeS!");
Z = tolower("HELLO JAMES!");
```

See also: ToUpper Function

ToNumber Function

Format

```
d = tonumber(string_expression);
```

Description

The `tonumber` function converts a ***string_expression*** into a number. Leading blanks in the string are ignored. The conversion stops at the first non-numeric character encountered.

Return Value

The function returns a decimal number. If the string begins with a non-numeric character (other than a blank), the function returns `default`.

Example

```
numeric districtParameter = tonumber(sysparm("DISTRICT"));
```

See also: Edit Function

ToUpper Function

Format

```
s = toupper(string_expression);
```

Description

The `toupper` function scans the given ***string_expression*** and converts any lowercase letters to uppercase letters.

Return Value

The function returns a string with all lowercase letters converted to uppercase letters.

Example

```
// each function call will return "HELLO JAMES!"
X = toupper("hello james!");
Y = toupper("Hello JaMeS!");
Z = toupper("HELLO JAMES!");
```

See also: ToLower Function

Multiple Occurrence Functions

Average Function

Format

```
d = average(multiple_item [where condition]);
```

Description

The `average` function returns the average of ***multiple_item***, an item that occurs multiple times. During data entry, the result of the average calculation depends on where the statement is located. If the function is executed prior to the form or roster containing the item, it returns `default`. If it is executed within the form or roster containing the item, it returns the average up to the current occurrence number. If it is executed after the form or roster containing the item, it returns the average for all occurrences of the item.

During batch editing, the function returns the average value for all occurrences of the item, regardless of the statement's placement in the program.

If a `where` condition is included, the function returns the average of the occurrences for which ***condition*** is true.

If the value of an occurrence of the item is a special value, the occurrence will not be included in the calculation. If none of the occurrences have values other than special values, `default` is returned.

Return Value

The function returns the decimal value of the average.

Examples

```
AVG_INCOME = average(INCOME);

AVG_FEMALE_INCOME = average(INCOME where SEX = 2);
```

Count Function

Format

```
i = count(multiple_item[ where = condition]);
```

Description

The `count` function returns the number of occurrences for a repeating form or roster. During data entry, the occurrence value is updated after execution of the postproc for the first field within the repeating form or roster. Note the situations impacting this value:

- If the count function is executed prior to the form or roster, it returns 0.
- If it is executed from a field within the form or roster, it returns the current occurrence number.
- If it is executed after the form or roster, it returns the total number of occurrences in the form or roster.

During batch editing, count always returns the total number of occurrences in the multiply-repeating item/record.

If a **where** condition is included, the function returns the number of occurrences for which the condition is true. If the **where** condition is not included, the count function and the `noccurs` function return the same result.

Return Value

The function returns an integer count value.

Examples

```
TOTAL_PERSONS = count(PERSONS);

NUM_CHILDREN = count(PERSONS where RELATIONSHIP = 3);
```

See also: NOccurs Function, SOccurs Function, TotOcc Function, CurOcc Function, Seek Function, Has Operator

CurOcc Function

Format:

```
i = curocc([group]);
```

Description:

The **curocc** function returns the current occurrence number for a roster, form, or record.
During data entry, you may determine the current occurrence of a roster or form. If the form does not repeat, **curocc** will return 1 (a roster must always repeat). The current occurrence can be determined by calling the **curocc** function from any field contained within the roster or form. If it is executed prior to the roster or repeating form it names, it returns 0. If it is invoked after entry of the roster or form has completed, it returns the total number of occurrences keyed.

During batch editing, you may determine the current occurrence of a record or repeating item. If the **curocc** function is used in a procedure not associated with an item on a record then **curocc** will return the total number of occurrences found. If the **curocc** function is used in a procedure associated with an item on the record, it will return the sequence number of the record in the case. The **curocc** of a repeating item will be its sequence number within the group.

Return value:

The function returns the occurrence number as an integer.

Examples 1:

```
PROC RELATION
  if curocc(PERSON_REC) = 1 then
    if (RELATION <> 1) then
      errmsg("First person must be head of household.");
    endif;
  endif;
```

See also: Maxocc Function, Totocc Function, Noccurs Function, Soccurs Function, Count Function , If Statement, Errmsg Function

Delete Function

Format

```
b = delete(group_name[(occurrence)]);

b = delete(group_name, first_occurrence)[, last_occurrence]);
```

Description

The `delete` function removes incomplete, or otherwise unneeded, records or item occurrences from the current case. It can be used to remove singly- or multiply-occurring records, although if the record is non-repeating (such as the housing record in a typical census application), then it must be defined as "Required=No" in the dictionary. This function is primarily intended for batch applications. It should be used with extreme caution in data entry applications because of possible conflicts between the operator's actions and the program logic.

The first syntax deletes a single occurrence of the record. When deleting a singly-occurring record, it is not necessary to specify the ***occurrence***. The second syntax deletes one or more occurrences of a record, starting with the ***last_occurrence*** and deleting up to the ***first_occurrence***. This allows for the mass deletion of record occurrences.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example 1 (Multiply-Occurring Records)

```
// remove "blank" person records
do numeric i = totocc(PERSON_REC) while i >= 1 by -1

    if RELATIONSHIP(i) = notappl and SEX(i) = notappl and AGE(i) = notappl then
        delete(PERSON_REC(i));
    endif;

enddo;
```

In this example blank person records are deleted from the case. Records following any deleted record are "shifted up" to cover the vacated area. For example, if you delete the second of four records, the third record shifts to the second position and the fourth record shifts to the third position.

It is best to delete the records starting with the last record and moving toward the first. Use a subscript that starts at the last occurrence then is decremented (decreased by 1). In this way you will not need to worry about the records that are shifting positions.

Example 2 (Singly-Occurring Records)

```
// if person is homeless, delete the record
if H01_TYPE = 6 then
    delete(HOUSING_REC); // note the absence of a subscript
endif;
```

See also: Insert Function

GetOccLabel Function

Format

```
s = getocclabel(item_name || group_name);
```

Description

The `getocclabel` function returns the occurrence label for a repeating item or group (i.e., the row label of a roster). If no ***item_name*** or ***group_name*** is given, the function returns the occurrence label of the current field, as calculated as follows:

```
s = getocclabel($(curocc($)));
```

In addition to uses in logic, the function can be used as a fill in question text.

Return Value

The function returns a string containing the occurrence label. If an occurrence label for the item or group does not exist, the function returns a blank string.

Example

```
if EXPENSES > SALES then
    errmsg("You could not have made a profit on %s if expenses were greater than
sales.",getocclabel());
    reenter;
endif;
```

See also: SetOccLabel Function, Occurrence Labels, Create Fills In Questions

Insert Function

Format

```
b = insert(group_name[(occurrence)]);

b = insert(group_name, first_occurrence)[, last_occurrence]);
```

Description

The `insert` function inserts missing or otherwise needed data records or item occurrences in the current case. It is primarily intended for use in batch applications. It should be used with extreme caution in data entry applications because of possible conflicts between the operator's actions and the program logic.

The first syntax inserts a single occurrence of the record. When inserting a singly-occurring record, it is not necessary to specify the ***occurrence***. The second syntax inserts one or more occurrences of a record, starting with the ***first_occurrence*** and inserting up to the ***last_occurrence***. This allows for the mass inserts of record occurrences.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example 1 (Multiply-Occurring Records)

In the following example there is a data item in the housing record called `H13_PERSONS`, which contains the total number of people living in the household. We have decided that if the number of population records found in the household is less than this variable, we will insert the missing number of population record(s).

```
do numeric i = count(PERSON_REC) + 1 while i <= H13_PERSONS
  insert(PERSON_REC(i));
enddo;
```

It makes no difference if the population record has been defined in the dictionary as required or not. What is important is that it was defined as a multiply-occurring record.

Example 2 (Singly-Occurring Records)

For this example, we are processing data that did not require housing records to be present. However, now we want to force the existence of housing records. We could implement this as follows:

```
if totocc(HOUSING_REC) = 0 then
    insert(HOUSING_REC); // note the absence of a subscript
endif;
```

To accomplish this, the housing record must be set to "Required=No" in the dictionary. You cannot use this function for a singly-occurring record when the record is required.

See also: Delete Function

Max Function

Format:

```
d = max(multiple-item [where condition]);
```

Description:

The **max** function returns the maximum value of an item that occurs multiple times. During batch editing, if the values of the items are not changed, the result of the maximum calculation is the same, no matter where the function is located.

During data entry, the result of the maximum calculation depends on where the statement is located. If the **max** function is executed prior to the form or roster containing the item, it returns DEFAULT. If it is executed within the form or roster containing the item, it returns the maximum value up to the current occurrence number. If it is executed after the form or roster containing the item, it returns the maximum value for all occurrences of the item.

During batch editing, max always returns the maximum value for all occurrences of the item.

If a **where** condition is included, the function returns the maximum value of the occurrences for which the condition is true.

If the value of an occurrence of the item is a special value (DEFAULT, MISSING, REFUSED, or NOTAPPL), the occurrence will not be included in the calculation. If none of the occurrences have values other than special values, DEFAULT is returned.

Return value:

The function returns a decimal value.

Examples:

```
MAX_INCOME = max(INCOME);
MAX_FEMALE_INCOME = max(INCOME where SEX = 2);
```

See also: Min Function

MaxOcc Function

Format:

```
i = maxocc([group]);
```

Description:

The **maxocc** function returns the maximum number of multiply occurring records or the maximum number of multiply-occurring items defined for a group in the dictionary.

This value remains the same throughout the application run.

Return value:

The function returns an integer value of the maximum number of occurrences.

Example 1:

```
PROC HOUSING
  errmsg("Maximum number of persons is %d", maxocc(PERSON));
```

Example 2:

```
PROC PERSON
  errmsg("Maximum number of persons is %d", maxocc());
```

See also: Curocc Function, Totocc Function, Soccurs Function, Noccurs Function

Min Function

Format:

```
d = min(multiple-item [where condition]);
```

Description:

The **min** function returns the minimum value of an item that occurs multiple times. During batch editing, if the values of the items are not changed, the result of the minimum calculation is the same, no matter where the function is located.

During data entry, the result of the minimum calculation depends on where the statement is located. If the **min** function is executed prior to the form or roster containing the item, it returns DEFAULT. If it is executed within the form or roster containing the item, it returns the minimum value up to the current occurrence number. If it is executed after the form or roster containing the item, it returns the minimum value for all occurrences of the item.

During batch editing, **min** always returns the minimum value for all occurrences of the item.

If a **where** condition is included, the function returns the minimum value of the occurrences for which the condition is true.

If the value of an occurrence of the item is a special value (DEFAULT, MISSING, REFUSED, or NOTAPPL) the occurrence will not be included in the calculation. If none of the occurrences have values other than special values, DEFAULT is returned.

Return value:

The function return a decimal value.

Examples:

```
MIN_INCOME = min(INCOME);
MIN_MALE_INCOME = min(INCOME where SEX = 1);
```

See also: Max Function

NOccurs Function

Format:

```
i = noccurs(group);
```

Description:

The **noccurs** function returns the number of occurrences of a roster, form, or record. It is equivalent to the count function without the **where** phrase.

During data entry, you may determine the current occurrence of a roster or form. The occurrence value is updated after the first entry into the first field. If the **noccurs** function is executed prior to the roster or form it specifies then it returns 0. If it is executed from a field within the roster or form, it returns the current occurrence number after the first field is on the path. For example, its value in the PREPROC of the first occurrence of the first item entered in a form or roster is zero (0), i.e., before entry. After entry its value will always be one (1). This is true for each occurrence keyed, the **noccurs** function is not incremented until the cursor is in or has passed through the first field on the roster or form. If it is executed after the form or roster, it returns the total number of occurrences in the form or roster. If the form does not repeat, **noccurs** will return 1 (a roster must always repeat). When used in Data entry **noccurs** and soccurs functions are the same.

During batch editing, **noccurs** always returns the total number of occurrences in the group.

Return value:

The function returns the number of occurrences as an integer value.

Example:

```
TOTAL_PERSONS = noccurs(PERSON);
```

See also: Totocc Function, Curocc Function, Soccurs Function, Count Function

Seek Function

Format:

```
i = seek(multiple-item condition[,numeric-expr]);
```

Description:

The **seek** function returns the occurrence number of the first item in a multiply occurring item that satisfies a certain condition. If *numeric-expr* is included, the function starts searching the multiply occurring record for a true condition starting at occurrence *numeric-expr*. If a @ symbol precedes the *numeric-expr*, the function searches for the nth occurrence of the condition.

Return value:

The function returns the occurrence number of an item that meets the condition or 0 if no such item is found.

Example 1:

```
numeric femaleIndex = seek(SEX = 2);

while femaleIndex do
    write("Person #%d is a female with name '%s'",femaleIndex,NAME(femaleIndex));
    femaleIndex = seek(SEX = 2,femaleIndex + 1);
enddo;
```

Example 2:

```
numeric secondSpouse = seek(RELATIONSHIP = 2,@2);
```

See also: Count Function, Has Operator, Seekmax Function, Seekmin Function

SeekMax Function

Format:

```
i = seekmax(multiple-item condition[,numeric-expr]);
```

Description:

The **seekmax** function returns the occurrence number of the item with the greatest value that satisfies a certain condition in a multiply occurring item. If *numeric-expr* is included, the function starts searching the multiply occurring record for a true condition starting at occurrence *numeric-expr*.

Return value:

The function returns the occurrence number of an item that meets the condition or 0 if no such item is found.

Example

```
// seekmax
numeric eldest_index = seekmax(AGE);
numeric eldest_aged_12to64_index = seekmax(AGE in 12:64);

// seekmin
numeric youngest_aged_65plus_index = seekmin(AGE in 65:95);
```

See also: Count Function, Has Operator, Seek Function, SeekMin Function, Max Function

SeekMin Function

Format:

```
i = seekmin(multiple-item condition[,numeric-expr]);
```

Description:

The **seekmin** function returns the occurrence number of the item with the lowest value that satisfies a certain condition in a multiply occurring item. If *numeric-expr* is included, the function starts searching the multiply occurring record for a true condition starting at occurrence *numeric-expr*.

Return value:

The function returns the occurrence number of an item that meets the condition or 0 if no such item is found.

Example

```
// seekmax
numeric eldest_index = seekmax(AGE);
numeric eldest_aged_12to64_index = seekmax(AGE in 12:64);

// seekmin
numeric youngest_aged_65plus_index = seekmin(AGE in 65:95);
```

See also: Count Function, Has Operator, Seek Function, SeekMax Function, Min Function

SetOccLabel Function

Format

```
b = setocclabel(item_name || group_name,occurrence_label);
```

Description

The `setocclabel` function sets the occurrence label for a repeating item or group (i.e., the row label of a roster). This is especially useful when designing applications for mobile devices, as it allows you to specify the label that is displayed in the navigational case tree. If no ***item_name*** or ***group_name*** is given, the function sets the occurrence label of the current field. The occurrence label is set to the string expression ***occurrence_label***.

Return Value

The function returns a logical value of 1 (true) if the label was set successfully and 0 (false) if the occurrence for the item or group item does not exist.

Example 1

```
PROC PERSON_NAME

    setocclabel(PERSON_ROSTER,strip(PERSON_NAME));
```

Example 2

```
PROC MYSURVEY_QUEST

preproc

    // fill in occurrence labels for the household roster
    do numeric ctr = 1 while ctr <= totocc(PERSON_ROSTER)
        setocclabel(PERSON_ROSTER(ctr),strip(NAME(ctr)));
    enddo;
```

See also: GetOccLabel Function, Occurrence Labels, ShowOcc Function

SOccurs Function

Format:

```
i = soccurs(record-name);
```

Description:

The **soccurs** function returns the total number of occurrences of a record.

During data entry, you may determine the current occurrence of a record. The occurrence value is updated after the first entry into the first field. If the **soccurs** function is executed prior to the roster or form it specifies then it returns 0. If it is executed from a field within the roster or form, it returns the current occurrence number after the first field is on the path. For example, its value in the PREPROC of the first occurrence of the first item entered in a form or roster is zero (0), i.e., before entry. After entry its value will always be one (1). This is true for each occurrence keyed, the **soccurs** function is not incremented until the cursor is in or has passed through the first field on the record. If it is executed after the form or roster, it returns the total number of occurrences in the form or roster. If the form does not repeat, noccurs will return 1 (a roster must always repeat). When used in Data entry noccurs and **soccurs** functions are the same.

During batch editing, **soccurs** always returns the total number of record occurrences found.

Return value:

The function returns the number of occurrences as an integer value.

Example:

```
NUM_HH_MEMBERS = soccurs(PERSON_REC);
```

See also: Totocc Function, Curocc Function, Noccurs Function, Count Function

Sort Function

Format

```
b = sort(group_name using [-]item_name1 [,[-]item_name2, ...] [where where_condition]);
```

Description

The `sort` function sorts occurrences of records or items based on the value of an item or multiple items. It orders the multiple records or items in the specified group in ascending sequence using the specified data item as the sort key. The sort key item must be contained within the record or item sorted. If a negative sign is included before the item name, the sort will be in descending order. If a ***where_condition*** is included, the function sorts only the occurrences for which the condition is true. When multiple sort keys are provided, the second, third, ... key is only evaluated when the values of the previous keys are identical.

Sort is primarily intended for use in batch applications. It should be used with extreme caution in data entry applications because of possible conflicts between the operator's actions and the program logic.

Return Value

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Examples

```
// sort the people using the line number
sort(PERSON_EDT using LINE_NUM);

// sort the children by age from oldest to youngest
sort(PERSON_EDT using -AGE where RELATIONSHIP = 3);

// sort the people first by relationship code (ascending) and then by age (descending)
sort(PERSON_EDT using RELATIONSHIP, -AGE);
```

See also: Swap Function, ValueSet.Sort Function

Sum Function

Format

```
d = sum(multiple_item [where condition]);
```

Description

The sum function returns the sum of an item that occurs multiple times. If a where ***condition*** is included, the function returns the sum of the occurrences for which the condition is true.

During data entry, the result of the sum calculation depends on where the statement is located. If the sum function is executed prior to the form or roster containing the item, it returns `default`. If it is executed within the form or roster containing the item, it returns the sum up to the current occurrence number. If it is executed after the form or roster containing the item, it returns the sum for all occurrences of the item.

During batch editing, sum always returns the sum for all occurrences of the item.

If the value of an occurrence of the item is a special value (`default`, `missing`, `refused`, or `notappl`) the occurrence will not be included in the calculation. If none of the occurrences have values other than special values, `default` is returned.

Return Value

The function returns a decimal value of the sum.

Example

```
TOTAL_INCOME = sum(INCOME);
TOTAL_FEMALE_INCOME = sum(INCOME where SEX = 2);
```

Swap Function

Format:

```
b = swap(group-name,record-number,record-number);
```

Description:

The **swap** function reorders the sequence of occurrences of records or items. It is useful for reorganizing the position (occurrence number) of items in a roster, for instance if you want to ensure that the head of household is the first person on the roster.

Swap is primarily intended for use in batch applications. It should be used with extreme caution in data entry applications because of possible conflicts between the operator's actions and the program logic.

Return value:

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Example:

```
swap(PERSON,1,ptrHead);
```

See also: Sort Function

TotOcc Function

Format:

```
i = totocc([group]);
```

Description:

The **totocc** function returns the total number of multiply occurring records or the total number of multiply-occurring items that a group currently contains.

During data entry, the occurrence value is updated in the preproc of the first field within a repeating form, record or roster. If the **totocc** function is executed prior to the entry of form or roster, it returns 0. If it is executed from a group or field within the form or roster, it returns the total number of occurrences currently entered. If it is executed after the form or roster, it returns the total number of occurrences in the form or roster.

During batch editing, **totocc** always returns the total number of occurrences in the group.

Return value:

The function returns an integer value of the number of occurrences.

Example 1:

```
if totocc(HOUSING) > 1 then
  errmsg("More than 1 housing record");
endif;
```

Example 2:

```
PROC HOUSING
if totocc() > 1 then
  errmsg("More than 1 housing record");
endif;
```

See also: Curocc Function, Maxocc Function, Count Function, Soccurs Function, Noccurs Function

General Functions

Compress Function

Format

```
i = compress(zip_file_name, input_file_list || input_file_name);
```

Description

The `compress` function compresses one or more files and saves the compressed files in a single .zip file. That file can be decompressed with the `decompress` function or with other popular tools. Compressing data, in addition to saving size, is a convenient way to bundle several files into a single file.

The argument ***zip_file_name*** is a string expression containing the file name of the output .zip file. If an existing .zip file exists with that name, it will be overwritten. The list of input files can come from a string list, ***input_file_list***, that could be returned from a `dirlist` function call. Alternatively, the list of input files can come from a string expression ***input_file_name***. If using a string expression, you can use the wildcard characters "*" and "?" to specify a group of files.

Return Value

The function returns the number of files (but not folders) added to the compressed .zip file. If there was an error creating the compressed file, the function returns `default`.

Example 1

```
compress("CollectedData.zip", "*.csdb");
```

Example 2

```
list string imageFilesListing;

dirlist(imageFilesListing, pathname(InputFile), "*.jpg", recursive);

compress("Images.zip", imageFilesListing);
```

See also: Decompress Function

Decompress Function

Format

```
i = decompress(zip_file_name[,directory_name]);
```

Description

The `decompress` function decompresses the compressed files contained in a .zip file. Compressed .zip files can be created using the `compress` function or with other popular tools.

The argument ***zip_file_name*** is a string expression containing the file name of the .zip file. An optional string expression ***directory_name*** indicates the folder to which the files should be decompressed. If no directory name is specified, the files are decompressed to the current application directory.

Return Value

The function returns the number of files (but not folders) decompressed from the .zip file. If there was an error decompressing files, the function returns `default`.

Example

```
decompress("Images.zip",pathname(InputFile) + "Images");
```

See also: Compress Function

Diagnostics Function

Format

```
s = diagnostics([property][, argument, ...]);
```

Description

The `diagnostics` function is used to get information about the version of CSPro you are using, as well as other functionality. It is used mostly for troubleshooting.

The function takes a string expression ***property*** and then optional string arguments that determine what it returns. If called without a property, the function will return all of the no-argument diagnostic properties. The properties are:

Property	Argument	Diagnostic Value
version		The currently running CSPro version (7.7).
version_detailed		The currently running CSPro version (7.7.2).
releasedate		The date that this version of CSPro was released (in YYYYMMDD format).
beta		Whether the current CSPro version is a beta version (1 if true, 0 if false).
serializer		The version of a .pen file that would be created with the current version of CSPro.
md5	File name	The MD5 hash value of the file.

Return Value

The function returns a string containing the diagnostic information. If the property is invalid, then the function returns a blank string.

Example

```
PROC MATERNITY_QUEST

preproc

    if diagnostics("version") <> "7.7" then
        errmsg("This application has not been tested with this version of CSPro. Things may
behave differently than expected.");
    endif;
```

Encode Function

Format

```
s = encode([encoding_type,] text);
```

Description

The encode function escapes special characters to facilitate writing to HTML or CSV files, or creates properly encoded strings for a variety of formats.

The optional argument ***encoding_type*** specifies the format for encoding. The ***text*** argument is a string expression that specifies the text to encode. If an encoding type is provided with no text, then the default encoding type for the application is modified. An application starts with the default encoding type set to HTML.

The encoding types are:

Encoding Type	Description
HTML	Encodes the characters < > & and newlines for writing to HTML files.
CSV	Encodes the characters , " and newlines for writing to comma-separated values files.
URI	Encodes a group of characters for writing uniform resource identifiers (URIs).
URIComponent	Encodes a group of characters for writing the components of URIs.
PercentEncoding	Encodes a group of characters for writing to formats such as connection strings.
JsonString	Encodes text for writing as a valid JSON string.
Slashes	Adds backslashes to escape quote and backslash characters.

Return Value

The function returns a string with the appropriate characters encoded. If the default encoding type is changed, the function returns a blank string.

Example

```
// write to a HTML file
file html_file;

html_file.write("<p>Enumerator name: %s</p>", encode(strip(ENUMERATOR_NAME)));

string house_image_filename = pathconcat("../House Images", key(HOUSING_DICT) + ".jpg");
html_file.write("<p><img src='file:///%s' alt='House' />
</p>", encode(URI, house_image_filename));


// write to a CSV file
file csv_file;

encode(CSV);
csv_file.write("Address,Head of Household");
csv_file.write("%s,%s", encode(strip(ADDRESS)), encode(strip(HEAD_NAME)));
```

See also: File.Write Function, Replace Function

ErrMsg Function

Format 1

```
i = errmsg(string_exp⌡, argument1, ..., argument20⌡)

          ⌡select(caption1, fieldname1⌡, ..., caption5, fieldname5⌡)⌡
          ⌡case|summary⌡
          ⌡denom=numeric_denom⌡;
```

Format 2

```
i = errmsg(msg_num⌠, argument1, ..., argument20⌡)

          ⌠select(caption1, fieldname1⌠, ..., caption5, fieldname5⌡)⌡
          ⌠case|summary⌡
          ⌠denom=numeric_denom⌡;
```

Format 3

```
i = errmsg(string_exp⌠, argument1, ..., argument20⌡)

    ⌠select(caption1, fieldname1⌠, ..., caption5, fieldname5⌡) ⌠default(default_option)⌡ ⌡
    ⌠case ‖ summary⌡
    ⌠denom = numeric_denom⌡;
```

Description

The errmsg function displays a message on the data entry screen (when used in a Data Entry application) or writes a message to the output listing file (when used in a Batch Edit application). If messages are defined via the message number ***msg_num***, then those messages must be stored in a message file [.mgf]. ***msg_num*** can be a number or numeric expression.

Each ***argument*** is sequentially inserted into the error message. Arguments can be numeric or alphanumeric expressions, but the type of the argument must match the type of the receiving field in the message text. The maximum number of arguments is 20.

Some of the most common formatting options include (see Message Formatting Options for a complete list):

%**[n]d** = Insert a number and display it as an integer
%**[n.d]f** = Insert a number and display it as a decimal value
%**[n.d]s** = Insert a text string

where ***"n"*** is the size of the field and ***"d"*** is the number of decimal places to show for a number.

Numbers are never truncated. Text strings are truncated only if ***".d"*** is used.

If ***"n"*** is positive, the insert is right-justified in the size of the field. If "n" is negative, the insert is left-justified in the size of the field. If ***"n"*** is a positive number with a leading zero, the insert is right-justified in the size of the field and zero filled to the left.

When inserting a number, if ***"n"*** is preceded by a +, the sign of the number is always displayed.

```
numeric integerValue = 23456;

errmsg( "%d"       , integerValue);      //  23456
errmsg( "%-10d"    , integerValue);      //  23456
errmsg( "%10d"     , integerValue);      //       23456
errmsg( "%+10d"    , integerValue);      //      +23456
errmsg( "%+010d"   , integerValue);      //  +000023456
errmsg( "%-010d"   , integerValue);      //  0000023456
errmsg( "%f"       , integerValue);      //  23456.000000  Note the usage of %f


numeric decimalValue = 12.567;

errmsg( "%f"        , decimalValue);    //   12.567000
errmsg( "%-10.3f"   , decimalValue);    //   12.567
errmsg( "%d"        , decimalValue);    //   12           Note the usage of %d
errmsg( "%10.2f"    , decimalValue);    //        12.57
errmsg( "%10.3f"    , decimalValue);    //       12.567
errmsg( "%+10.3f"   , decimalValue);    //      +12.567
errmsg( "%+010.3f"  , decimalValue);    //   +00012.567
errmsg( "%010.3f"   , decimalValue);    //   000012.567
errmsg( "%10.5f"    , decimalValue);    //     12.56700


string stringValue = "abcdef";

errmsg( "%s"        , stringValue);     //   abcdef
errmsg( "%-10s"     , stringValue);     //   abcdef
errmsg( "%-10.3s"   , stringValue);     //   abc
errmsg( "%10s"      , stringValue);     //       abcdef
errmsg( "%10.3s"    , stringValue);     //          abc
```

The `case` and `summary` keywords give you some control over the output listing. By default, the output listing shows you messages case-by-case, and summarized by the number of times the message was triggered for the run (there is an optional denominator, described below). You can limit the output listing to only case-by-case reporting, or only summary reporting, by using these keywords.

The `denom` keyword allows you to specify a denominator, so that you can show percentages in the summary portion of the output listing. This is very useful for showing, for example, edit failure rates (and as such, this option is geared towards batch applications rather than data entry applications). In Format 1's Example 2 below, the output listing will show the number of times the head of household had more than one spouse, divided by the number of households processed during the run. Note that it is the responsibility of the application designer to write logic to put the proper values into the denominator variable.

The **select** keyword is used to give the option of specifying the field to go to in response to the error message. A button is placed on the error message window for each caption and fieldname pair specified as a **select** argument, with a maximum of 5 possible **select** argument pairs allowed. The caption will be displayed on the button. When a button is chosen, the system goes to the field specified by fieldname. If `continue` is used as the fieldname, the system proceeds to the next line of logic following the `errmsg` call. If a ***default_option*** is given, it will place the focus on the select option chosen when the message is presented. This is a good method to prioritize what option you want the user to make (reenter a value vs. continue for example). If a ***default_option*** is not given, no button will have the focus.

Return Value

The function returns the sequence number of the button pressed. When the **select** keyword is not used, the function will return 1, as only the "OK" button is presented to the user, corresponding to button #1 ("Cancel" is never presented, and so a code of zero is not possible).

When the **select** keyword is used, the function returns the sequence number of the button that was pressed, with the first button having the value 1. A maximum of 5 select argument pairs is allowed.

Format 1 Examples

Example 1

```
errmsg("Head of household is %d years old.", AGE);
```

Example 2

```
if count(PERSONS where RELATIONSHIP=2) > 1 then
    errmsg("Head of HH has more than 1 spouse") denom = HH_COUNT summary;
endif;
```

Example 3

```
errmsg("Head of household is %d years old.  Age must be >= 12", AGE)
      select("Go to RELATIONSHIP", RELATIONSHIP,"Go To AGE", AGE)
      denom = PERSON_COUNT;
```

Format 2 Examples

```
OK = errmsg (1,"June",30,31);
```

where the message file contains the following entry:

```
1 %s has only %d days. You entered %d!
```

Note the `errmsg` call could have also been invoked as follows:

```
i = 1;
OK = errmsg (i,"June",30,31);
```

See also: Warning Function, Message Formatting Options, Message File (.mgf)

ExecSystem Function (Desktop)

Format

```
b = execsystem(action[, flags]);
```

Description

The `execsystem` function, on a desktop, starts another Windows application or process. There is a related version for applications running on mobile devices.

The string expression ***action*** is the name of the application or process to be started. Command line parameters may be included in this expression. If folder or file names contain blanks, then quotation marks (") must surround the names. In this case CSPro text strings can be surrounded by apostrophes (').

An optional set of three ***flags*** controls the way that the application is opened, and includes the following options:

Category	Flag	Description
Program Control	**nowait**	The application is opened alongside CSPro. This is the default option.
	wait	The application is opened and CSPro waits until it closes to resume program control.
	stop	CSPro immediately closes and then opens the application. This is similar to what would occur if using the `stop` function with an argument of 1.
Window Size	**normal**	The application's window is opened in its normal state. This is the default option.
	maximized	The application's window is opened in its maximized state.
	minimized	The application's window is opened in its minimized state.

Window Focus	**focus**	The application is opened and immediately has focus (is "active"). This is the default option.
	nofocus	The application is opened without focus.

A combination of flags can be used, with each set separated by a comma.

Return Value

The function returns a logical value of 1 (true) if the new application is started successfully and 0 (false) otherwise.

Examples

```
// opens the Windows calculator
execsystem("calc.exe");

// opens documentation in Text Viewer
execsystem(maketext('"%sTextView.exe"
"C:\Survey\Helps.txt"', pathname(CSPro)), maximized, wait);

// open a website in Internet Explorer
execsystem(maketext('"%sInternet Explorer\iexplore.exe"
"https://www.csprousers.org/forum"', pathname(ProgramFiles32)));
```

See also: SystemApp Object, ExecSystem Function (Mobile), ExecPFF Function, View Function

ExecSystem Function (Mobile)

Format

```
b = execsystem(action⌈, stop⌋);
```

Description

The `execsystem` function, on mobile devices (Android), starts another application or executes a task. There is a related version for applications running on desktop CSEntry.

If `stop` is coded, the application will immediately close after launching the application, in the same way that would occur if using the `stop` function with an argument of 1.

The string expression ***action*** must begin with one of the following commands:

Command	Description
app	Open another application on the device. (You can get more control using SystemApp instead.)
browse	View a website in the Internet browser.
call	Place a phone call.
camera	Take a photo in .jpg or .png format.
gps	View a latitude/longitude point using an installed mapping engine (e.g., Google Maps).
html	View a website from within CSEntry (equivalent to using the `view` function).
signature	Bring up a screen where someone can sign the mobile device, an image of which can be saved in .jpg or .png format.
sms	Send a text message.
view	Display an image, play a sound clip, or view a movie (equivalent to using the `view` function).

Return Value

The function returns a logical value of 1 (true) if the new application or task is started successfully and 0 (false)

otherwise.

Examples

```
execsystem("app:com.google.android.gm"); // open Gmail

execsystem("browse:https://www.census.gov");

execsystem("call:+13017631451"); // call the CSPro support number

execsystem("camera:" + pathconcat(Application, "photo.jpg"));
execsystem("camera:" + pathconcat(Application, "photo.png"));

execsystem("gps:38.84839,-76.931098");
execsystem("gps:38.84839,-76.931098,CSPro Team at the U.S. Census Bureau");

execsystem("html:https://www.census.gov");
execsystem("html:file:///" + pathconcat(Application, "local_webpage.html"));

execsystem("signature:" + pathconcat(Application, "saved-signature.jpg"));
execsystem("signature:" + pathconcat(Application, "saved-signature.png"));

execsystem("sms:+13017631451");
execsystem("sms:+13017631451,Hello, CSPro Team!");

execsystem("view:" + pathconcat(Application, "picture.jpg"));
execsystem("view:" + pathconcat(Application, "audio.mp3"));
execsystem("view:" + pathconcat(Application, "movie.3gp"));
```

See also: SystemApp Object, ExecSystem Function (Desktop), ExecPFF Function, Multimedia Features, View Function

ExecPFF Function

Format

```
b = execpff(pff_filename || pff_name[, flags]);
```

Description

The `execpff` function starts another CSPro application. This function is useful when designing menu programs or when conducting surveys that use multiple machines on which the path of the CSPro executables may differ. The function is very similar to `execsystem`, but instead of passing the name of an application or task, you supply a string expression, ***pff_filename***, containing the name of a CSPro application's Program Information File (.pff), or ***pff_name***, the name of a pff object. If using a modified but unsaved pff object, it will be saved to a temporary file and then executed.

An optional set of ***flags***, described in the `execsystem` helps, allow you to control how the CSPro application is opened.

Tools Running in the Same Process

Typically the `execpff` function opens a CSPro application as a new program (meaning, for example, that two processes will be running: the one that called `execpff` and one for the launched program). However, some tools can run in the same process as the application that called the function. If the `wait` flag is used when calling `execpff`, the following tools will run in the same process: Compare Data, Concatenate Data, Index Data, Paradata Concatenator, Reformat Data, and Sort Data.

Mobile Behavior

On mobile devices, multiple data entry applications cannot be run simultaneously, so the function simply stores the file name of the PFF. When the mobile application returns to the Entry Applications screen, whether via user interaction or via the `stop` function, the passed PFF will be launched. Using `stop` as a flag will cause the PFF to be launched

immediately.

If the PFF file that launched the application that calls this function specified an OnExit parameter, the ***pff_filename*** specified by this function will override the OnExit parameter.

Return Value

The function returns a logical value of 1 (true) if the new application is started successfully and 0 (false) otherwise.

Example

```
PROC MENU

    if MENU = 1 then
        execpff("Household.pff", stop);

    elseif MENU = 2 then
        execpff("Agriculture.pff", stop);

    endif;
```

See also: Pff.Exec Function, ExecSystem Function (Desktop), ExecSystem Function (Mobile)

GetProperty Function

Format

```
s = getproperty([symbol, ]property_name);
```

Description

The `getproperty` function returns a string containing the current value associated with the property specified by the string expression ***property_name***. If an optional dictionary ***symbol***, which can be either an item or a field, is specified, then the property must apply to an item or field. If no symbol is specified, then the property must be a system or application property.

Properties List

Name	Settable?	Values
System Properties		
MaxDisplayWidth		(integer of the maximum display width)
MaxDisplayHeight		(integer of the maximum display height)
Application Properties		
AppType		DataEntry, Batch, Tabulation
AutoAdvanceOnSelection *M	✔	Yes, No
AutoPartialSaveMinutes	✔	(integer of the interval)
CAPI		Yes, No
CaseTree		Always, Desktop, Mobile, Never
CenterForms *D		Yes, No
ComboBoxShowOnlyDiscreteValues	✔	Yes, No
CreateListing		Yes, No
CreateLog		Yes, No
DecimalComma *D		Yes, No

DisplayCodesAlongsideLabels *M	✔	Yes, No
NotesDeleteOtherOperators	✔	Yes, No
NotesEditOtherOperators	✔	Yes, No
OperatorID		Yes, No
ParadataCollection		AllEvents, SomeEvents, No
ParadataDeviceStateMinutes	✔	(integer of the interval)
ParadataGpsLocationMinutes	✔	(integer of the interval)
ParadataRecordCoordinates	✔	Yes, No
ParadataRecordIteratorLoadCases	✔	Yes, No
ParadataRecordValues	✔	Yes, No
PartialSave	✔	Yes, No
Path		Yes (system controlled), No (operator controlled)
ShowEndCaseDialog	✔	Yes, No
ShowErrorMessageNumbers *M	✔	Yes, No
ShowFieldLabels *M		Yes, No
ShowLabelsInCaseTree *M	✔	Yes, No
ShowNavigationControls *M	✔	Yes, No
ShowRefusals	✔	Yes, No
ShowSkippedFields*M	✔	Yes, No
SpecialValuesZero	✔	Yes, No
UpdateSaveArrayFile	✔	Yes, No
ValidateAlphaFields	✔	Yes, No
WindowTitle	✔	(string of the window title)

Item Properties

DataType		Alpha, Numeric
Decimal		(integer)
DecimalChar		Yes, No
Len		(integer)
ZeroFill		Yes, No

Field Properties

AllowMultiLine		Yes, No
AutoIncrement		Yes, No
CanEnterNotAppl	✔	No, Confirm, NoConfirm
CanEnterOutOfRange	✔	No, Confirm, NoConfirm
CapturePosX	✔	(integer of the X position)
CapturePosY	✔	(integer of the Y position)
CaptureType	✔	Barcode, CheckBox, ComboBox, Date, DropDown, NumberPad, RadioButton, Slider, TextBox, ToggleButton
CaptureDateFormat	✔	(string of the date format, e.g., YYYYMMDD)
ForceOutOfRange	✔	Yes, No
HideInCaseTree	✔	Yes, No
Keyboard	✔	(integer of the keyboard ID)
Persistent		Yes, No
Protected	✔	Yes, No
Sequential		Yes, No
ShowExtendedControlTitle*D	✔	Yes, No
SkipTo		(string of the field name)

UpperCase	✔	Yes, No
UseEnterKey	✔	Yes, No
UseUnicodeTextBox		Yes, No
ValidationMethod	✔	Default, Confirm, NoConfirm, Custom
Verify		Yes, No

*D desktop only, *M mobile only

Return Value

The function returns a string with the property information. If the property is invalid, a blank string is returned.

Example

```
PROC APPLICATION_FF

preproc

    if getproperty("OperatorID") = "No" then
        setoperatorid(getusername());
    endif;
```

See also: SetProperty Function

GetLabel Function

Format

```
s = getlabel(dictionary_symbol [, value]);
```

Description

The `getlabel` function returns the label of a ***dictionary_symbol*** or the text associated with a particular value of the symbol as defined in a value set. If the ***value*** argument is not specified, then the dictionary symbol's label is returned. The symbol can be the name of a dictionary, level, record, item, or value set.

The ***value*** argument can only be used if the dictionary symbol is an item or a value set. The ***value*** is either a numeric or string expression, based on the type of the item. If the argument is used, the label associated with the specified value is returned. If the symbol is an item name, then the value labels from the current value set are returned. If the symbol is a value set, then the value labels from that value set are returned. If no label is associated with the value, then an empty string is returned. This behavior can also be achieved with the `getvaluelabel` function.

When using the ***value*** argument, a value's code can be returned rather than the value label by using the optional `by label` (as opposed to the default `by code` behavior).

Return Value

The function returns a string containing the label.

Example

```
PROC EXAMPLE

    write("%s", getlabel(SEX));                      // Sex
    write("Crop Type = %s", getlabel(CROP_VS2, 23)); // Crop Type = Maize

PROC RELATIONSHIP

    write("%s = %s", getlabel($), getlabel($, $));   // Relationship = Head,
                                                     // Relationship = Child, etc.

PROC BY_EXAMPLE

    write("%s", getlabel(SEX by code, 1));           // Male
    write("%s", getlabel(SEX by label, "Male"));     // 1
```

See also: GetValueLabel Function, GetSymbol Function, GetImage Function

GetLanguage Function

Format

```
s = getlanguage();
```

Return Value

The `getlanguage` function returns a string with the name of the language currently being used by the application.

Example

```
if getlanguage() = "EN" then
    errmsg("Hello");
elseif getlanguage() = "FR" then
    errmsg("Bonjour");
endif;
```

See also: SetLanguage Function, OnChangeLanguage Global Function, Multiple Language Applications

GetSymbol Function

Format

```
s = getsymbol();
```

Description

The `getsymbol` function returns the name of the current procedure being executed.

An optional syntax, `getsymbol(savepartial)`, returns the field name and occurrence information (for multiply-occurring items) of the location of the last partial save.

Return Value

The function returns a string with the procedure's name.

Example

```
function OnStop()

    if getsymbol() = "ACTION" then
        stop(1);

    else
        reenter ACTION;

    endif;

end;
```

See also: GetLabel Function, GetRecord Function

GetValueLabel Function

Format

```
s = getvaluelabel(item_name);
```

Description

The `getvaluelabel` function returns the value label associated with the current value of ***item_name***. The label is determined in the following order, with the first valid label returned:

1. The label for the value in the item's current value set.
2. The label for the value in the item's primary value set.
3. The value itself, formatted for displaying.

The function returns a value similar to calling:

```
s = getlabel(ITEM_NAME, ITEM_NAME);
```

In addition to uses in logic, the function can be used as a fill in question text.

Return Value

The function returns a string containing the label.

Example

```
PROC GRADE_ATTENDING

    if GRADE_ATTENDING > 3 and AGE < 15 then
        errmsg("Someone aged under 15 cannot attend %s.", getvaluelabel(GRADE_ATTENDING));
        reenter;
    endif;

    // using the "%l" message formatter, the above error message can also written as:
    errmsg("Someone aged under 15 cannot attend %l.", GRADE_ATTENDING);
```

See also: GetLabel Function, Message Formatting Options

Hash Function

Format

```
s = hash(value [, length[, salt]]);
```

Description

The hash function generates a hash value for the string or numeric expression ***value***. This hash value is a fixed-length representation of the value and can be used for purposes such as storing passwords without storing the actual plaintext password. The hash value of the same text string will always be the same, but it is nontrivial (impossible for CSPro purposes) to work backwards to calculate the original text from the hash value.

The hash value is calculated using a SHA-256 key derivation algorithm. This results in a 32-byte value, which is returned as a hexadecimal string with each byte represented as two characters, thus resulting in a string of a length 64.

Optionally, you can specify a numeric expression ***length***, which specifies the length of the hash value in bytes. The string returned will always be twice the value of ***length***. The maximum value for ***length*** is 500.

You can also specify a string expression, ***salt***, which is an additional input in generating the hash value. You must store this salt value somewhere to be able to use this hash function to perform any checks. More information on hash values, key derivation algorithms, and salt values is readily available online.

Return Value

The function returns a hash value represented as a hexadecimal string.

Example

```
string user_password = prompt("Enter your password", password);

if hash(user_password) <> USER_PASSWORD_HASH then
    errmsg("Invalid password. You cannot access this system.");
    stop(1);
endif;
```

See also: HashMap Object

HtmlDialog Function

The htmldialog function allows for the customization of CSPro dialogs in two ways. First, HTML templates can receive dynamic input data from the CSPro application which can be used as text in the dialogs or to change the behavior of the dialog. Second, user-created HTML templates give the application developer complete control of the appearance and behavior of the dialogs.

Format

```
s = htmldialog(html_filename[, input_data]);

s = htmldialog(html_filename,
               [inputData := input_data,]
               [displayOptions := display_options]);
```

Description

The htmldialog function must be passed a a string expression, ***html_filename***, that specifies the HTML template to be used for the dialog.

In the first version, the optional string expression ***input_data*** argument allows input data to be passed to the HTML template.

In the second version, the optional named argument ***input_data*** allows input data to be passed to the HTML template.

The optional named argument ***display_options*** argument is a JSON string that allows control of the display settings. All possible display options are shown below in a table.

Return Value

The function returns a string. The contents of the string are determined by the HTML template and returned by a JavaScript call to `CSPro.returnData()`.

HTML Templates

CSPro includes HTML templates in the installation folder for the dialogs it uses. User-created HTML templates allow for further customization of the dialogs. When the `htmldialog` function is called, the ***html_filename*** is first searched for in the current application directory. If not found and a HtmlDialogs directory override is specified in the PFF file, that directory is searched. If a match is still not found, then the directory *html/dialogs* in the CSPro installation folder is also searched.

Writing a user-created HTML template can be simplified by first selecting an existing HTML template that is most similar to the planned dialog. Make a copy of this HTML template in the current application directory and extend it to fit the new dialogs requirements.

Input Data

The input data argument is a mechanism to pass data from the CSPro application to the HTML template. The contents of the input data could be a simple text string that represents a single value or a JSON string that represents multiple values. The string contents of the input data are retrieved by calling `CSPro.getInputData()` from the script block of the HTML template.

```
// Simple string
var inputString = CSPro.getInputData();

// JSON string
var inputJSON = JSON.parse(CSPro.getInputData());
```

Display Options

The display options can be set using the optional named argument ***display_options*** in CSPro logic, or can be specifed as part of the HTML dialog by executing a JavaScript call to `CSPro.setDisplayOptions(displayOptions)`.

Name	Platform	Values
width	Both	Width in display units
height	Both	Height in display units
resizable	Windows	True, False
borderColor	Windows	CSS name or hex code (e.g., "black" or "#000000")
titleBarColor	Windows	CSS name or hex code
titleBarHeight	Windows	Height in display units
keyboard	Android	True, False (whether to show an on-screen keyboard)

The default values for *resizeable* and *keyboard* are false. If a width and height are not specified, CSPro will wait a few seconds before showing the dialog as it expectantly waits for a call to `CSPro.setDisplayOptions(displayOptions)`. This behavior is undesirable so it is important to specify the dialog width and height.

Example: Existing HTML Template

```
string input_data = '{ "title": "Enter Some Data!",'
                    '  "multiline": true,'
                    '  "allowEmptyText": false }';

string result = htmldialog("text-input.html", input_data);
```

Example: User-Defined HTML Template

```
string input_data = "Hello world!";
string display_options = maketext('{ "width": "%d",'
                                  '  "height": %d,'
                                  '  "borderColor": "#FF0000" }',
                                  tonumber(getproperty("MaxDisplayWidth")) * 0.8,
                                  tonumber(getproperty("MaxDisplayHeight")) * 0.8);

string result = htmldialog("user-defined-dialog.html",
                           inputData := input_data,
                           displayOptions := display_options);
```

See also: HTML in CSPro, JavaScript Interface

InValueSet Function

Format

```
b = invalueset(item_name⌈, value_set_name⌋);
```

Description

The `invalueset` function determines whether an item's current value is within the permissible ranges of a value set. In addition to specifying the ***item_name***, an optional argument, ***value_set_name***, allows you to specify in which value set to check if the item's value is within the permissible ranges. If no value set is specified, the item's current value set is used.

Return Value

The function returns a logical value of 1 (true) if the item's value is within the value set's ranges and 0 (false) otherwise. If the item has no value set, the function return 1 (true).

Example 1

```
if not invalueset(P08_MARITAL_STATUS) then
    errmsg("Marital status is out of range. Value is %d", P08_MARITAL_STATUS);
endif;
```

Example 2

```
if D02_SEX = 1 and not invalueset(D05_CAUSE_DEATH, D05_CAUSE_DEATH_MALE_VS) then
    errmsg("A man cannot die of %l", D05_CAUSE_DEATH);
    reenter;
endif;
```

See also: In Operator, SetValueSet Function

Invoke Function

Format

```
s = invoke(function_name⌈, argument1, ..., argumentN⌋);

s = invoke(function_name, arguments := arguments_json);
```

Description

The `invoke` function allows you to execute user-defined functions using runtime binding (where the function name is only known at runtime). When compiling an application, the CSPro compiler only knows about functions declared up to the point where a function call is coded, so for more dynamic applications, it can be a challenge to manage function calls when you would like to call a function declared after the point of the call. One workaround is to use function pointers; the `invoke` function provides another way to bypass the CSPro compiler limitation.

The `invoke` function evaluates the string expression ***function_name*** at runtime, not compile-time. In the function's first version, zero, one, or more arguments are provided directly at compile-time. At runtime, the arguments are checked against the function's parameters, and if they are compatible, the function call is executed.

The function's second version, using named arguments, allows arguments to be passed using the string expression ***arguments_json***. This text is a JSON string containing the arguments to the function. In this version, only `numeric`, `string`, and `list` arguments can be provided. List arguments must be provided as JSON arrays.

The JavaScript interface provides versions of these functions that can be run in JavaScript: `CSPro.invoke` and `CSPro.invokeAsync`.

Return Value

The function returns the return value of the user-defined function as a string when successful, and a blank string otherwise. If the function has a numeric return value, the value is converted to a string, which you can convert back to a number using the `tonumber` function.

Example 1

```
function numeric ManipulateValues(numeric value1, numeric value2,
                                  string function_name = "AddValues")

    string result = invoke(function_name, value1, value2);
    exit tonumber(result);

end;

function AddValues(numeric value1, numeric value2)
    exit value1 + value2;
end;

function MultiplyValues(numeric value1, numeric value2)
    exit value1 * value2;
end;

// ...

ManipulateValues(16, 16);                    // 32
ManipulateValues(16, 16, "MultiplyValues"); // 256
ManipulateValues(16, 16, "DivideValues");   // runtime error (DivideValues function does not
exist)
invoke("ManipulateValues", 16, 16);         // "32"
```

Example 2

```
function string CombineText(list string values)

    string result;

    do numeric ctr = 1 while ctr <= values.length()
        result = result + values(ctr) + " ";
    enddo;

    exit strip(result);

end;

// ...

invoke("CombineText", arguments := '{ "values": [ "Hello,", "World!" ] }'); // Hello, World!
```

See also: User-Defined Functions, JavaScript Interface

IsChecked Function

Format

```
b = ischecked(code, check_box_field_name);
```

Description

The `ischecked` function returns whether a code is checked as part of a check box field's selections. The ***check_box_field_name*** is the name of an alphanumeric field that is represented on a form as a check box. The ***code*** is a string expression containing the code to be checked.

The codes of CSPro check boxes are placed at uniformly spaced offsets based on the size of the code. For example, if the check box field has a length of 20 and each code has a length of 2, then the function checks positions 1-2 for the code, then positions 3-4, and so on until positions 19-20.

Return Value

The function returns a logical value of 1 (true) if the code was found (at the correct offset) in the check box field and 0 (false) otherwise.

Example

```
PROC LANGUAGES_SPOKEN_OTHER

preproc

    ask if ischecked("Z", LANGUAGES_SPOKEN);
```

See also: Pos Function, StartsWith Function

LoadSetting Function

Format

```
s = loadsetting(attribute [,default_setting]);
```

Description

The `loadsetting` function, using the string expression ***attribute***, retrieves a setting from an attribute-value pair, a setting

that has been previously saved using the `savesetting` function. An optional string or numeric expression ***default_setting*** can be provided and will be returned if no value has been saved for the given attribute.

The file from which settings are retrieved can be modified by altering the CommonStore attribute of a PFF file. If an attribute does not exist in the specified file but does exist in the global settings database (shared by all CSPro applications), then that global attribute-value pair is used.

Return Value

The function returns a string containing the value associated with the attribute. If no value is associated with the attribute, then the function returns a blank string unless a default setting has been provided, in which case it returns the default setting.

Example

```
PROC GLOBAL

function OnChangeLanguage()
    // save the selected language so that the change is persistent,
    // being available the next time that the program is run
    savesetting("Language", getlanguage());
end;


PROC MY_APPLICATION_FF

preproc

    // set the language at the start of the program, defaulting to English
    setlanguage(loadsetting("Language", "EN"));
```

See also: SaveSetting Function, User and Configuration Settings, CSEntry Settings Modification

LogText Function

Format

```
b = logtext(message | message_number⌈, argument1, ..., argumentN⌋);
```

Description

The `logtext` function writes a user-defined message to the paradata log.

The ***message*** is either a string expression or a numeric message number that contains the text that is written to the file. If the text contains any message formatters, the optional arguments ***argument1*** to ***argumentN*** will be inserted into the text.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise. If no paradata log file is open, then the function returns 0.

Example

```
PROC RELATIONSHIP

    if RELATIONSHIP = 1 then
        logtext("Head of household registered as %s",strip(NAME));
    endif;
```

See also: Paradata, Trace Function, Write Function, Message Formatting Options

MaxValue Function

Format

```
d = maxvalue(item_name || value_set_name);
```

Description

The `maxvalue` function returns the maximum valid value for an item or value set. If a numeric ***item_name*** is provided, the function looks at the current value set for the item, or if no value set exists, looks at the item itself (returning a maximum value based on the item's number of digits). If a numeric ***value_set_name*** is provided, the function returns the maximum range or discrete value in the value set.

Return Value

The function returns the maximum valid value for an item or value set.

Example

```
PROC EDUCATION_CURRENT

    if EDUCATION_CURRENT = 1 and AGE > maxvalue(EDUCATION_CURRENT_PRESCHOOL_VS) then
        errmsg("Someone aged %d cannot be currently attending preschool.", AGE);
        reenter;
    endif;
```

See also: MinValue Function, High Function, GetLabel Function

MinValue Function

Format

```
d = minvalue(item_name || value_set_name);
```

Description

The `minvalue` function returns the minimum valid value for an item or value set. If a numeric ***item_name*** is provided, the function looks at the current value set for the item, or if no value set exists, looks at the item itself (returning a minimum value based on the item's number of digits). If a numeric ***value_set_name*** is provided, the function returns the minimum range or discrete value in the value set.

Return Value

The function returns the minimum valid value for an item or value set.

Example

```
PROC AGE

    if RELATIONSHIP = 1 and AGE < minvalue(AGE_HEAD_VS) then
        errmsg("The head of household must be at least %d-years-old.", minvalue(AGE_HEAD_VS));
        reenter;
    endif;
```

See also: MaxValue Function, Low Function, GetLabel Function

Paradata Function

Format

```
d = paradata(action_type, argument1, ..., argumentN);
```

Description

The `paradata` function performs an action related to the paradata log. The first argument, ***action_type***, must be one of the following:

action_type	Description
concat	Concatenates several paradata logs into a single log. The first argument is the file name of the output log. The subsequent arguments are the file names (as string expressions or string lists) of input logs. You can concatenate into the currently open paradata log, but in that case you must also include the currently open paradata log as an input.
query	Executes a query on the paradata log. This is the same as: `sqlquery(filename(paradata),...)`.
open	Opens a paradata log to start logging events. Specify the file name as a string expression as the sole argument.
close	Closes the currently open paradata log.
flush	Writes all cached paradata events to the log. This is rarely needed, and may only be necessary if you want to query the paradata log from an external application.

It is not recommended that you open or close paradata logs via logic, as the log will not contain all of the initializing features that exist when specifying the paradata log when the application begins (and via the PFF file).

Return Value

For the **open**, **close**, and **flush** actions, the function returns a logical value of 1 (true) if successful and 0 (false) otherwise. For the **concat** action, the function returns the number of paradata logs concatenated, or `default` upon error. Look at the documentation for the `sqlquery` function to see the return values for the **query** action.

Example - Concat

```
list string logs_list;

dirlist(logs_list, "C:\Survey\Paradata", "*.cslog");

// add the paradata logs into the currently open log
paradata(concat, filename(paradata), filename(paradata), logs_list);
```

Example - Query

```
string message_events_query = maketext(
    "SELECT COUNT(*) "
    "FROM message_event "
    "JOIN event ON message_event.id = event.id "
    "JOIN case_instance ON event.case_instance = case_instance.id "
    "JOIN case_info ON case_instance.case_info = case_info.id "
    "WHERE case_info.uuid = '%s';",
    uuid(DICT_NAME));

numeric message_events_for_this_case = paradata(query, message_events_query);
```

See also: Paradata, SqlQuery Function, SyncParadata Function

PathConcat Function

***Feature Upgrade**: Starting with CSPro 7.5, you are encouraged to use the `path.concat` function.*

Format

```
s = pathconcat(path1[, ..., pathN]);
```

Description

The `pathconcat` function concatenates the values of multiple string expressions (***path1*** + ... + ***pathN***) into one string representing a file path. The first argument can be either a string expression or one of the path types documented in the `pathname` function. The combined paths are returned as an absolute, not relative, path.

Return Value

The function returns the concatenated path string.

Example

```
string image_filename = pathconcat(Application, "../Images", maketext("%v%v.jpg", CLUSTER, HHNO)
);
```

See also: Path.Concat Function, PathName Function, Concat Function

PathName Function

Format

```
s = pathname(path_type || dictionary_name || file_handler);
```

Description

The following ***path_type*** arguments return the folder name associated with a special directory on your device:

path_type	Description
Application	The folder where the application file (.ent or .bch) is located.
CSPro	The folder where the CSPro executable files are located.
Downloads	The operating system's directory for storing downloaded files.
Html	The folder where CSPro's HTML files are located.
InputFile	The folder where the main data file (associated with the primary dictionary) is located.
Temp	The operating system's temporary directory.

There are some options specific to Windows:

path_type	Description
Desktop	The current user's Desktop folder.
Documents	The current user's Documents folder.
ProgramFiles32	The folder containing installed 32-bit applications.
ProgramFiles64	The folder containing installed 64-bit applications. If the computer has only a 32-bit processor, this returns the same value as ProgramFiles32.
Windows	The folder contains the Windows operating system.

There are some options specific to mobile devices:

path_type	Description
CSEntry	The folder from which CSPro applications are run.
CSEntryExternal	On devices with removable storage (e.g., an SD card), the folder on removable storage to which CSPro applications can write.

If supplying a ***dictionary_name***, the function returns the folder name of the dictionary file (.dcf) itself, not of the data file associated with the dictionary. On mobile devices, the function returns the folder name of the .pen file because dictionary files (.dcf) are not used directly when running .pen applications.

If providing a ***file_handler*** (declared in a `file` statement), the function returns the folder name containing the text file associated with the file handler.

Return Value

The function returns a string containing the fully qualified folder name.

Example

```
setfile(log_file, pathname(Desktop) + "Census Daily Report.txt", create);
```

See also: Path Namespace, FileName Function

Report Function (Deprecated)

***Feature Upgrade**: Starting with CSPro 7.7, the version of Templated Reports that uses `report` has been superseded by a new version, Templated Reports. This version will be removed in a future version of CSPro.*

Format

```
b = report(template_file_name[,output_file_name]);
```

Description

The `report` function generates a templated report. The function reads the template specified in the string expression ***template_file_name*** and analyzes the template for any queries that need to be executed. After running the queries, and passing to the template any data created using the `setreportdata` function, a report is created based off the template.

The optional string expression ***output_file_name*** specifies the name for the created report. If this argument is not specified, the report will be saved to a temporary file and it will be displayed in a web browser. These temporary reports are deleted when the application closes.

Return Value

The function returns a logical value of 1 (true) if the report was generated correctly, and 0 (false) otherwise. Any errors encountered when creating the report are displayed, which will help you diagnose any problems with the template.

Example

```
// create and view the report
report("interviewer_report_template.html");

// alternatively, save the report
report("interviewer_report_template.html","interviewer_report." + getoperatorid() + ".html");
```

See also: SetReportData Function (Deprecated), Templated Reports (Deprecated)

SaveSetting Function

Format

```
b = savesetting(attribute, value);
```

Description

The `savesetting` function saves a setting as an attribute-value pair using the string expression ***attribute*** and the string or numeric expression ***value***. You can save multiple settings using different attribute strings and can retrieve the settings using the `loadsetting` function. These settings are stored locally on the device and can be accessed by other programs using the attribute string. Using settings is a way to share data between different CSPro applications on the same device.

The file from which settings are retrieved can be modified by altering the CommonStore attribute of a PFF file.

All settings can be cleared using the following command: `savesetting(clear);`

Return Value

The function returns 1 (true) if successful or 0 (false) if the function failed.

Example

```
PROC INTERVIEWER_NAME

preproc

    string InterviewerNameKey = "InterviewerName";

    INTERVIEWER_NAME = loadsetting(InterviewerNameKey);

postproc

    if INTERVIEWER_NAME = "" then
        errmsg("You must enter the interviewer's name.");
        reenter;
    endif;

    savesetting(InterviewerNameKey, INTERVIEWER_NAME);
```

See also: LoadSetting Function, User and Configuration Settings, CSEntry Settings Modification

SetLanguage Function

Format

```
b = setlanguage(language_name);
```

Description

The `setlanguage` function changes the language being used by an application. Pass a string expression, ***language_name***, with the name, not the label, of the language to which you want to change.

Return Value

The function returns 1 (true) if the language exists and the language was successfully changed. Otherwise it returns 0 (false).

Example

```
PROC INTERVIEW_LANGUAGE

    if INTERVIEW_LANGUAGE = 1 then
        setlanguage("EN");
    elseif INTERVIEW_LANGUAGE = 2 then
        setlanguage("FR");
    else
        setlanguage("ZH");
    endif;
```

See also: GetLanguage Function, OnChangeLanguage Global Function, Multiple Language Applications

SetProperty Function

Format

```
i = setproperty([symbol,] property_name, property_value);
```

Description

The `setproperty` function modifies the current property, specified by the string expression ***property_name***, associated with an application or a field. If an optional dictionary ***symbol*** is specified, then the property must apply to a field. If the symbol is not a field but is intead a block, group, form, or dictionary, then the function will try to apply the property to all fields belonging to that symbol. If no symbol is specified, then the property must be a system or application property. The property will be modified to the value given in the numeric or string expression ***property_value***.

Properties List

Name	Settable?	Values
System Properties		
MaxDisplayWidth		(integer of the maximum display width)
MaxDisplayHeight		(integer of the maximum display height)
Application Properties		
AppType		DataEntry, Batch, Tabulation
AutoAdvanceOnSelection *M	✔	Yes, No
AutoPartialSaveMinutes	✔	(integer of the interval)
CAPI		Yes, No
CaseTree		Always, Desktop, Mobile, Never
CenterForms *D		Yes, No
ComboBoxShowOnlyDiscreteValues	✔	Yes, No
CreateListing		Yes, No
CreateLog		Yes, No
DecimalComma *D		Yes, No
DisplayCodesAlongsideLabels *M	✔	Yes, No
NotesDeleteOtherOperators	✔	Yes, No
NotesEditOtherOperators	✔	Yes, No
OperatorID		Yes, No
ParadataCollection		AllEvents, SomeEvents, No

ParadataDeviceStateMinutes	✔	(integer of the interval)
ParadataGpsLocationMinutes	✔	(integer of the interval)
ParadataRecordCoordinates	✔	Yes, No
ParadataRecordIteratorLoadCases	✔	Yes, No
ParadataRecordValues	✔	Yes, No
PartialSave	✔	Yes, No
Path		Yes (system controlled), No (operator controlled)
ShowEndCaseDialog	✔	Yes, No
ShowErrorMessageNumbers *M	✔	Yes, No
ShowFieldLabels *M		Yes, No
ShowLabelsInCaseTree *M	✔	Yes, No
ShowNavigationControls *M	✔	Yes, No
ShowRefusals	✔	Yes, No
ShowSkippedFields*M	✔	Yes, No
SpecialValuesZero	✔	Yes, No
UpdateSaveArrayFile	✔	Yes, No
ValidateAlphaFields	✔	Yes, No
WindowTitle	✔	(string of the window title)

Item Properties

DataType		Alpha, Numeric
Decimal		(integer)
DecimalChar		Yes, No
Len		(integer)
ZeroFill		Yes, No

Field Properties

AllowMultiLine		Yes, No
AutoIncrement		Yes, No
CanEnterNotAppl	✔	No, Confirm, NoConfirm
CanEnterOutOfRange	✔	No, Confirm, NoConfirm
CapturePosX	✔	(integer of the X position)
CapturePosY	✔	(integer of the Y position)
CaptureType	✔	Barcode, CheckBox, ComboBox, Date, DropDown, NumberPad, RadioButton, Slider, TextBox, ToggleButton
CaptureDateFormat	✔	(string of the date format, e.g., YYYYMMDD)
ForceOutOfRange	✔	Yes, No
HideInCaseTree	✔	Yes, No
Keyboard	✔	(integer of the keyboard ID)
Persistent		Yes, No
Protected	✔	Yes, No
Sequential		Yes, No
ShowExtendedControlTitle*D	✔	Yes, No
SkipTo		(string of the field name)
UpperCase	✔	Yes, No
UseEnterKey	✔	Yes, No
UseUnicodeTextBox		Yes, No
ValidationMethod	✔	Default, Confirm, NoConfirm, Custom
Verify		Yes, No

*D desktop only, *M mobile only

Return Value

The function returns the number of items whose property was successfully changed. If either the property name or value is invalid, then the function returns `default`.

Example

```
PROC CENSUS_QUEST

preproc

    if STAFF_ROLE = 1 then // interviewer
        setproperty("DisplayCodesAlongsideLabels", false);
        setproperty(CENSUS_DICT,"CanEnterNotAppl", "No");
        setproperty(CENSUS_DICT,"CanEnterOutOfRange", "No");

    else // supervisor
        setproperty("DisplayCodesAlongsideLabels", true);
        setproperty(CENSUS_DICT,"CanEnterNotAppl", "NoConfirm");
        setproperty(CENSUS_DICT,"CanEnterOutOfRange", "NoConfirm");

    endif;
```

See also: GetProperty Function, Protect Function

SetReportData Function (Deprecated)

***Feature Upgrade**: Starting with CSPro 7.7, the version of Templated Reports that uses `setreportdata` has been superseded by a new version, Templated Reports. This version will be removed in a future version of CSPro.*

Format

```
b = setreportdata([attribute,] data_source);
```

Description

The `setreportdata` function stores in memory a piece of data that will later be used by the `report` function to generate a templated report. The string expression ***attribute*** specifies the key, or tag, that uniquely identifies the piece of data. The attribute is case sensitive. In the report template, you will use this attribute to refer to this data.

The ***data_source*** must be one of the following:

data_source	**Description**
value	A string expression containing some text.
record_name	The contents of a record, including multiply occurring records. In a data entry application, only information up to the number of entered occurrences will be stored.
list_name	The contents of a list.
array_name	The contents of a one- or multiple-dimensional array.
sqlquery	The result set of a query executed by the `sqlquery` function. Do not specify that function's optional ***result_set*** argument, as the result set is stored in memory when called by `setreportdata`.

It is not necessary to specify the ***attribute*** when working with a ***record_name***, ***list_name***, or ***array_name***. If no ***attribute*** is specified, then it will default to the name of the record, list, or array.

You can clear all stored data by executing: `setreportdata(clear)`.

Return Value

The function returns a logical value of 1 (true) if the data was stored successfully, and 0 (false) otherwise (for example if the `sqlquery` function encountered a problem).

Example - Text and Record

```
setreportdata("title", "Demographic Report");
setreportdata(PERSON_REC);
```

Example - SqlQuery

```
string DeviceQuery =
    "SELECT device_model, COUNT(*) AS device_count "
    "FROM device_info "
    "GROUP BY device_model;";

setreportdata("device_info", sqlquery(paradata, DeviceQuery));
```

See also: Report Function (Deprecated), SqlQuery Function, Templated Reports (Deprecated)

SetValueSet Function

Format

```
b = setvalueset(item_name, value_set_name);
```

Description

The `setvalueset` function allows you to dynamically change an item's current value set. An item's current value set is used to determine whether or not entered values are out of range. The current value set is also used to provide the value choices displayed in capture types. The changing of a value set is not permanent; it remains in effect only during the processing of the current case or until the next call to `setvalueset`.

The ***item_name*** is the name of the item in the data dictionary whose value set is to be changed. The argument is usually an item name but, preceded by the `@` symbol, it can also be a string expression that evaluates to the name of an item; for example:

```
setvalueset(@getsymbol(), MY_VALUESET_VS1);
```

The argument ***value_set_name*** is the name of an existing value set in the dictionary for the specified item. The function will replace the values of the current value set for the item with the values from the value set specified by ***value_set_name***, which is either the name of a value set or a string expression that evaluates to a value set name. The ***value_set_name*** can also be the name of a dynamic value set.

The `setvalueset` function can also create dynamic value sets from array arguments.

Return Value

When using a value set specified in the dictionary, the function returns a logical value of 1 (true) if the value set was changed successfully and 0 (false) otherwise (for example, if the value set name is invalid). With dynamic value sets, the function returns the number of codes in the newly created value set.

Example 1

```
PROC D05_CAUSE_DEATH

onfocus

    if D02_SEX = 1 then
        setvalueset(SEX, D05_CAUSE_DEATH_MALE_VS);

    else
        setvalueset(SEX, D05_CAUSE_DEATH_FEMALE_VS);

    endif;
```

Example 2

```
PROC DISTRICT

preproc

    setvalueset(DISTRICT, maketext("DISTRICT_VS%v", PROVINCE));
```

Example (Dynamic Value Set)

```
PROC SECOND_FAVORITE_DRINK

preproc

    valueset drink_valueset = FAVORITE_DRINK_VS;

    drink_valueset.remove(FAVORITE_DRINK);

    setvalueset(SECOND_FAVORITE_DRINK, drink_valueset);
```

See also: SetValueSets Function, InValueSet Function

SetValueSet Function With Arrays

***Feature Upgrade**: Starting with CSPro 7.3, you should no longer use this function as it may soon be removed from CSPro. To replicate the behavior of the function, you can use the `setvalueset` function with dynamic value sets.*

Format

```
i = setvalueset(item_name, codes_array, labels_array[, images_array]);
```

Description

The `setvalueset` function allows you to dynamically change an item's current value set. An item's current value set is used to determine whether or not entered values are out of range. The current value set is also used to provide the value choices displayed in capture types. The changing of a value set is not permanent; it remains in effect only during the processing of the current case or until the next call to `setvalueset`.

The ***item_name*** is the name of the item in the data dictionary whose value set is to be changed. The second argument, ***codes_array***, is an array of codes to use in the new value set and the third argument, ***labels_array***, is an array of labels that correspond to the values in the codes array (the nth element of the labels array is the label for the nth element in the codes array). If the item is numeric, ***codes_array*** must be a numeric array. If the item is alphanumeric, ***codes_array*** must be a string array. The ***labels_array*** must always be a string array. The optional fourth argument, ***images_array***, is a string array pointing to image filenames to be used as value set images.

When working with the ***codes_array***, to indicate the end of a numeric array, assign the special value `notappl` to the position after the last valid code. To indicate the end of a string array, assign a blank value. This indicates when to stop loading values from the array.

In versions of CSPro earlier than 6.0, the value set labels and codes array needed to start at index position 0. Zero-based arrays are still permitted, but if a one-based array is passed to the function (and the zeroth element has never been assigned to), then CSPro will start processing the dynamic value set at array position 1.

Return Value

The function returns the number of codes in the newly created value set.

Example

```
PROC GLOBAL

array vsCodes(100);
array string vsLabels(100);

PROC DISTRICT

preproc

    vsCodes(1) = 1;
    vsLabels(1) = "District 1";

    vsCodes(2) = 2;
    vsLabels(2) = "District 2";

    vsCodes(3) = 3;
    vsLabels(3) = "District 3";

    vsCodes(4) = notappl; // mark the end of the array

    setvalueset(DISTRICT, vsCodes, vsLabels);
```

See also: SetValueSet Function

SetValueSets Function

Format

```
i = setvaluesets(value_sets_search_query);
```

Description

The `setvaluesets` function allows you to dynamically change the value sets associated with multiple items. Unlike the `setvalueset` function, which only operates on one item, the `setvaluesets` function traverses every item in the primary dictionary and searches each item's value sets to see if they have a name that contains the text found in the string expression ***value_sets_search_query***. If the text is found, the first value set with the name containing the expression is the value set used for the item. Otherwise the currently used value set is maintained.

Before multiple language dictionary labels were added in CSPro 6.2, this function was useful for changing the language of value sets in a multi-language application.

Return Value

The function returns the number of items whose value sets were changed.

Example

```
function ResetLanguage(language)

    if language = 1 then // English
        setvaluesets("_ENG");

    elseif language = 2 then // Russian
        setvaluesets("_RUS");

    else // Tajik
        setvaluesets("_TAJ");

    endif;

end;
```

See also: SetValueSet Function

Special Function

Format

```
b = special(numeric_expression);
```

Description

The `special` function determines whether the value of ***numeric_expression*** is one of the four special values: `missing`, `refused`, `notappl`, or `default`.

Return Value

The function returns a logical value of 1 (true) if the variable is a special value and 0 (false) otherwise.

Example

```
if special(CHILDREN_EVER_BORN) then
    CHILDREN_EVER_BORN = 0;
endif;
```

See also: CountNonSpecial Function, In Operator

SqlQuery Function

Format

```
d = sqlquery(sqlite_database[, result_set], sql_query);
```

Description

The `sqlquery` function executes a query on a SQLite database and returns a result set in a variety of formats. This function is intended for advanced users who are familiar with writing SQL expressions. While many files used by CSPro applications are text files, there are some files that are stored as SQLite databases, including CSPro DB data files and paradata logs. You can query these files with this function.

The first argument, ***sqlite_database***, must be one of the following:

sqlite_database	Description
`paradata`	The query will be executed on the currently open paradata log. This is the same as:

	paradata(query,...).
dictionary_name	The data source pointed to by ***dictionary_name*** must be of type CSPro DB, Encrypted CSPro DB, or text. The query will be executed on the file associated with this data source. If working with text files, you can query the file's index.
file_name	The string expression ***file_name*** gives the file name of a SQLite database. The database will be opened, the query executed, and then the database will be closed. This allows you to work with databases maintained outside of CSPro. You can query encrypted databases by specifying a password in a connection string or by having an operator enter the password manually.

The optional second argument, ***result_set***, indicates the destination for the result set generated following the execution of the query. If you do not provide a result set for your query, the query must be a scalar query (one that returns a single result such as `"SELECT COUNT(*) FROM ..."`). The result set must be one of the following:

result_set	**Description**
record_name	The results of a query can fill a CSPro dictionary's record. The ***record_name*** must point to a record in a working storage dictionary. The function will look at the column names of the result set to determine which columns of the results get stored in which items of the record. The results cannot be stored in subitems or multiply occurring items.
list_name	Because lists are only one-dimensional, only the first column of results will be stored in ***list_name***, which can be of type `numeric` or `string`. Because the size of a list is dynamic, the length of the list will be equal to the number of rows in the result set (up to a maximum of 10,000 rows).
array_name	You can use one- or two-dimensional arrays of type `numeric` or `string`. When using a one-dimensional array, only the first column of results will be stored in ***array_name***. When using a two-dimensional array, each column of results will be stored in a separate column of ***array_name***.

Using a working storage record for the results is advantageous because the results are stored in their proper type. Because lists and arrays are all of one type (`numeric` or `string`), some results may be converted to a invalid type (for example, a string value may be stored in a numeric array).

The final argument, ***sql_query***, is a string expression containing the SQL query.

Return Value

If executing a scalar query, the function returns the queried value. Otherwise, the function returns the number of rows stored in the result set. If there was an error executing the query, then the function returns `default`.

Example - Scalar Query on CSPro DB File

```
numeric numberVerifiedCases = sqlquery(SURVEY_DICT, "SELECT COUNT(*) FROM cases WHERE
cases.verified != 0;");
errmsg("%d cases have been verified", numberVerifiedCases);
```

Example - Query on CSPro DB File, Storing Results in List

```
list string verifiedCaseKeys;

sqlquery(DICT_NAME, verifiedCaseKeys, "SELECT cases.key FROM cases WHERE cases.verified != 0;");

do numeric ctr = 1 while ctr <= verifiedCaseKeys.length()
    errmsg("Verified case key: %s", verifiedCaseKeys(ctr));
enddo;
```

Example - Query on Paradata Log, Storing Results in Array

```
string SessionQuery =
    "SELECT operatorid_info.operatorid, event_start.time, event_end.time "
    "FROM session_event session_event_start "
    "JOIN event event_start ON session_event_start.id = event_start.id "
    "JOIN session_event session_event_end "
    "JOIN event event_end ON session_event_end.id = event_end.id AND
event_start.session_instance = event_end.session_instance "
    "JOIN session_instance ON event_start.session_instance = session_instance.id "
    "JOIN session_info ON session_instance.session_info = session_info.id "
    "JOIN operatorid_info ON session_info.operatorid_info = operatorid_info.id "
    "WHERE session_event_start.action = 1 AND session_event_end.action = 0 "
    "ORDER BY event_start.time;";

array string entrySessions(500, 3);

numeric numberSessions = sqlquery(paradata, entrySessions, SessionQuery);

do numeric ctr = 1 while ctr <= numberSessions
    // columns two and three are numeric values, but because they are stored in a
    // string array, we must convert them before working with them
    numeric startTimestamp = tonumber(entrySessions(ctr, 2));
    numeric endTimestamp = tonumber(entrySessions(ctr, 3));

    errmsg("Operator '%s' worked on %s at %s for %0.2f minutes",
        entrySessions(ctr, 1),
        edit("9999-99-99", sysdate("YYYYMMDD", startTimestamp)),
        edit("99:99", systime("HHMM", startTimestamp)),
        ( endTimestamp - startTimestamp ) / 60);
enddo;
```

Example - Query on Paradata Log, Storing Results in Working Storage Record

This example displays the same results as the previous example but instead stores the information in a working storage record. This multiply occurring record, `WS_ENTRY_SESSIONS_REC`, contains three items: `OPERATOR_NAME`, `START_TIMESTAMP`, and `END_TIMESTAMP`, with only the first item being alphanumeric. Note that the result set's column names are specified using `"AS"`.

```
string SessionQuery =
    "SELECT operatorid_info.operatorid AS OPERATOR_NAME, event_start.time AS START_TIMESTAMP,
event_end.time AS END_TIMESTAMP "
    "FROM session_event session_event_start "
    "JOIN event event_start ON session_event_start.id = event_start.id "
    "JOIN session_event session_event_end "
    "JOIN event event_end ON session_event_end.id = event_end.id AND
event_start.session_instance = event_end.session_instance "
    "JOIN session_instance ON event_start.session_instance = session_instance.id "
    "JOIN session_info ON session_instance.session_info = session_info.id "
    "JOIN operatorid_info ON session_info.operatorid_info = operatorid_info.id "
    "WHERE session_event_start.action = 1 AND session_event_end.action = 0 "
    "ORDER BY event_start.time;";

sqlquery(paradata, WS_ENTRY_SESSIONS_REC, SessionQuery);

do numeric ctr = 1 while ctr <= count(WS_ENTRY_SESSIONS_REC)
    errmsg("Operator '%s' worked on %s at %s for %0.2f minutes",
        strip(OPERATOR_NAME(ctr)),
        edit("9999-99-99", sysdate("YYYYMMDD", START_TIMESTAMP(ctr))),
        edit("99:99", systime("HHMM", END_TIMESTAMP(ctr))),
        ( END_TIMESTAMP(ctr) - START_TIMESTAMP(ctr) ) / 60);
enddo;
```

Example - Using Callback Functions

Adding the directive `sql` after the `function` keyword when defining a user-defined function indicates that the SQLite query can call back into CSPro logic. These functions can accept `numeric` or `string` arguments as part of the query. In the following example, the same results are displayed as in the previous two examples but without having to store the

result set.

```
PROC GLOBAL

function sql DisplayEntrySession(string operatorName, numeric startTime, numeric endTime)
    errmsg("Operator '%s' worked on %s at %s for %0.2f minutes",
        operatorName,
        edit("9999-99-99", sysdate("YYYYMMDD",startTime)),
        edit("99:99", systime("HHMM",endTime)),
        ( endTime - startTime ) / 60);
end;

PROC EXAMPLE

    string SessionQuery =
        "SELECT DisplayEntrySession(operatorid_info.operatorid, event_start.time,
event_end.time) "
        "FROM session_event session_event_start "
        "JOIN event event_start ON session_event_start.id = event_start.id "
        "JOIN session_event session_event_end "
        "JOIN event event_end ON session_event_end.id = event_end.id AND
event_start.session_instance = event_end.session_instance "
        "JOIN session_instance ON event_start.session_instance = session_instance.id "
        "JOIN session_info ON session_instance.session_info = session_info.id "
        "JOIN operatorid_info ON session_info.operatorid_info = operatorid_info.id "
        "WHERE session_event_start.action = 1 AND session_event_end.action = 0 "
        "ORDER BY event_start.time;";

    sqlquery(paradata, SessionQuery);
```

See also: CSPro DB File Format, Paradata Function

Stop Function

Format

```
b = stop([stop_code]);
```

Description

The `stop` function prematurely ends a CSPro application. The optional numeric expression ***stop_code*** determines how the program ends. If not specified, the code defaults to 0.

If the function is used in a data entry application, the stop code determines whether data entry is stopped just for the current case or whether the entire data entry application is closed. If the code is 0, entry of the current case is stopped but CSEntry remains active, which is the same behavior as pressing the stop button. If the code is 1, entry of the current case is stopped and CSEntry terminates. If the function is executed in the `postproc` of the first (highest) level, then the data for the case is saved. Otherwise, any data entered for the current case is lost. If you want to avoid losing data, you can call `savepartial` just before the `stop` function.

If the function is used in a batch edit application, the run is always terminated. If an output file was specified in the batch run, neither the current case nor subsequent cases will be saved to the output file. If the code is not 1, the application postproc will be executed, which allows you to execute tasks like writing summary information to a listing file. This is as if you terminated the batch run using the Cancel button. If the code is 1, the application postproc is not executed.

Return Value

The function returns a logical value of 0 (false) if the program cannot be stopped (for example, if the function is called from an external form). Otherwise the function returns 1 (true).

Data Entry Example

```
if VALUE = 99 then
    savepartial();
    stop(1);
endif;
```

Batch Edit Example

```
if number_total_errors > 100 then
    stop();
endif;
```

See also: EndCase Statement, Skip Case Statement

SysParm Function

Format

```
s = sysparm(parameter_name);
```

Description

The `sysparm` function returns the value of a parameter provided in a data entry or batch edit PFF file. The function returns the parameter as a left-justified string. If no parameter was given in the PFF file, the function returns a blank string.

If a string expression ***parameter_name*** is provided, the function looks to see if there is a parameter with the given name specified in the **[Parameters]** section of the PFF file. If such a parameter exists, it is returned as a string. Using this functionality, you can pass several parameters to your programs.

On Windows, it is possible to check for the presence of a command line argument using this function. If ***parameter_name*** is not in the PFF file, the function will check if that string is specified in the command line and if so, will return a non-blank string.

Return Value

The function returns a string containing the parameter.

Example PFF

```
[Parameters]
Parameter=Otter
State=20
County=5
```

Example Logic

```
errmsg("%s",sysparm());                 // displays Otter
STATE = tonumber(sysparm("State"));     // sets STATE to 20
COUNTY = tonumber(sysparm("County"));   // sets COUNTY to 5
```

See also: Pff.GetProperty Function

Tr Function

Format

```
s = tr(string_expression || message_number);
```

Description

The `tr` function translates text into the appropriate language based on the application current's language. The text can come from a ***string_expression***, generally a string literal, or from a ***message_number***. Text translations are defined in the application's message file.

Return Value

The function returns a string with the translated message. If there is no available translation, then the original message is returned.

Example

```
FR("Hello") Bonjour

// with the preceding message file, the error message displays:
errmsg(tr("Hello"));  // Hello
errmsg(tr("World!")); // World!

setlanguage("FR");
errmsg(tr("Hello"));  // Bonjour
errmsg(tr("World!")); // World!
```

See also: Message File (.mgf), Multiple Language Applications, SetLanguage Function

Trace Function

Programmers use tracing to obtain low-level information about how an application runs, with the information often used for debugging purposes to understand why a program does not execute as expected. CSPro offers limited tracing functionality for two objectives: logging user-generated information and tracking executed statements. Tracing messages can be displayed in a window or saved to a file.

Activating and Disabling Tracing

```
// turns on tracing and outputs messages to a window
trace(on);

// turns on tracing and appends messages to "filename"
trace(on,"filename");

// turns on tracing, clears any contents in the file, and writes messages to "filename"
trace(on,"filename",clear);

// turns off tracing, closing all open trace windows or files
trace(off);
```

It is possible, by calling the `trace` function twice, to send messages to both a window and a file. If the filename does not contain a directory, the file will be placed in the application folder. On mobile devices, it is only possible to trace to a file and any trace statements output to a window will be ignored.

Logging User-Generated Information

To send a message to the trace log, simply pass a string to the `trace` function using any message formatting options (like used by the `errmsg` function).

```
if distance > 100 then
    trace("distance (%d) > 100!",distance);
endif;
```

Tracking Executed Statements

Occasionally a programmer wants to observe how logic statements are executed, particularly when the logic behaves in a manner inconsistent with the programmer's expectations. The trace window or file can display each line of logic as it is executed. Because in some applications this may be a very large number of statements, the programmer must specify what elements of the logic should be outputted:

```
// logic statements after this point will be outputted
set trace;

// same as above
set trace(on);

// logic statements after this point will not be outputted
set trace(off);
```

The `set trace` statement indicates to CSPro that logic statements should or should not be outputted but the statements will only be outputted if tracing is activated, thus the `trace` function and `set trace` statements must be used together.

Example

```
trace("There is no trace window open so this message is discarded");

// opens the trace file and clears previous contents
trace(on,"trace.txt",clear);

trace("This message appears in the file");

trace("Complex strings can be outputted using errmsg-style formatting; e.g., e = %0.3f",exp(1));

// closes the trace file
trace(off);

// opens the trace file and now messages will be appended to the end of the file
trace(on,"trace.txt");

set trace;

numeric value = 10;

if value > 10 then
    errmsg("A");
elseif value < 10 then
    errmsg("B");
else
    errmsg("C");
endif;

errmsg("This statement will appear on the trace window");

set trace(off);

errmsg("This statement will not appear on the trace window");
```

Trace Output

As the following trace results show, the output for conditional statements (e.g., if) and loops (e.g., do) is limited. Trace results show the line numbers to the left of the executed statements.

```
Trace started at 02/03/11 20:28:35

TRACE   This message appears in the file
TRACE   Complex strings can be outputted using errmsg-style formatting; e.g., e = 2.718

Trace stopped at 02/03/11 20:28:35
Trace started at 02/03/11 20:28:35

31   :  numeric value = 10;
33   :  if value > 10 then
38   :  errmsg("C");
41   :  errmsg("This statement will appear on the trace window");
43   :  set trace(off);

Trace stopped at 02/03/11 20:28:35
```

See also: LogText Function, Message Formatting Options

UUID Function

Format

```
s = uuid([dictionary_name]);
```

Description

This `uuid` function returns a human-readable string containing a universally unique identifier (UUID), also referred to as a globally unique identifier (GUID). This is a 36-character string that will be formatted in groups of 8, 4, 4, 4, and 12 characters, separated by hyphens. For example:

e459e2b6-1fec-4d71-86c1-078cc2fe7433

This is a string representation of a 128-bit number that can essentially be considered globally unique. The probability of generating the same UUID twice is near zero. You will receive a different string every time you call the function.

If an optional ***dictionary_name*** is specified, the function returns the UUID (internal key) associated with the currently loaded case. This may be useful when working with paradata queries.

Return Value

The function returns a string containing the human-readable representation of the identifier.

Example

```
PROC UNIQUE_ID

preproc

if UNIQUE_ID = "" then
    UNIQUE_ID = uuid();
endif;
```

See also: Random Function

View Function

Format

```
b = view(filename || website_address || object_name⌡, optional_named_arguments⌡);
```

Description

The `view` function displays the contents of a file or shows a website. The function expects a string expression representing a file (***filename***) or a website URL (***website_address***), or a CSPro object (***object_name***) such as a document, freq, image, or report. If a website URL or a file with the extension *.html* or *.htm* is provided, it is shown in an embedded window. For other files, an external application is launched to display the file using the system's default viewer for that file type.

HTML files or websites are displayed in an embedded window using a Chromium-based web browser (Edge WebView2 on Windows; WebView on Android). In this web browser, it is possible to use the JavaScript interface to run user-defined functions or to execute other CSPro operations.

Optional named arguments can be used to control how the embedded web browser displays on Windows:

- `title :=` string expression giving the title of the window.
- `width :=` numeric expression specifying the width of the window in display units.
- `height :=` numeric expression specifying the height of the window in display units.
- `showCloseButton :=` conditional expression that, if false, hides the window's *Close* button.

If either the height or width is specified, then both values have to be provided. Hiding the *Close* button also removes the margin around the web browser, so the browser will fill the entire embedded window.

Return Value

The function returns a logical value of 1 (true) if the file exists and the external application or embedded web browser could be launched successfully and 0 (false) otherwise.

Examples

```
view("https://www.census.gov/data/software/cspro.html");

view("../Nineties Images/carn03.jpg");

view("hh-report.html", title := "Household Completion Report",
                       width := 400, height := 600,
                       showCloseButton := false);
```

See also: ExecSystem Function (Desktop), ExecSystem Function (Mobile), HTML in CSPro, JavaScript Interface

Warning Function

Format 1

```
i = warning(string_exp⌡, argument1, ..., argument20⌡);
```

Format 2

```
i = warning(msg_num⌡, argument1, ..., argument20⌡);
```

Format 3

```
i = warning(string_exp[, argument1, ..., argument20])

    [select(caption1, fieldname1[, ..., caption5, fieldname5]) [default(default_option)] ]
    [case || summary]
    [denom = numeric_denom];
```

Description

The warning function is similar to the errmsg function, with the function only having special behavior within a data entry application. If used in a batch or tabulation application, it behaves identically to errmsg.

The warning function differs from errmsg in that it can be used for "soft check" error messages. These messages will not be displayed to the operator when advancing in the case (such as when resuming from a partial save). This can also be useful for information you want to display just once to the operator, such as during initial entry of a field, rather than when the operator is moving from one section to another section, and the field in question lays somewhere in between.

Each argument is sequentially inserted into the warning message. Arguments can be numeric or alphanumeric expressions, but the type of the argument must match the type of the receiving field in the message text (see errmsg for a complete description of the function's arguments). The maximum number of arguments is 20. If messages are defined via the message number ***msg_num***, then those messages will be stored in a message file [.mgf]. ***msg_num*** can be a number or numeric expression.

If using the select option, then one of the field arguments must be continue or you must specify a ***default_option***, which corresponds to one of the select choices begin presented. This is necessary if advancing within the application, for while the message will not be displayed, the logic will be evaluated, and CSPro needs to know where to go. See below for an example on this usage. If the system is not processing an advance, using ***default_option*** places the focus on the select option chosen when the message is presented. If a ***default_option*** is not given, no button will have the focus.

Return Value

The function returns the cardinal number of the button pressed. When the **select** keyword is not used, the function will return 1, as only the "OK" button is presented to the user, corresponding to button #1 ("Cancel" is never presented, and so a code of zero is not possible).

When the **select** keyword is used, the function returns the sequence number of the button that was pressed, with the first button having the value 1. A maximum of 5 select argument pairs is allowed.

Example

```
PROC MARITAL_STATUS
postproc
    if curocc() > 1 and $ in 1,2 { married, partner } and SEX=SEX(HEAD) then
        warning("Is the sex of this person the same as the head of household's?")
               select("Correct", continue,
                      "Incorrect: Change this person's sex", SEX,
                      "Incorrect: Change the head's sex", SEX(HEAD))
                      default(1); // during an advance, use the first choice, continue
                             // if not processing an advance, the focus will be on button 1
    endif;
```

See also: ErrMsg Function, Message Formatting Options, InAdvance Function

Date and Time Functions

AdjLBA Function

Format

```
i = adjlba(lower_cmc1,upper_cmc1,lower_cmc2,upper_cmc2,age);
```

Description

The `adjlba` function adjusts the lower bound of the *century month code* (CMC) of an event based on an age. It is generally used to calculate the minimum date of birth based on the date of an interview.

The ***lower_cmc1*** and ***upper_cmc1*** arguments are numeric expressions generally representing the date of birth CMC range. The ***lower_cmc2*** argument is a numeric expression of the CMC of an event, generally the date of an interview. The ***upper_cmc2*** argument is not used (but is used by `adjuba`). The ***age*** argument is a numeric expression expressing the age of the person in complete years.

Return Value

The function returns the adjusted lower bound of the CMC of an event based on the age. If the adjusted lower bound is greater than ***upper_cmc1***, the function returns the error code of -1.

Example

```
t = adjlba(ldob,udob,di,di,HL6);
```

See also: AdjUBA Function, AdjLBI Function, AdjUBI Function, CMCode Function, SetLB Function, SetUB Function

AdjLBI Function

Format

```
i = adjlbi(lower_cmc1,upper_cmc1,lower_cmc2,upper_cmc2,interval);
```

Description

The `adjlbi` function adjusts the lower bound of the *century month code* (CMC) of an event based on an interval. It is generally used to calculate the minimum *next* date of birth based on a date of birth and a minimum number of months allowed between births.

The ***lower_cmc1*** argument is a numeric expression generally representing the date of birth CMC of a child. The ***upper_cmc1*** argument is not used (but is used by ***adjubi***). The ***lower_cmc2*** and ***upper_cmc2*** arguments are numeric expressions generally representing the date of birth CMC range of the child that was born after the child represented in ***lower_cmc1***. The ***interval*** argument is a numeric expression expressing the minimum number of months between events, which is generally the minimum number of months allowed to pass between a woman's successive childbirths.

Return Value

The function returns the adjusted lower bound of the CMC of an event based on the number of months represented by the interval. If the adjusted lower bound is greater than ***upper_cmc2***, the function returns the error code of -1. It the adjusted lower bound is otherwise valid, the function returns either the adjusted lower bound or ***lower_cmc2***, whichever is greater.

Example

```
t = adjlbi(ldc(i),udc(i),ldc(j),udc(j),7);
```

See also: AdjUBI Function, AdjLBA Function, AdjUBA Function, CMCode Function, SetLB Function, SetUB Function

AdjUBA Function

Format

```
i = adjuba(lower_cmc1,upper_cmc1,lower_cmc2,upper_cmc2,age);
```

Description

The `adjuba` function adjusts the upper bound of the *century month code* (CMC) of an event based on an age. It is generally used to calculate the maximum date of birth based on the date of an interview.

The ***lower_cmc1*** and ***upper_cmc1*** arguments are numeric expressions generally representing the date of birth CMC range. The ***lower_cmc2*** argument is not used (but is used by `adjlba`). The ***upper_cmc2*** argument is a numeric expression of the CMC of an event, generally the date of an interview. The ***age*** argument is a numeric expression expressing the age of the person in complete years.

Return Value

The function returns the adjusted upper bound of the CMC of an event based on the age. If the adjusted upper bound is less than ***upper_cmc1***, the function returns the error code of -1.

Example

```
t = adjuba(ldob,udob,di,di,HL6);
```

See also: AdjLBA Function, AdjLBI Function, AdjUBI Function, CMCode Function, SetLB Function, SetUB Function

AdjUBI Function

Format

```
i = adjubi(lower_cmc1,upper_cmc1,lower_cmc2,upper_cmc2,interval);
```

Description

The `adjubi` function adjusts the upper bound of the *century month code* (CMC) of an event based on an interval. It is generally used to calculate the maximum *previous* date of birth based on a date of birth and a minimum number of months allowed between births.

The ***lower_cmc1*** and ***upper_cmc1*** arguments are numeric expressions generally representing the date of birth CMC range of the child that was born prior to the child represented in ***upper_cmc2***. The ***lower_cmc2*** argument is not used (but is used by ***adjlbi***). The ***upper_cmc2*** argument is a numeric expression generally representing the date of birth CMC of a child. The ***interval*** argument is a numeric expression expressing the minimum number of months between events, which is generally the minimum number of months allowed to pass between a woman's successive childbirths.

Return Value

The function returns the adjusted upper bound of the CMC of an event based on the number of months represented by the interval. If the adjusted upper bound is less than ***lower_cmc1***, the function returns the error code of -1. It the adjusted

upper bound is otherwise valid, the function returns either the adjusted upper bound or ***upper_cmc1***, whichever is less.

Example

```
t = adjubi(ldc(i),udc(i),ldc(j),udc(j),7);
```

See also: AdjLBI Function, AdjLBA Function, AdjUBA Function, CMCode Function, SetLB Function, SetUB Function

DateAdd Function

Format

```
i = dateadd(start_date, period⌈, period_type⌋);
```

Description

The `dateadd` function calculates a new date by adding a period to a starting date. The numeric expression ***start_date*** must be in YYYYMMDD (year/month/day) format. If no year is present, then the current or previous year is assumed in order satisfy the condition that the date is not in the future (based on the computer's system clock). The numeric expression ***period*** is the amount of time that is added to the starting date, and defaults to days. Alternatively, an optional string expression, ***period_type***, modifies the period, and can be one of the following values:

period_type	Description
"d"	days (default)
"w"	weeks
"m"	months
"y"	years

If the period is not an integer, the period will be modified and the calculation will use days. For example, the following calculations are identical:

```
dateadd(20200319, 1.5, "y");
dateadd(20200319, 1.5 * 365.25, "d");
```

Return Value

The function returns a date calculated by adding the period to the starting date. If the starting date cannot be processed, the function returns `default`.

Examples

```
dateadd(20121225, 7);       // returns 20130101

dateadd(20120228, 1);       // returns 20120229
dateadd(20130228, 1);       // returns 20130301

dateadd(20040820, 3, "w");  // returns 20040910
dateadd(20040820, 3, "m");  // returns 20041120
dateadd(20040820, 3, "y");  // returns 20070820

dateadd(20001010, -3, "m"); // returns 20000710
```

See also: DateDiff Function, DateValid Function, SysDate Function

DateDiff Function

Format

```
i = datediff(start_date, end_date [,diff_type]);
```

Description

The `datediff` function returns the difference between two dates as a number, defaulting to the number of days between the dates. This function acts similarly to Microsoft Excel's **datedif** function. The numeric expressions ***start_date*** and ***end_date*** must be in YYYYMMDD (year/month/day) format. If no year is present then the current or next year is assumed in order to satisfy the condition that the start date is earlier than the end date. If years are present, it is possible for the start date to be later than the end date, in which case the function returns a negative difference. An optional string expression, ***diff_type***, indicates the type of date difference requested, and can be one of the following values:

diff_type	Description
"d"	days between the start and end dates (default)
"w"	weeks between the start and end dates
"m"	months between the start and end dates
"y"	years between the start and end dates
"md"	days between the start and end dates ignoring both the years and months of the dates
"ym"	months between the start and end dates ignoring the years of the dates
"yd"	days between the start and end dates ignoring the years of the dates

Return Value

The function returns the requested difference in dates, defaulting to the number of days between the dates. If the dates cannot be processed, the function returns `default`. The function returns values equal to Excel's function for most arguments, but due to processing differences involving leap years, sometimes CSPro's function will return a value different from Excel's.

Example 1

```
datediff(19790404, 19820605);       // returns 1158
datediff(19790404, 19820605, "d");  // returns 1158
datediff(19790404, 19820605, "w");  // returns 165
datediff(19790404, 19820605, "m");  // returns 38
datediff(19790404, 19820605, "y");  // returns 3
datediff(19790404, 19820605, "md"); // returns 1
datediff(19790404, 19820605, "ym"); // returns 2
datediff(19790404, 19820605, "yd"); // returns 62
```

Example 2

```
numeric date1 = 20090120;
numeric date2 = 20121106;

string outputText = "The difference between January 20, 2009 and November 6, 2012 is %d years,
%d months, and %d days";

errmsg(outputText, datediff(date1, date2, "y"), datediff(date1, date2, "ym"), datediff(date1, da
te2, "md"));

// returns ... 3 years, 9 months, and 17 days
```

See also: DateAdd Function, DateValid Function, SysDate Function

DateValid Function

Format

```
b = datevalid(date);
```

Description

The datevalid function determines whether a date is valid. The numeric expression ***date*** must be in YYYYMMDD (year/month/day) format.

Return Value

The function returns 1 (true) if the date is valid and 0 (false) otherwise.

Example

```
datevalid(20120229); // returns 1
datevalid(20130229); // returns 0
```

See also: DateAdd Function, DateDiff Function, SysDate Function

PublishDate Function

Format

```
i = publishdate();
```

Description

The publishdate function returns the date and time that the code was compiled. Generally, this is not particularly meaningful because the code is compiled right before it is executed. The exception to this is when you create compiled binary data entry applications. These files, with the extension .pen, are used to run applications on mobile devices. In the case of a .pen file, the value returned by the function will be the date and time that the .pen file was created.

Return Value

The function returns a number in the form YYYYMMddHHmmss , where YYYY is the year, MM is the month, dd is the day, HH is the hour, mm is the minute, and ss are the seconds.

Example

```
numeric publishDay = int(publishdate() / 1000000); // ignore the time

if datediff(publishDay,sysdate("YYYYMMDD")) > 20 then
    errmsg("This program expires after 20 days. Sync to get a new program.");
    stop(1);
endif;
```

See also: DateAdd Function, DateDiff Function, SysDate Function, SysTime Function

SetLB Function

Format

```
i = setlb(month, year, minimum_cmc);
```

Description

The setlb function returns the lower bound of the *century month code* (CMC) of an event. The arguments ***month***, ***year***, and ***minimum_cmc*** are all numeric expressions.

If the month and year are both valid, the function returns the CMC of that date, as if calling:

```
i = cmcode(month, year);
```

If the month is invalid but the year is valid, the function returns the CMC of January in the valid year, as if calling:

```
i = cmcode(1, year);
```

If both the month and year are invalid, the function returns the lower bound represented as ***minimum_cmc***.

Return Value

The function returns the lower bound of the CMC of an event.

Example

```
if CM12Y < 9997 then
    ldoblc = setlb(CM12M, CM12Y, 0);
    udoblc = setub(CM12M, CM12Y, 9999);
endif;
```

See also: SetUB Function, CMCode Function, AdjLBA Function, AdjLBI Function, AdjUBA Function, AdjUBI Function

SetUB Function

Format

```
i = setub(month, year, maximum_cmc);
```

Description

The setlb function returns the upper bound of the *century month code* (CMC) of an event. The arguments ***month***, ***year***, and ***maximum_cmc*** are all numeric expressions.

If the month and year are both valid, the function returns the CMC of that date, as if calling:

```
i = cmcode(month, year);
```

If the month is invalid but the year is valid, the function returns the CMC of December in the valid year, as if calling:

```
i = cmcode(12, year);
```

If both the month and year are invalid, the function returns the upper bound represented as ***maximum_cmc***.

Return Value

The function returns the upper bound of the CMC of an event.

Example

```
if CM12Y < 9997 then
    ldoblc = setlb(CM12M, CM12Y, 0);
    udoblc = setub(CM12M, CM12Y, 9999);
endif;
```

See also: SetLB Function, CMCode Function, AdjLBA Function, AdjLBI Function, AdjUBA Function, AdjUBI Function

SysDate Function

Format

```
i = sysdate([date_format[,timestamp_value]]);
```

Description

The `sysdate` function returns the current system date as an integer. The optional string expression ***date_format*** is composed of a combination of DD (days), MM (months), and/or YY or YYYY (years). YY returns the current year in two digits, while YYYY returns it in four digits. The strings DD, MM and YY or YYYY can be put together in any order to make up a customized format. If no date format is specified, the function returns the date in the format YYMMDD.

Passing an optional numeric expression, ***timestamp_value***, means that instead of using the system date, the function will use the date, in the local machine's time zone, of a date/time value that came from `timestamp` function.

The current date can be returned as a string using the `edit` function as follows:

```
edit("9999-99-99",sysdate("YYYYDDMM"));
```

Return Value

The function returns the system date as an integer. If the date format is invalid, the function returns 0.

Example

```
// if the current date is December 17, 1999...
x = sysdate("DDMMYYYY");    // returns 17121999
x = sysdate("MMYYYY");      // returns 121999
x = sysdate("DD");          // returns 17
x = sysdate();              // returns 991217

// if the current date is March 8, 2000...
x = sysdate("DDMMYYYY");    // returns 8032000
x = sysdate("MMYYYY");      // returns 32000
x = sysdate("MMYY");        // returns 300
x = sysdate("DD");          // returns 8
x = sysdate();              // returns 308
```

See also: DateAdd Function, DateDiff Function, DateValid Function, PublishDate Function, SysTime Function, Timestamp Function

SysTime Function

Format

```
i = systime([time_format[,timestamp_value]]);
```

Description

The `systime` function returns the current system time as a six-digit integer in the form HHMMSS, where HH is the hour, MM are the minutes, and SS are the seconds.

Optionally, a string expression ***time_format*** can be supplied, using a combination of HH, MM, and SS, to extract only part of the current system time.

Passing an optional numeric expression, ***timestamp_value***, means that instead of using the system time, the function will use the time, in the local machine's time zone, of a date/time value that came from `timestamp` function.

The current time can be returned as a string using the edit function as follows:

```
edit("99:99:99",systime());
```

Return Value

The function returns the system time as an integer.

Example

```
SURVEY_START_HOUR = systime("HH");
SURVEY_START_MINUTE = systime("MM");
```

See also: SysDate Function, Timestamp Function, Edit Function

Timestamp Function

Format

```
d = timestamp();
```

Description

The `timestamp` function returns the UNIX time, which is the number of seconds that have elapsed since January 1, 1970 in Coordinated Universal Time (UTC). This value is used by many computer programs and is thus useful for storing times as well as calculating timespans. Because the time is in UTC, it eliminates the need to worry about time zones when working with times.

Return Value

The function returns the UNIX time as a decimal value. The integer part of this value is the number of elapsed seconds, while the decimal part contains the milliseconds.

If you wish to store this value in a fixed-width numeric variable, the variable should have an integer length of 10 and a decimal length of 3 (for a total length of 13 without a decimal character, or 14 with one).

Example

```
PROC GLOBAL

numeric surveyStartTime;

PROC CROPS_QUEST

preproc

    surveyStartTime = timestamp();

    // if the survey began at 10:58pm on 12/28/2021 EST,
    // the UTC time (in seconds) would be 1640750243

    errmsg("Survey Start Time = %d", surveyStartTime);

postproc

    SURVEY_DURATION = timestamp() - surveyStartTime;
```

See also: TimeString Function, SysTime Function, SysDate Function, FileTime Function, SyncTime Function

TimeString Function

Format

```
s = timestring([timestamp_format][,timestamp_value]);
```

Description

The `timestring` function returns a formatted string containing the date and time as represented by a UNIX time, but using the local machine's time zone (instead of UTC).

The optional string expression ***timestamp_format*** contains date and time formatters. You can view a list of these formatters on a webpage about the C function strftime. If no formatter is supplied, the function returns an expression based on the machine's local settings using the formatter `"%c"` (note that the `"%c"` formatter is locale-dependent and so will return a slightly different layout depending on where it is invoked).

The optional numeric expression ***timestamp_value*** allows you to use a date/time value that came from the `timestamp` function. If not supplied, the function uses the current date/time.

If querying a SQL database via the `sqlquery` function, you can call this function with the name ***cspro_timestring***.

Return Value

The function returns the custom formatted date and time string.

Example

```
// if the current date (the default when no argument is given) is 12/27/2021,
// then the text displayed would be "Week 52"
errmsg("Week %W", timestring());

// if living in EST, the string will display "Fri May  5 08:00:00 2000"
numeric cspro_first_release_timestamp = 957528000;
errmsg("%s", timestring(cspro_first_release_timestamp));

// the following will display the string "Friday May 05, 2000"
errmsg("%s", timestring("%A %B %d, %Y", cspro_first_release_timestamp));
```

See also: Timestamp Function, SysTime Function, SysDate Function

External File Functions

Clear Function

Format

```
b = clear(dictionary_name);
```

Description

The `clear` function initializes the memory values of data items defined for the external or working storage dictionary ***dictionary_name***. Numeric items are initialized to `notappl` (blank) for external dictionaries and 0 for working storage dictionaries. Alphanumeric items are initialized to blank. The function will also set the number of occurrences of records and items to 0.

Return Value

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Example

```
clear(OTHERS_DICT);
enter OTHERS_FF;
```

See also: Array.Clear Function

Close Function

Format

```
b = close(dictionary_name || file_handler);
```

Description

The `close` function closes a file, whether opened as a dictionary or via a `file` statement. The function closes either a dictionary named ***dictionary_name*** or a text file associated with ***file_handler***.

If closing a dictionary, the dictionary will be set to a none data source and will remain set to that source until the previous file is reopened using the `open` function or if a new file is associated with the dictionary using the `setfile` function.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
file report_file;

// ...

close(report_file);

view(filename(report_file));
```

See also: Open Function, SetFile Function, File.Close Function

CountCases Function

Format

```
i = countcases(dictionary_name [where condition]);
```

Description

The `countcases` function counts the number of cases in a file referenced by ***dictionary_name***. The dictionary can be either the main input dictionary or an external dictionary. An optional logical ***condition***, referencing items in a dictionary, can be supplied to count the number of cases in the file that meet a certain criterion. When using this condition, the dictionary must be an external dictionary.

It is possible, by specifying dictionary access parameters after the dictionary name, to limit the cases that the `countcases` function processes.

Return Value

The function returns the number of cases in the file, or if a condition is given, the number of cases that meet that condition. If there is an error processing the file, the function returns `default`.

Example

```
PROC NEW_INTERVIEWER_NAME

    if countcases(STAFF_DICT where STAFF_NAME = NEW_INTERVIEWER_NAME) >= 1 then
        errmsg("You must give the interviewer a unique name.");
        reenter;
    endif;
```

See also: ForCase Statement, Find Function

CurrentKey Function

Format

```
s = currentkey(dictionary_name);
```

Description

The `currentkey` function returns a string containing the key of the case currently referenced by ***dictionary_name***. A dictionary key is a single string comprised of the IDs for a case. The `currentkey` is created using the current values of the case's IDs.

Return Value

The function returns a string containing the current key.

Example

```
PROC LOCATION

    errmsg("Confirm that the farm stand's identifier is: %s", currentkey(FARM_STAND_DICT))
    select("Correct", continue,
           "Incorrect", reenter);
```

Differences Between Key and CurrentKey

The key and currentkey functions will often return the same key string, but when modifications are made to ID values, the key will not change whereas currentkey will reflect the modifications. For example:

```
loadcase(DICT_NAME, "0102");

key(DICT_NAME);        // 0102
currentkey(DICT_NAME); // 0102

ID1 = 99;
ID2 = 8;

key(DICT_NAME);        // 0102
currentkey(DICT_NAME); // 9908
```

See also: Key Function

DelCase Function

Format

```
b = delcase(dictionary_name⌈, id1, ..., idN⌋);
```

Description

The delcase function marks a case for deletion in an external file.

The ***dictionary_name*** must be supplied and refers to an external dictionary associated with your application.

The optional list of ID variables, ***id1*** ... ***idN***, specifies the list of variables that will identify the case to delete from the external file. Generally, each of the variables in the list is defined in a dictionary. The combined length of the variables in the list must equal the length of the case IDs defined for the external dictionary. The function concatenates the variables in the ID list to form a string. Alternatively, you can pass, as a string, the full key of the case to delete. The function then deletes from the external file the case whose case identifier matches the string constructed from the list.

If no ID list is provided, the current values of the ID items for the external dictionary are used.

Return Value

The function returns a logical value of 1 (true) if the case existed and was successfully marked for deletion and 0 (false) otherwise.

Example

```
delcase(ASSIGNMENT_DICT, CLUSTER, HH);
```

See also: LoadCase Function, WriteCase Function

DirCreate Function

Format

```
b = dircreate(directory_name);
```

Description

The `dircreate` function creates a new directory whose name comes from the string expression ***directory_name***. If necessary, parent directories will be added to support the creation of the directory.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise. If the directory already exists, the function returns 1.

Example

```
string backupDirectoryName = pathname(InputFile) + "Backups";
dircreate(backupDirectoryName);
```

See also: DirExist Function, FileCreate Function

DirDelete Function

Format

```
i = dirdelete(directory_name);
```

Description

The `dirdelete` function deletes an empty directory or group of empty directories. The argument ***directory_name*** is a string expression giving a directory name.

You can use the wildcard characters "*" and "?" to specify a group of directories to delete.

Return Value

The function returns the number of directories deleted. If there was an error deleting directories, the function returns `default`.

Example

```
// delete directory "Backups" from the application folder
dirdelete(pathname(Application) + "Backups");

// delete all directories that begin with "Backup" from the application folder
dirdelete(pathname(Application) + "Backup*");
```

See also: DirCreate Function, DirExist Function, DirList Function, FileDelete Function

DirExist Function

Format

```
b = direxist(directory_name);
```

Description

The `direxist` function determines whether a directory exists. ***directory_name*** is a string expression.

Return Value

The function returns a logical value of 1 (true) if the directory exists and 0 (false) otherwise.

Example

```
if direxist("C:\Backups") then
    // process backups
endif;
```

See also: DirCreate Function, FileExist Function

DirList Function

Format

```
b = dirlist(list_to_receive_paths, directory_path
            [, filter := wildcard]
            [, recursive := recursive_flag]);
```

Description

The `dirlist` function fills ***list_to_receive_paths***, a string list, with the paths of files and directories located in a directory specified by ***directory_path***. The paths included in the list are fully evaluated absolute paths. Hidden and system files are not included in the directory listing.

The ***directory_path*** can be one of the following:

- A string expression specifying a directory.
- One of the path types documented in the `pathname` function.
- A media type, which will list the files of the specified media type on Android. These files can reside in multiple directories.

Using named arguments, you can specify a string expression, ***wildcard***, that is applied to filter the paths added to the string list. For example, `"*.csdb"` would include only CSPro DB files. You can specify multiple wildcard filters by separating each filter with a semicolon. The filter `"*.jpg;*.jpeg;*.bmp"` would include both JPEG and BMP files. Instead of using a string expression, you can also use a predefined wildcard expression:

- `FileType.Audio`: Files that the audio object can read and write.
- `FileType.Geometry`: Files that the geometry object can read and write.
- `FileType.Image`: Files that the image object can read and write.

By default, only files and directories located in the directory are added to the list. To recursively add paths, creating a listing of the specified directory as well as any subdirectories, set the numeric expression ***recursive_flag*** to `true`.

Return Value

The function returns a logical value of 1 (`true`) if successful and 0 (`false`) if the directory does not exist or cannot be accessed. If the list is read-only, the function returns `default`.

Example 1

```
write("Shapefile and tile packages in folder and subfolders:");

list string map_files;

dirlist(map_files, "C:\Survey\Maps", filter := "*.shp;*.tpk", recursive := true);

do numeric ctr = 1 while ctr <= map_files.length()
    write("%s", map_files(ctr));
enddo;
```

Example 2

```
// get a list of all shared images on Android that can be loaded using the image object
list string image_files;

dirlist(image_files, Media.Images, filter := FileType.Image);
```

See also: Path.SelectFile Function, DirCreate Function, DirExist Function

FileConcat Function

Format

```
b = fileconcat(dictionary_name, output_file_name, file_name1, ..., file_nameN);
```

Description

The `fileconcat` function concatenates a number of files into a single file. The files to concatenate can be either individual file names or wildcard file specifications. The function supports two different methods of concatenation:

- **Concatenate cases**: Reads cases from each input file and writes them to the output file, skipping invalid and duplicate cases. This mode requires a data dictionary that describes the format of the input and output data files so that cases may be validated and duplicates may be identified. Concatenate cases mode may be used with all types of CSPro data files.
- **Concatenate file contents**: Simply appends one file to the end of another regardless of the file contents. This mode will not work for CSPro data files that are not text files (such as CSPro DB files).

To peform a **case** concatenation, you must specify a ***dictionary_name*** that specifies the layout of the data files being concatenated. When the dictionary is specified, the files are combined by reading each case from each of the input files and writing them to the output file. In this mode, invalid cases and duplicate cases are automatically removed from the output file.

To perform **content** concatenation, do not specify a dictionary. In this case, the input files are appended together without examining their contents. This will be faster but will not eliminate duplicates or invalid cases.

The ***output_file_name*** is a string expression giving the name of the output file that will contain the concatenated data. The arguments ***file_name1*** to ***file_nameN*** are either string expressions or string lists that contain the names of specific files or a wildcard specification of a group of files.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example 1

```
fileconcat("DMV.txt", "District of Columbia.txt", "Maryland.txt", "Virginia.txt");
```

Example 2

```
fileconcat(CENSUS_DICT, "14combined.csdb", "../Data/14*.csdb");
```

See also: FileCopy Function

FileCopy Function

Format

```
i = filecopy(file_name, output_file_name);
```

Description

The `filecopy` function copies one file to another file or copies a group of files to a folder. The ***file_name*** is a string expression or string list giving the source file name(s). The ***output_file_name*** is a string expression giving the destination file or folder name. The arguments can also be file handlers declared in `file` statements.

You can use the wildcard characters "*" and "?" to specify a group of files to copy as the ***file_name***. If using wildcard characters, the ***output_file_name*** must be the name of a folder.

Return Value

The function returns the number of files copied. If there was an error copying files, the function returns `default`.

Example 1

```
filecopy("Virginia.csdb", "Virginia_Backup.csdb");
```

Example 2

```
string backup_folder_name = pathconcat(Application, maketext("Backup%d", timestamp()));
dircreate(backup_folder_name);

filecopy(pathconcat(Application, "*.csdb"), backup_folder_name);
```

See also: FileRename Function, FileConcat Function

FileCreate Function

Format

```
b = filecreate(file_name);
```

Description

The `filecreate` function creates a new file named after the string expression (or `file` handler) ***file_name***. If the file already exists, it is truncated to length three (the bytes in the file will consist only of the UTF-8 byte order mark).

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
string report_filename = pathconcat(Application, "EA Completion Rate.html");
filecreate(report_filename);
```

See also: DirCreate Function, Open Function, SetFile Function

FileDelete Function

Format

```
i = filedelete(file_name);
```

Description

The `filedelete` function deletes a file or group of files. The argument ***file_name*** is either a string expression or string list giving a file name or names, or is a file handler declared in a `file` statement.

You can use the wildcard characters "*" and "?" to specify a group of files to delete.

Return Value

The function returns the number of files deleted. If there was an error deleting files, the function returns `default`.

Example

```
// delete file "Household.csdb" from the backups folder
filedelete(pathconcat(Application, "Backups/Household.csdb"));

// delete all data from the backups folder
filedelete(pathconcat(Application, "Backups/*.csdb"));
```

See also: FileCopy Function, FileRename Function, DirDelete Function

FileEmpty Function

Format

```
b = fileempty(file_name);
```

Description

The `fileempty` function determines whether or not a file is empty and has no content. This function is useful for checking the status of files without needing to know the text encoding. An empty ANSI file is 0 bytes but an empty UTF-8 file is 3 bytes. The argument ***file_name*** is either a string expression giving a file name or is a file handler declared in a `file` statement in `PROC GLOBAL`.

Return Value

The function returns a logical value of 1 (true) if the file exists and is empty and 0 (false) if the file is not empty. If the file does not exist, the function returns `default`.

Example

```
if fileempty(customReportFile) then
    // print the header
    filewrite(customReportFile,"SECONDS,KEYSTROKES");
endif;
```

See also: FileExist Function, FileSize Function

FileExist Function

Format

```
b = fileexist(file_name);
```

Description

The `fileexist` function determines whether a file exists. ***file_name*** is either a string expression giving a file name or is a file handler declared in a `file` statement in `PROC GLOBAL`.

You can use the wildcard characters "*" and "?" in the ***file_name***. In that case the function will determine whether any file matching the specification exists.

Prior to CSPro 6.2, the function determined whether either a file or a directory existed. The `direxist` function must now be used to check for the existence of a directory.

Return Value

The function returns a logical value of 1 (true) if the file exists and 0 (false) otherwise.

Example

```
if fileexist(householdPffFilename) then
    filedelete(householdPffFilename);
endif;
```

See also: FileEmpty Function, FileSize Function, DirExist Function

FileName Function

Format

```
s = filename(dictionary_name || file_handler || pff_name || object_name || paradata);
```

Description

The `filename` function returns the fully qualified name of a file. If supplying a ***dictionary_name***, the function returns the name of the data file associated with a data dictionary. If providing a ***file_handler*** (declared in a `file` statement), the function returns the name of the text file associated with the file handler.

If providing a ***pff_name*** (declared in a `pff` statement), the function returns the name of an existing PFF file, or the name of a temporarily created PFF file. If a PFF file was loaded (via `pff.load`) and not modified, the name of that PFF file is returned. If the PFF has been modified and not saved, then calling `filename` will save the PFF to a temporary file and the name of that file is returned.

For several other CSPro objects, if passed an ***object_name***, the function returns the file name associated with the content of the object. This functionality works on audio, document, geometry, image, and report objects.

The function can also take ***paradata*** as its argument, in which case it returns the file name of the currently open paradata log.

Return Value

The function returns a string containing the folder and file name.

Example

```
filewrite(report_file, "Report file: %s", filename(report_file));
filewrite(report_file, "Census data file: %s", filename(CENSUS_DICT));
```

See also: Path Namespace, PathName Function, FileExist Function, FileSize Function

FileRead Function

Format

```
b = fileread(file_handler, string_variable || string_list);
```

Description

The `fileread` function reads one or more lines of text from the file associated with ***file_handler***. After the read the file pointer is positioned to the next line in the file. This function reads lines sequentially.

If a ***string_variable*** is provided as a argument, a single line of text is read and placed in the variable, which is either a temporary `string` variable or an alphanumeric dictionary item. If the dictionary item is longer than the line of text, blanks will be added at the end. If the item is shorter, the line of text will be truncated.

Alternatively, a string list can be used as an argument. In this case, all remaining lines in the file are read and stored in ***string_list***.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
PROC GLOBAL

file OperatorIdOverrideFile;

PROC APPLICATION_FF

preproc

    string operatorid_override;

    if fileread(OperatorIdOverrideFile, operatorid_override) then
        setoperatorid(operatorid_override);
    endif;
```

See also: FileWrite Function, File.Read Function

FileRename Function

Format

```
b = filerename(old_file_name, new_file_name);
```

Description

The `filerename` function changes the name of a file or group of files on the disk. The argument ***old_file_name*** is a string expression or string list giving the existing file name(s). The argument ***new_file_name*** is a string expression giving the new file name or destination folder name. The arguments can also be file handlers declared in a `file` statement.

You can use the wildcard characters "*" and "?" to specify a group of files to rename as the ***old_file_name***. If using wildcard characters, the ***new_file_name*** must be the name of a folder.

Return Value

The function returns the number of files renamed. If there was an error renaming files, the function returns `default`.

Example

```
filerename("data.csdb", maketext("old_data%d.csdb",timestamp()));
```

See also: FileCopy Function, FileDelete Function

FileSize Function

Format

```
i = filesize(file_name);
```

Description

The `filesize` function returns the size of a file in bytes. The argument ***file_name*** is either a string expression giving a file name or is a file handler declared in a `file` statement in `PROC GLOBAL`.

Return Value

If the file exists, the function returns the size of the file in bytes. If the file does not exist, the function returns `default`.

Example 1

```
if filesize("report.txt") > 0 then
    errmsg("The report already exists!");
endif;
```

Example 2

```
PROC GLOBAL

file reportFile;

PROC EXAMPLE

    numeric reportSize = filesize(reportFile);
```

See also: FileEmpty Function, FileExist Function

FileTime Function

Format

```
d = filetime(file_name);
```

Description

The `filetime` function returns the last modified date and time of a file or a directory. The argument ***file_name*** is either a string expression giving a file or directory name or is a file handler declared in a `file` statement in `PROC GLOBAL`.

Return Value

If the file or directory exists, the function returns a timestamp value indicating the last modified date and time. If the file or directory does not exist, the function returns `default`.

Example

```
list string photos_list;
dirlist(photos_list, "../Photos", "*.jpg");

do numeric counter = 1 while counter <= photos_list.length()

    // only sync photos that weren't previously synced
    if filetime(photos_list(counter)) > last_sync_timestamp then
        syncfile(PUT, photos_list(counter), "/Photos/");
    endif;

enddo;
```

See also: Timestamp Function, TimeString Function

FileWrite Function

Format

```
b = filewrite(file_handler, message[, argument1, ..., argumentN]);
```

Description

The `filewrite` function writes one or more lines of text to the file associated with ***file_handler***.

The ***message*** is either a string expression or a numeric message number that contains the text that is written to the file. If the text contains any message formatters, the optional arguments ***argument1*** to ***argumentN*** will be inserted into the text. There are some additional options for file output:

- If you want to break a line of text into two lines, place **\n** where you want the line divided.
- If you want a text line to begin on a new page, place **\f** at the beginning of the text string.
- To output **\n** or **\f** as text instead of a new line or a new page, use a double backslash (e.g., **\\n**).

Alternatively, the ***message*** can be a string list. If a list is provided, then each string contained in the list is written to the file, allowing for the output of multiple lines of text with one function call.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
PROC GLOBAL

file CsvFile;

PROC EXAMPLE

    filewrite(CsvFile, "Name,Sex,Age");

    do numeric counter = 1 while counter <= totocc(PERSON_REC)
        filewrite(CsvFile, "%s,%d,%d", encode(CSV, NAME(counter)), SEX(counter), AGE(counter));
    enddo;
```

See also: FileRead Function, File.Write Function, Write Function, Message Formatting Options, Encode Function

Find Function

Format

```
b = find(dictionary_name, relational_operator, key_prefix);
```

Description

The `find` function determines the existence of a case in a data file that matches a specified condition. The function searches the index of a file and determines whether any case matches the specified condition. The position in the file is not changed, unlike the similar `locate` function, and no case is loaded into memory.

The ***dictionary_name*** refers to either an external dictionary or the main input file of an entry or batch application.

The ***relational_operator*** is one of the following: =, <, <=, >, >=, or `startswith`.

The string expression ***key_prefix*** specifies the condition to use when searching for cases. Cases with a key that begins with or equals the ***key_prefix*** are considered eligible cases.

An additional option, using `uuid` instead of a relational operator, determines the existence of a case based on the case's UUID, not the case's key. This may be useful when looking up duplicate or deleted cases.

Return Value

The function returns a logical value of 1 (true) if a case is found and 0 (false) otherwise.

Example

```
PROC OCCUPATION

    if not find(OCCUPATION_CODES_DICT, =, maketext("%04d", OCCUPATION)) then
        errmsg("The occupation %d is not valid.", OCCUPATION);
        reenter;
    endif;
```

See also: Locate Function, CountCases Function

Key Function

Format

```
s = key(dictionary_name);
```

Description

The key function returns a string containing the key of the case currently loaded in the file referenced by ***dictionary_name***. A dictionary key is a single string comprised of the IDs for a case. The key is created using the case's IDs as they existed when the case was last loaded or written..

Return Value

The function returns a string containing the key. If there has been no previous activity on the file and no key has been established, the key function returns a blank string.

Example

```
// do not output any cases with IDs starting with 04
if key(CENSUS_DICT)[1:2] = "04" then
    skip case;
endif;
```

Differences Between Key and CurrentKey

The key and currentkey functions will often return the same key string, but when modifications are made to ID values, the key will not change whereas currentkey will reflect the modifications. For example:

```
loadcase(DICT_NAME, "0102");

key(DICT_NAME);        // 0102
currentkey(DICT_NAME); // 0102

ID1 = 99;
ID2 = 8;

key(DICT_NAME);        // 0102
currentkey(DICT_NAME); // 9908
```

See also: CurrentKey Function, KeyList Function, Key PFF Attribute, GetCaseLabel Function

KeyList Function

Format

```
i = keylist(dictionary_name⌈, key_listing⌋);
```

Description

The keylist function fills ***key_listing***, an optional string list, with the keys in the file referenced by ***dictionary_name***. The dictionary can be either the main input dictionary or an external dictionary. A dictionary key is a single string comprised of the IDs for a case.

It is possible, by specifying dictionary access parameters after the dictionary name, to limit the keys that the keylist function returns.

Return Value

The function returns the number of cases in file.

Example 1

```
PROC ACTION

    if ACTION in 1,3,6 and keylist(CENSUS_DICT) = 0 then // actions that require cases
        errmsg("No cases have been entered yet.");
        reenter;
    endif;
```

Example 2

```
PROC HOUSEHOLD_NUMBER

    list string enteredKeys;

    keylist(CENSUS_DICT, enteredKeys);

    do numeric keyCtr = 1 while keyCtr <= enteredKeys.length()

        if HOUSEHOLD_NUMBER = tonumber(enteredKeys(keyCtr)[9:2]) then
            errmsg("The household number %d has already been entered.", HOUSEHOLD_NUMBER);
            reenter;
        endif;

    enddo;
```

See also: Key Function

LoadCase Function

Format

```
b = loadcase(dictionary_name[, id1, ..., idN]);
```

Description

The `loadcase` function reads a specified case from an external file into memory. Once the case is loaded, all variables defined in the corresponding dictionary are available for use.

The ***dictionary_name*** must be supplied and refers to an external dictionary associated with your application.

The optional list of ID variables, ***id1*** ... ***idN***, specifies the list of variables that will identify the case to load from the external file. This process is similar to matching files on the basis of key variables in statistical and database software packages. Generally, each of the variables in the list is defined in a dictionary. The combined length of the variables in the list must equal the length of the case IDs defined for the external dictionary. The function concatenates the variables in the ID list to form a string. Alternatively, you can pass, as a string, the full key of the case to load. The function then loads from the external file the case whose case identifier matches the string constructed from the list.

If no ID list is provided, the next logical case in the external file will be loaded. The determination of what is "next" is based on the dictionary's access parameters.

(In versions of CSPro prior to 7.0, calling this function would automatically `open` an external file. The function no longer does this, so if you call `close` on an external dictionary, you must `open` it manually before calling this function.)

Return Value

The function returns a logical value of 1 (true) if the case was loaded successfully and 0 (false) otherwise.

Example

```
if loadcase(SAMPLE_DICT, CLUSTER, HH) then
    WEIGHT = SAMPLE_WEIGHT;
endif;
```

See also: Retrieve Function, WriteCase Function

Locate Function

Format

```
b = locate(dictionary_name, relational_operator, key_prefix);
```

Description

The `locate` function finds, but does not load, a case in a data file that matches a specified condition. The function searches the index of a file and finds the first case that matches the specified condition. The case pointer is positioned to the case's location, but the case is not loaded into memory. To load the case into memory, use the `retrieve` function or the `loadcase` function without arguments.

The ***dictionary_name*** refers to either an external dictionary or the main input file of a batch application.

The ***relational_operator*** is one of the following: =, <, <=, >, >=, or `startswith`.

The string expression ***key_prefix*** specifies the condition to use when searching for cases. Cases with a key that begins with or equals the ***key_prefix*** are considered eligible cases. If the relational operators are < or <=, then the file is positioned at the case with the largest key which satisfies the condition. If the relational operators are > or >=, then the file is positioned at the case with the smallest key which satisfies the condition. If `startswith` is used, the file is positioned at the first case whose key starts with the ***key_prefix***.

An additional option, using `uuid` instead of a relational operator, determines the existence of a case based on the case's UUID, not the case's key. This may be useful when looking up duplicate or deleted cases.

Return Value

The function returns a logical value of 1 (true) if a case is found and 0 (false) otherwise.

Example

```
if locate(SURVEY_DICT, >=, "55") then

    do while loadcase(SURVEY_DICT) and PROVINCE = 55
        // process all cases in province 55
    enddo;

endif;
```

See also: Find Function, Set First Statement, Set Last Statement, Retrieve Function, LoadCase Function

NMembers Function

Format

```
i = nmembers(dictionary_name [mark_type]);
```

Description

The `nmembers` function returns the number of cases selected by an operator during a `selcase` function call.

The ***dictionary_name*** must be supplied and refers to an external dictionary associated with your application. An optional argument, ***mark_type***, indicates what kinds of cases to process, and can be one of the following values:

mark_type	Description
marked	Process only the cases selected by the operator during the `selcase` function call.
unmarked	Process the cases **not** selected by the operator.
all	Process all of the cases that could have been selected by the operator.

If not specified, the ***mark_type*** defaults to **marked**.

Return Value

The function returns the number of cases that are described by the ***mark_type*** query.

Example

```
while 1 do

    selcase(OCCUPATION_DICT,"Plantation") multiple;

    if nmembers(OCCUPATION_DICT) < 3 then
        errmsg("You must select at least three options.");

    else
        break;

    endif;

enddo;
```

See also: SelCase Function, For (Dictionary) Statement

Open Function

Format:

```
b = open(ext-dict-name | file-name[, update | append | create]);
```

Description:

The **open** function opens and keeps open an external file or file declared in a **file** statement.

Under most circumstances, neither an **open** nor a **close** function is necessary to manipulate the file. In data entry applications, by default, the file is opened when it is operated on with a file function, such as **loadcase**, **writecase**, **readfile**, or **writefile** and closed immediately afterward. In batch applications, by default, the file is opened at the beginning of the run and closed at the end.

If you want to control the opening and closing of the file, you can use the **open** and close functions to do this. If you code an **open** function anywhere is the application logic, then you must control all opening and closing of the file

The *ext-dict-name* or *file-name* must be supplied. *Ext-dict-name* is the name of a data dictionary defining an external data file. *File-name* is the name of a file declared in a **file** statement.

The keywords **update**, **append** or **create** are optional. If no keyword is coded, the file is opened in **update** mode.

When an *ext-dict-name* is used, if **update** or **append** is used, the file is opened, its contents are not changed, and the file is ready to update cases. If **create** is coded, the file is opened, all previous records are removed and the file is ready to add cases.

When a *file-name* is used, if **update** is used, the file is opened and you are positioned at the beginning of the file. If **append** is coded, the file is opened, its contents are not changed, and you are positioned at the end of the file. If **create** is coded, the file is opened, all previous records are removed and you are positioned at the beginning of the file.

Return value:
The function returns a logical value of 1 (true) if file is opened and 0 (false) otherwise.

Example 1:
```
OK = open(LOOKUP);
```

Example 2:
```
OK = open(REPORT, append);
```

Retrieve Function

Format

```
b = retrieve(dictionary_name);
```

Description

The `retrieve` function reads a case into memory from the current position of an external file. It is intended for use only after a successful execution of the `locate` function. The ***dictionary_name*** must refer to an external dictionary.

Return Value

The function returns a logical value of 1 (true) if a case is retrieved and 0 (false) otherwise.

Example

```
if locate(SURVEY_DICT,>=,"55") and retrieve(SURVEY_DICT) then
    // process the case
endif;
```

See also: Locate Function, LoadCase Function

Set Access Statement

Format

```
set access(dictionary_name, type.value[, type.value, ...]);
```

Description

The `set access` statement is used to control the way that CSPro iterates through a data file. By default, CSPro processes all non-deleted cases in file order for the main input file of a batch application or in ascending key order (e.g., A -> B -> C) for all other files. With this statement, you can change the default behavior to process cases in a different order or to control what cases are processed.

The ***dictionary_name*** refers to either an external dictionary or the main input file of a batch application. If the dictionary access parameters are modified while in the middle of iterating through the data file, the next case loaded will be based on the location of the currently loaded case. For example, if you are running a batch application on cases 1 to 5, and then after processing case 3 you change the order to descending, the next cases loaded will be 2 and 1.

Dictionary Access Parameters

Type	Values	Description
OrderType	*Indexed*	Iterate based on the alphabetical value of the case's key (default for external dictionaries)
	Sequential	Iterate based on the case's position in the file (default for batch inputs)
Order	*Ascending*	Iterate from low to high value (key or file position) (default)
	Descending	Iterate from high to low value (key or file position)
CaseStatus	*All*	Iterate through all cases, including deleted cases
	NotDeleted	Iterate through all non-deleted cases (default)
	Partial	Iterate through non-deleted partially saved cases
	Duplicate	Iterate through non-deleted cases where there are at least two cases with the same key in the file

Temporarily Modifying the Dictionary Access Parameters

Using `set_access`, the dictionary access parameters are modified until the next call to `set_access`. However, it is also possible to temporarily change the dictionary access parameters when using several functions or statements. By specifying the parameters in parentheses after the dictionary name in uses of `countcases`, `forcase`, `keylist`, and `selcase`, the access to the dictionary will only be changed for the duration of that routine.

An additional parameter that can be used with these functions is `startswith`, functionality that allows for partial iteration of only cases with keys that start with the specified key prefix.

Example 1

```
PROC CENSUS_FF

preproc

    // process the batch input in order from the last province to the first
    set_access(CENSUS_DICT, OrderType.Indexed, Order.Descending);
```

Example 2

```
forcase CENSUS_DICT(CaseStatus.Partial) do
    interview_vs.add("Continue interview: " + getcaselabel(CENSUS_DICT), key(CENSUS_DICT));
endfor;
```

Example 3

```
// only process assignments in the area where the supervisor is working
forcase GEOCODES_DICT(startswith, strip(MENU_GEOCODE_SELECTION)) do
    // ...
endfor;
```

See also: Set First Statement, Set Last Statement, Locate Function

SetFile Function

Format

```
b = setfile(dictionary_name || file_handler, file_name[, update || append || create]);
```

Description

The `setfile` function assigns a data file to the dictionary associated with ***dictionary_name*** or associates the file ***file_handler*** to a text file on the disk. The ***file_name*** argument is a string expression containing the file name of the file to be associated with the dictionary or file. With dictionaries, the ***file_name*** can also be a connection string.

Using the keyword ***update***, ***append*** or ***create*** is optional. If no keyword is used, the file is opened in ***update*** mode.

Dictionary behavior: If ***update*** or ***append*** is used, the data file's contents are not changed and the file is ready to update cases. If ***create*** is used, all previous cases are removed and the file is ready to add cases.

File behavior: If ***update*** is used, you are positioned at the beginning of the file. If ***append*** is used, the file's contents are not changed and you are positioned at the end of the file. If ***create*** is used, all previous content in the file is removed and you are positioned at the beginning of the file.

For both dictionaries and files, if ***create*** or ***append*** is used and the file does not already exist, a new empty file will be created. If ***update*** is used and the file does not already exist, the function will fail and return 0.

Return Value

The function returns a logical value of 1 (true) if the new file is successfully assigned and 0 (false) otherwise.

Dictionary Example

```
if REGION = 1 then
    setfile(GEOCODES_DICT, "Eastern.csdb");
else
    setfile(GEOCODES_DICT, "Western.csdb");
endif;
```

File Example

```
file household_report_file;

setfile(household_report_file, "Household Report.txt", create);
```

See also: File Statement, FileName Function, SetOutput Function, File.Open Function

Set First Statement

Format

```
set first(dictionary_name);
```

Description

The `set first` statement is used, like the `locate` function, to find a specific case in a data file. The case pointer is positioned to the first case. Based on the dictionary access parameters, the case is either the first case by indexed (key) order or by sequential (file) order. To load the case into memory, use the `retrieve` function or the `loadcase` function without arguments.

The ***dictionary_name*** refers to either an external dictionary or the main input file of a batch application. Using this statement with the main input file of a batch application allows you to pass through the same data file more than once in a single run of a batch application.

Example

```
PROC MENU

    set first(SURVEY_DICT);

    do while loadcase(SURVEY_DICT)
        // process the cases in ascending key order
    enddo;
```

See also: Set Last Statement, Locate Function, Set Access Statement

Set Last Statement

Format

```
set last(dictionary_name);
```

Description

The `set last` statement is used, like the `locate` function, to find a specific case in a data file. The case pointer is positioned to the last case. Based on the dictionary access parameters, the case is either the last case by indexed (key) order or by sequential (file) order. To load the case into memory, use the `retrieve` function or the `loadcase` function without arguments.

The ***dictionary_name*** refers to either an external dictionary or the main input file of a batch application.

Example

```
PROC MENU

    set access(SURVEY_DICT, Order.Descending);

    set last(SURVEY_DICT);

    do while loadcase(SURVEY_DICT)
        // process the cases in descending key order
    enddo;
```

See also: Set First Statement, Locate Function, Set Access Statement

Write Function

Format

```
b = write(message || msg_num[, argument1, ..., argumentN]);
```

Description

The write function writes text to a write file that can be used as a report. Each argument is sequentially inserted into the write string. Arguments can be numeric or alphanumeric expressions, but the type of the argument must match the type of the receiving field in the message text. If no write file is specified at run time, the write file lines are placed in the default data entry or batch error report.

In the string expression:

%**[n]d** = Insert a number and display it as an integer
%**[n.d]f** = Insert a number and display it as a decimal value
%**[n.d]s** = Insert a text string

where "**n**" is the size of the field and "d" is the number of decimal places to show for a number. *For a complete list of possible message formatters, see Message Formatting Options*.

Numbers are never truncated. Text strings are truncated only if "**.d**" is used. If "**n**" is positive, the insert is right-justified in the size of the field. If "**n**" is negative, the insert is left-justified in the size of the field. If "**n**" is a positive number with a leading zero, the insert is right-justified in the size of the field and zero-filled to the left. When inserting a number, if "**n**" is preceded by a "+", the sign of the number is always displayed.

Return Value

The function returns a logical value 1 (true) if successful and 0 (false) otherwise.

Example 1

```
write("Sex = %d", SEX);
```

Example 2

```
numeric integerValue = 23456;

errmsg( "%d"        , integerValue);      //  23456
errmsg( "%-10d"     , integerValue);      //  23456
errmsg( "%10d"      , integerValue);      //       23456
errmsg( "%+10d"     , integerValue);      //      +23456
errmsg( "%+010d"    , integerValue);      //  +000023456
errmsg( "%-010d"    , integerValue);      //  0000023456
errmsg( "%f"        , integerValue);      //  23456.000000  Note the usage of %f


numeric decimalValue = 12.567;

errmsg( "%f"         , decimalValue);    //   12.567000
errmsg( "%-10.3f"    , decimalValue);    //   12.567
errmsg( "%d"         , decimalValue);    //   12           Note the usage of %d
errmsg( "%10.2f"     , decimalValue);    //        12.57
errmsg( "%10.3f"     , decimalValue);    //       12.567
errmsg( "%+10.3f"    , decimalValue);    //      +12.567
errmsg( "%+010.3f"   , decimalValue);    //   +00012.567
errmsg( "%010.3f"    , decimalValue);    //   000012.567
errmsg( "%10.5f"     , decimalValue);    //     12.56700


string stringValue = "abcdef";

errmsg( "%s"         , stringValue);     //   abcdef
errmsg( "%-10s"      , stringValue);     //   abcdef
errmsg( "%-10.3s"    , stringValue);     //   abc
errmsg( "%10s"       , stringValue);     //       abcdef
errmsg( "%10.3s"     , stringValue);     //          abc
```

See also: FileWrite Function, LogText Function, Message Formatting Options

WriteCase Function

Format

```
b = writecase(dictionary_name);
```

Description

The `writecase` function writes a case from memory to an external data file. It can be used to update existing cases or to

write new ones.

The ***dictionary_name*** must be supplied and refers to an external dictionary associated with your application.

After a case is written to an external file, the dictionary variables for that case remain in memory. If the application does not assign new values to all variables in the external dictionary before the next `writecase` function is executed, then the values from the previous case will be written to the external data file. Use the `clear` function to reset the values of these variables.

No Index Mode

In a batch application it is possible to have the function output cases without updating the file's index. This allows the external data file to have duplicate cases, i.e., cases sharing the same set of IDs. This may be useful if the batch application is a tool to reformat large sets of data, a situation in which maintaining the file's index is very time consuming. Since no check is done for duplicates you should be certain that duplicates cannot be generated, or that you want duplicates in the data file. You cannot use other external file functions like `loadcase` or `retrieve` if using no index mode. To use this special mode, in the function arguments write (noindex) after the dictionary name.

```
b = writecase(dictionary_name(noindex));
```

Return Value

The function returns a logical value of 1 (true) if the write is successful and 0 (false) otherwise.

Example

```
WORK_RECORD_ID = HOUSEHOLD_ID;
LAST_WORK_TIME = timestamp();

writecase(WORK_RECORD_DICT);
```

See also: LoadCase Function, DelCase Function

Synchronization Functions

SyncConnect Function

Format

```
b = syncconnect(differing_arguments);
```

Description

The `syncconnect` function opens a connection to a server to start a synchronization session. The function has four forms based on the type of connection, with each form described on a separate page:

- CSWeb
- Bluetooth
- Dropbox
- FTP
- LocalDropbox
- LocalFiles

After a successful connection, call `syncdata` and `syncfile` to synchronize data and files between the device and the server. When finished transferring data, call `syncdisconnect` to close the connection.

Return Value

The function returns a logical value of 1 (true) if the connection was successful and 0 (false) otherwise.

See also: Synchronization Overview, SyncDisconnect Function, SyncServer Function, SyncData Function, SyncFile Function, SyncParadata Function

SyncConnect Function (CSWeb)

This page describes the `syncconnect` function when connecting to a CSWeb server. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(CSWeb,csweb_url[,username[,password]]);
```

Description

The `syncconnect` function opens a connection to a CSWeb server to synchronize data and files over the Internet. The first argument must be the keyword ***CSWeb***. The second argument, ***csweb_url***, is a string expression of the full URL of the CSWeb server. The third and fourth arguments, ***username*** and ***password*** are optional string arguments of the credentials used to login to the server. If the username and password are not specified, the user will be prompted to enter them the first time the synchronization is run. After the credentials have been entered the first time, they are saved in secure storage on the device and do not need to be entered again. You can use the CSPro Settings dialog to clear any cached credentials.

For more information about CSWeb servers, see the Synchronization Overview and the CSWeb help documentation.

Example

```
string ServerUrl = "https://www.myserver.org/api";

// connect to the CSWeb server
if syncconnect(CSWeb, ServerUrl) then

    // send the latest cases to the CSWeb server
    syncdata(PUT, SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncConnect Function (Bluetooth), SyncConnect Function (Dropbox), SyncConnect Function (FTP), SyncConnect Function (LocalDropbox), SyncConnect Function (LocalFiles)

SyncConnect Function (Bluetooth)

This page describes the `syncconnect` function when connecting, via Bluetooth, to another device running a CSPro application. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(Bluetooth[, server_device_name]);
```

Description

The `syncconnect` function opens a connection, via Bluetooth, to another device (a computer, laptop, tablet, or phone) running a local server. The first argument must be the keyword ***Bluetooth***. The second argument, ***server_device_name***, is an optional string expression of the name of the device to which you want to connect. For Android devices, the device name can be found in Bluetooth settings on the device running the server. For Windows devices, this is the PC name that can be found in Windows settings. The `getbluetoothname` function can be used on the other device to find the device name. If no server device name is specified, a list of nearby devices is displayed and the operator may choose which one to connect to.

In order for the two devices to connect successfully, the device running the server must call the function `syncserver` at roughly the same time as the call to `syncconnect` is made by the client device.

When connecting over Bluetooth, no Internet connection is required, however the two devices must be in close physical proximity. This is useful when transferring data in areas where no Internet connection is available.

Example

```
string server_name = "supervisor21";

// connect to the supervisor's device using Bluetooth
if syncconnect(Bluetooth, server_name) then

    // send the latest cases to the supervisor's device
    syncdata(PUT, SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: SyncConnect Function, SyncConnect Function (CSWeb), SyncConnect Function (Dropbox), SyncConnect Function (FTP), SyncConnect Function (LocalDropbox), SyncConnect Function (LocalFiles), SyncMessage Function, GetBluetoothName Function, SetBluetoothName Function

SyncConnect Function (Dropbox)

This page describes the `syncconnect` function when connecting to Dropbox. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(Dropbox);
```

Description

The `syncconnect` function opens a connection to Dropbox to synchronize data and files over the Internet. The first and only argument must be the keyword ***Dropbox***. This requires an account with the online file sharing service Dropbox. It does **NOT** require that the Dropbox client software be installed on the device. The first time the connection is made, the user will see a prompt to enter the Dropbox username and password and these settings will be saved in secure storage for future synchronizations. You can use the CSPro Settings dialog to clear any cached settings.

Example

```
// connect to Dropbox
if syncconnect(Dropbox) then

    // send the latest cases to Dropbox
    syncdata(PUT,SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncConnect Function (CSWeb), SyncConnect Function (Bluetooth), SyncConnect Function (FTP), SyncConnect Function (LocalDropbox), SyncConnect Function (LocalFiles)

SyncConnect Function (LocalDropbox)

This page describes the `syncconnect` function when connecting to a Dropbox folder on the local computer. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(LocalDropbox);
```

Description

The `syncconnect` function opens a connection to Dropbox to synchronize data and files using the Dropbox folder on the local computer. The first and only argument must be the keyword ***LocalDropbox***. This requires that the Dropbox client software be installed on the computer. Rather than synchronize files over the Internet this connection relies on the Dropbox client software to do file synchronization. CSPro simply copies files and data to/from the Dropbox folder. This is mainly useful when downloading large quantities of data which would take a long time to do using the online Dropbox connection. The LocalDropbox connection is only supported on Windows.

Example

```
// connect to local Dropbox folder
if syncconnect(LocalDropbox) then

    // Download the latest cases from Dropbox
    syncdata(GET, SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncConnect Function (Dropbox), SyncConnect Function (LocalFiles), SyncConnect Function (CSWeb), SyncConnect Function (Bluetooth), SyncConnect Function (FTP)

SyncConnect Function (LocalFiles)

This page describes the `syncconnect` function when connecting to a folder on the local computer such as the home directory of an FTP server. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(LocalFiles, local_path);
```

Description

The `syncconnect` function opens a connection to synchronize data and files using the a folder on the local computer. The first argument must be the keyword ***LocalFiles*** and the second argument, the string expression ***local_path***, must be the path to a local folder on the computer. Rather than synchronize files over the Internet this connection assumes that some other service, such as an FTP server, will handle file transfers. CSPro simply copies files and data to/from the local folder. This is mainly useful when downloading large quantities of data which would take a long time to do using a network connection.

Example

```
// connect to local folder
if syncconnect(LocalFiles, "C:/FTPDirectory/") then

    // download the latest cases
    syncdata(GET, SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncConnect Function (CSWeb), SyncConnect Function (Bluetooth), SyncConnect Function (FTP)

SyncConnect Function (FTP)

This page describes the `syncconnect` function when connecting to an FTP server. For generalized information on the function, see the documentation for the `syncconnect` function.

Format

```
b = syncconnect(FTP,ftp_url⌡,username⌡,password⌡⌡);
```

Description

The syncconnect function opens a connection to an FTP server to synchronize data and files over the Internet. The first argument must be the keyword ***FTP***. The second argument, ***ftp_url***, is a string expression of the full URL of the FTP server. The third and fourth arguments, ***username*** and ***password*** are optional string arguments of the credentials used to login to the server. If the username and password are not specified, the user will be prompted to enter them the first time the synchronization is run. After the credentials have been entered the first time, they are saved in secure storage on the device and do not need to be entered again. You can use the CSPro Settings dialog to clear any cached credentials.

CSPro synchronization can work with FTP servers using encrypted transmission via TLS/SSL. There are two modes that CSPro supports for encrypted FTP: explicit and implicit. To use explicit TLS/SSL, provide a URL for the server that starts with `"ftpes://"`. To use implicit TLS/SSL, provide a URL for the server that starts with `"ftps://"`. For servers that do not support TLS/SSL, use a URL that starts with `"ftp://"`. FTP over SSH (SFTP) is not supported.

You can optionally specify a port number by adding it to the end of the URL, separated by a colon. For example, the URL `"ftp://myserver.com:27"` tells CSPro to connect to port 27. If no port is specified, the standard FTP port will be used (port 21 for FTP or port 990 for FTPS).

Example

```
string ftpServerUrl = "ftpes://myserver.org";

// connect to the FTP server
if syncconnect(FTP,ftpServerUrl) then

    // send the latest cases to the FTP server
    syncdata(PUT,SURVEY_DICT);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncConnect Function (CSWeb), SyncConnect Function (Bluetooth), SyncConnect Function (Dropbox), SyncConnect Function (LocalDropbox), SyncConnect Function (LocalFiles)

SyncServer Function

Format

```
b = syncserver(Bluetooth[, file_root_path]);
```

Description

The syncserver function runs a local server that allows peer-to-peer synchronization between two devices via Bluetooth, without needing an Internet connection. The function waits for a connection from another device (a client) made when the client calls syncconnect to start a Bluetooth synchronization. Once the two devices are connected, the client device calls syncdata and syncfile to copy data and/or files to and from the server. Finally, the client device calls syncdisconnect to end the session.

The server in a peer-to-peer synchronization is passive. It does not call syncdata or syncfile. It simply responds to requests initiated from the client. The Bluetooth server runs until either the client successfully connects and disconnects or the operator cancels the synchronization.

The syncserver function displays a dialog to the operator indicating that it is waiting for a client device to connect. During this time, no other logic on the server is executed. Once a connection is made, the dialog displays the progress of the synchronization. When the client disconnects by calling syncdisconnect, the dialog is removed and syncserver returns. If no connection is made, the operator can use the cancel button on the dialog to cause syncserver to return and continue executing any logic that follows. The server only allows connections while the syncserver function is running, so the server must call syncserver before the client device calls syncconnect.

The first argument must be the keyword ***Bluetooth***. The second argument, ***file_root_path***, is an optional string

expression of the path to use as the base directory from which to transfer files when the client calls `syncfile`. The remote path specified in the call to `syncfile` is appended to this base directory to determine the full path of files on the server. In other words, `"/"` in a `syncfile` call from the client refers to the ***file_root_path*** directory on the server. For example, if ***file_root_path*** is `"C:\MyApp\MyFiles"` and the client calls `syncfile(GET,"/Images/image.jpg")`, then the file **image.jpg** will be read from the directory **C:\MyApp\MyFiles\Images** on the server. If no ***file_root_path*** is specified, the application directory is used. The rules that dictate which files can be synced on the server differ depending on the operating system. On Android files inside the **csentry** directory can be synced by default. To sync outside the csentry directory use ***file_root_path*** to specify a path outside the csentry directory. On Windows any file can be synced.

When connecting over Bluetooth, no Internet connection is required, but the two devices must be in close physical proximity.

Return Value

The function returns a logical value of 1 (true) if the synchronization was successful and 0 (false) otherwise.

Example 1

```
// start a peer-to-peer Bluetooth server with the application directory as the file root path
syncserver(Bluetooth);
```

Example 2

```
// start a peer-to-peer Bluetooth server with the application parent directory as the file root
path
syncserver(Bluetooth, "..");
```

See also: Synchronization Overview, SyncConnect Function, SyncData Function, SyncDisconnect Function, SyncFile Function, SyncMessage Function, SetBluetoothName Function

SyncDisconnect Function

Format

```
b = syncdisconnect();
```

Description

The `syncdisconnect` function ends a synchronization session previously started by calling `syncconnect`.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
if syncconnect(Dropbox) then

    // perform synchronization actions

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncData Function, SyncFile Function, SyncParadata Function, SyncServer Function

SyncData Function

Format

```
b = syncdata(direction, dictionary_name[, sync_universe]);
```

Description

The `syncdata` function transfers cases in a CSPro data file between a device and a synchronization server. Before calling `syncdata`, you must first connect to the server by calling `syncconnect`.

The function can upload cases from the local device (client) to the server as well as download cases from the server. The ***direction*** argument determines which of these operations is performed. It must be one of the following values:

- **GET**: Download any cases that were modified on the server since the last sync and update or add them to the local data file.
- **PUT**: Upload to the server any cases that were modified in the local data file since the last sync.
- **BOTH**: Sync cases in the local data file with the server in both directions (i.e., perform both a GET and a PUT).

The ***dictionary_name*** argument must be the name of a data dictionary corresponding to the data file to synchronize. This dictionary must be an external dictionary. CSPro uses the data file that is currently associated with this dictionary, either because it was specified when the application was started or via a call to `setfile` from application logic.

For synchronization with a CSWeb server, the dictionary must first be uploaded to the server. (See the CSWeb help documentation for more information.)

For peer-to-peer synchronization, the data is written to the file associated with the dictionary of the same name on the device running the server. This means that both devices must have this dictionary added to the currently running CSPro application, either as the main dictionary or as an external dictionary.

By providing an optional string expression for the ***sync_universe*** argument, you can limit the cases that are transferred. The universe is matched against the ID items of each of the cases. Only cases whose ID items concatenated together match the universe will be transferred. For example, if the universe is `"123"` then cases with IDs `"1234"` and `"1235"` would be synced but a case with IDs `"2345"` would not.

The `syncdata` function keeps track of which cases are transferred each time the client and server are synchronized and uses this information to only transfer cases that have been modified since the last synchronization. This significantly reduces the amount of data transferred and therefore reduces bandwidth and the cost of air time. It also reduces the chance that two interviewers overwrite each other's work by both syncing to the same data file on the server. As long as the two interviewers do not modify the same case at the same time, they may both synchronize to the same server without overwriting each other's data.

The `syncdata` function is only supported on data files in CSPro DB or Encrypted CSPro DB formats. Data files in the legacy text format may not be synchronized using `syncdata` but instead can be transferred using the `syncfile` function. That function always transfers the entire file, however, increasing the amount of data transferred and allowing for one interviewer to overwrite the changes of another.

You can use the `synctime` function to get the time of the last `syncdata` call.

Return Value

The function returns a logical value of 1 (true) if the transfer was successful and 0 (false) otherwise.

Example 1

```
if syncconnect(CSWeb, "https://www.myserver.org/api") then

     // send the latest survey data to the server
    syncdata(PUT, SURVEY_DICT);

     // get the latest assignments lookup file from the server
    syncdata(GET, ASSIGNMENTS_DICT);

    syncdisconnect();

endif;
```

Example 2

```
if syncconnect(Bluetooth) then

    // sync data for only the province and district assigned to the interviewer
    string assignment_universe = maketext("%v%v", ASSIGNED_PROVINCE, ASSIGNED_DISTRICT);
    syncdata(BOTH, CENSUS_DICT, assignment_universe);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncFile Function, SyncParadata Function, SyncDisconnect Function, SyncServer Function, SyncTime Function

SyncFile Function

Format

```
b = syncfile(direction, from_path[, to_path]);
```

Description

The `syncfile` function transfers files between a device and a synchronization server. Files may be of any type: application files, images, text files, etc. However, for transferring CSPro data files it is more efficient to use the function `syncdata`. Before calling `syncfile`, you must first connect to a server by calling `syncconnect`.

The `syncfile` function uploads or downloads one or more files from the server. Unlike `syncdata`, this sends or retrieves an entire file, not individual cases. It is therefore useful for transferring files that are not data files such as images or new versions of an application.

The function can upload files from the local device (client) to the server as well as download files from the server. The ***direction*** argument determines which of these operations is performed. It must be one of the following values:

- **GET**: Download files from the server and save them on the client.
- **PUT**: Upload files from the client to the server.

The ***from_path*** argument is a string expression that specifies the path of the file to transfer. If the direction is GET then this is the path of the file on the server to download. If the direction is PUT then this is the full path to the file name on the client to upload.

The optional ***to_path*** argument is a string expression that specifies the destination path for the file. If the direction is GET then this is the full path of the destination file name on the client device. If the direction is PUT then this is the path of the destination file or folder on the server. If ***to_path*** is not specified, then the file will have the same name as the file in the ***from_path*** and will be saved to the application directory on the client (for GET) or the server root directory (for PUT).

The ***from_path*** argument may contain the wildcard characters "*" and "?" to specify a group of files to transfer. In this case, the ***to_path*** argument should be the name of a folder to copy all files that match the wildcard pattern into.

When downloading files using GET, ***syncfile*** first looks to see if the file already exists on the client device. If it does, it only downloads the file if the server has a different version. It uses the file signature (MD5) to determine if the version of the file on the server is different. This reduces the bandwidth used when downloading large files.

Return Value

The function returns a logical value of 1 (true) if the transfer was successful and 0 (false) otherwise.

Example

```
if syncconnect(CSWeb, "https://www.myserver.org/api") then

    // upload all images in images directory to the images directory on the server
    syncfile(PUT, "Images/*.jpg", "/Images/");

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncData Function, SyncParadata Function, SyncApp Function, SyncDisconnect Function, SyncServer Function

SyncMessage Function

Format

```
// Bluetooth client
s = syncmessage(message_name⌈, message_value⌋);

// Bluetooth server
function string OnSyncMessage(string message_name, string message_value)
    exit message_response;
end;
```

Description

The `syncmessage` function sends a string message from one device to another device that is acting as a Bluetooth server. The string expression ***message_name*** identifies the message, and an optional string expression ***message_value*** defines a value associated with the message. The client device sends this message to a Bluetooth server.

The application running on the Bluetooth server must have a user-defined function with the name `OnSyncMessage` that has two string parameters and returns a string. While the `syncserver` function is running, the `OnSyncMessage` function is called anytime a message is received and the return value of that function is returned to the Bluetooth client.

Return Value

The function returns the message returned by the `OnSyncMessage` function running on the Bluetooth server. If no message is returned, a blank string is returned.

Example

```
// on the client...
syncmessage("STAFF_CODE", STAFF_CODE);

if syncmessage("TRAINING_MODE") <> maketext("%d", TRAINING_MODE) then
    errmsg("You cannot sync with a device that is not in the same training mode as your
device");
    syncdisconnect();
endif;


// on the server
PROC GLOBAL

function string OnSyncMessage(string message_name, string message_value)

    // store information about the sync in the paradata log
    if message_name = "STAFF_CODE" then
        logtext("Syncing with %s at %s", message_value, timestring());

    // return the training mode
    elseif message_name = "TRAINING_MODE" then
        exit maketext("%d", TRAINING_MODE);

    endif;

end;
```

See also: Synchronization Overview, SyncConnect Function (Bluetooth), SyncServer Function

SyncParadata Function

Format

```
b = syncparadata(direction);
```

Description

The `syncparadata` function transfers paradata between a device and a synchronization server. Before calling `syncparadata`, you must first connect to the server by calling `syncconnect`.

The function can upload paradata from the local device (client) to the server as well as download paradata from the server. The ***direction*** argument determines which of these operations is performed. It must be one of the following values:

- **GET**: Download any paradata that was sent to the server since the last sync and add it to the currently open paradata log.
- **PUT**: Upload to the server any paradata that was collected since the last sync.
- **BOTH**: Sync paradata in the currently open paradata log with the server in both directions (i.e., perform both a GET and a PUT).

The `syncparadata` function keeps track of what paradata has been transferred each time the client and server are synchronized and uses this information to only transfer paradata that has been collected since the last synchronization. This significantly reduces the amount of data transferred and therefore reduces bandwidth and the cost of air time.

Paradata logs can be very large so be careful about adding paradata synchronizations to your applications. If you are interested in syncing paradata, a general suggestion is to use BOTH when syncing between devices using Bluetooth, and to use PUT to a CSWeb, Dropbox, or FTP server.

Return Value

The function returns a logical value of 1 (true) if the transfer was successful and 0 (false) otherwise.

Example

```
if syncconnect(Bluetooth) then

    // other sync actions...

    syncparadata(BOTH);

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncData Function, SyncFile Function, Paradata, Paradata Function

SyncApp Function

Format

```
b = syncapp();
```

Description

The `syncapp` function updates application files, such as a .pen and .pff, from a new version of the application package (**package.json**) on a server. This function is used to make updates, such as bug fixes and other modifications, to an application already deployed on mobile devices in the field.

The function checks for an updated version of the application on the server that the program is currently connected to from a previous call to `syncconnect`. If the version of the application on the server is newer than the currently running version, then the new package is downloaded from the server and the application on the device is updated.

This function only works on Android. It is not supported on Windows. It may be used with CSWeb, Dropbox and FTP servers as well as with peer to peer synchronization via Bluetooth.

Using `syncapp` is typically only for advanced scenarios, such as when you want to update an application over Bluetooth or when you want to initiate an update from within a menu program. In most cases it is simpler to use Update Installed Applications from the menu in the Entry Applications screen of CSEntry as it does the same update as `syncapp` without requiring any custom logic.

Before calling `syncapp`, you must first connect to a server by calling `syncconnect`.

To use the `syncapp` function, you must first upload the application package to a server using the Deploy Application tool. The `syncapp` function looks for a package on the server with the same package name as the currently running application. If no package matching the current package name has been deployed to the server, the function will display an error. Note that the application package is not necessarily the same as the name of the .pen file. The package name is the name that was entered in the Deploy Application tool when the package was deployed to the server. It is important therefore to always use the same package name whenever deploying updates to an application.

The `syncapp` function only downloads the application if the version of the application on the server is more recent than the version on the mobile device. This is accomplished by storing the date and time the package was uploaded to the server by the Deploy Application tool. This date and time is stored in the application package and is used to determine if the package on the server is newer than the package on the mobile device.

Furthermore, if you are using a CSWeb server or Bluetooth, in order to keep data usage to a minimum, only files that are different in the version of the package on the server from those in the version on the mobile device are downloaded. This way if you have large files in your application package that do not change as part of the update, they will not slow down the update. This optimization is not done for Dropbox or FTP servers.

Files in the downloaded package overwrite the corresponding files on the mobile device. To avoid generating multiple copies of the same file, use the same directory layout, file and folder names in the updated package as were used in the original package.

Files that are marked **Only on first install** in the deployment package that already exist on the mobile device will be ignored by `syncapp` and are not updated even if a newer version is included in the updated package.

Be careful when including data files in an application package that is used with `syncapp` since any data files that are in use by the current application will be locked which will result in errors when `syncapp` attempts to overwrite them. Either `close` any files that may be updated prior to calling `syncapp` or mark the data files as **Only on first install** in the Deploy Applications tool and use `syncdata` to update them instead.

Note that if as part of the application update the currently running .pen file is updated to a newer version, `syncapp` will restart the application in order to run the updated version. This means that any logic that follows the call to `syncapp` will not be executed. To avoid problems, make sure that the call to `syncapp` is the last call in the procedure. For example, to combine the application update with calls to `syncdata` to update data files, place the calls to `syncdata` before the call to `syncapp`.

Return Value

The function returns a logical value of 1 (true) if the update was successful and 0 (false) otherwise.

Example

```
if syncconnect(CSWeb,"https://www.myserver.org/api") then

    // get the latest versions of the application files from the server
    syncapp();

    syncdisconnect();

endif;
```

See also: Synchronization Overview, SyncConnect Function, SyncData Function, SyncDisconnect Function, SyncServer Function

SyncTime Function

Format

```
d = synctime(dictionary_name[, device_identifier, case_identifier]);
```

Description

The `synctime` function returns the UNIX time of the last synchronization of a data file or a specific case, or `notappl` if no such synchronization has occurred. The ***dictionary_name*** argument indicates which data file's details are returned. This dictionary can be either the dictionary for the main data file or an external dictionary. CSPro uses the data file that is currently associated with this dictionary, either because it was specified when the application was started or via a call to `setfile` from application logic. Unlike the other synchronization functions, this function can be called without connecting to a server as it returns metadata stored offline in the CSPro DB file.

Providing an optional string expression ***device_identifier*** allows you to query the last synchronization time to a specific device. You can supply the identifier by using a name. A name like `"Bluetooth"` will return information on the last synchronization with *any* Bluetooth device. A name like `"https://www.myserver.org/api"` will return information on the last synchronization with the specified server.

Alternatively, you can specify ***device_identifier*** using a device ID. This is most relevant if trying to differentiate between Bluetooth syncs. The device ID corresponds to the value that would be returned by calling `getdeviceid` on the Bluetooth device of interest. For example, providing a device ID like `"6ce7d9416fa72a85"` will return information on the last

synchronization with the device that has that ID. If no ***device_identifier*** is supplied, or if it is blank, then the function will return synchronization information for any device.

The optional string expression ***case_identifier*** allows you to specify a case UUID to get synchronization information on a case-by-case basis. A case's UUID is a value that uniquely identifies a case, even when multiple cases share the same key (ID items), and can be retrieved using the `uuid` function. The function only returns the synchronization time of the current iteration of the case, so if a case had been synced but has since been modified, the function will return `notappl`.

Return Value

The function returns the time of the last synchronization, or `notappl` if there was not a synchronization. In summary:

Device Identifier	Case Identifier	Last Synchronization Time...
Undefined or Blank	Undefined or Blank	...of any case to any device
Not Blank	Undefined or Blank	...of any case to the specified device
Blank	Not Blank	...of the specified case to any device
Not Blank	Not Blank	...of the specified case to the specified device

Example 1

```
numeric unsynced_cases = countcases(SURVEY_DICT where synctime(SURVEY_DICT, "", uuid(SURVEY_DICT
)) = notappl);

if unsynced_cases > 0 then
    errmsg("There are %d cases on your device that have not been synced to any
device.", unsynced_cases);
endif;
```

Example 2

```
string ServerUrl = "https://www.myserver.org/api";
numeric SecondsInWeek = 60 * 60 * 24 * 7;

when synctime(SURVEY_DICT, ServerUrl);
     notappl                          -> errmsg("You have never synced. Please sync as soon as
possible.");
     < ( timestamp() - SecondsInWeek ) -> errmsg("You have not synced in a week. Please sync as
soon as possible.");
endwhen;
```

See also: Synchronization Overview, SyncData Function, Timestamp Function, TimeString Function

GetBluetoothName Function

Format

```
s = getbluetoothname();
```

Description

The `getbluetoothname` function returns the name of the device as broadcast to other Bluetooth devices. When other Bluetooth-enabled devices scan for nearby devices to connect to, this is the name that they will see.

When using Bluetooth for synchronization between two devices, the Bluetooth name of the device is what is shown in the screen asking the user which device to connect to. The Bluetooth name of one device may be passed as an argument to the `syncconnect` function on the other device in order to connect directly to that device, bypassing the screen asking the user which device to connect to.

Because it is possible to change the Bluetooth device name using logic (`setbluetoothname`) or in the settings on Android and in the Device Manager on Windows, the Bluetooth name can be used as a way to name devices for management and audit purposes. The device can be assigned a unique identifer or serial number which is used as the Bluetooth device name. Using `getbluetoothname`, this identifier can be written to an item in the data file for audit purposes, supplied as a username for synchronization, or used as a key in lookup file as part of a survey management system.

Return Value

The function returns a string containing the name of the device.

Example

```
PROC DEVICE_NAME

preproc

    DEVICE_NAME = getbluetoothname();
    noinput;
```

See also: SetBluetoothName Function, GetDeviceID Function, GetOperatorId Function, GetOS Function, GetUserName Function, SyncConnect Function (Bluetooth)

SetBluetoothName Function

Format

```
b = setbluetoothname(bluetooth_name);
```

Description

The `setbluetoothname` function modifies the name of the device as broadcast to other Bluetooth devices, setting it to the string expression ***bluetooth_name***. When other Bluetooth-enabled devices scan for nearby devices to connect to, this is the name that they will see.

When using Bluetooth for synchronization between two devices, the Bluetooth name of the device is what is shown in the screen asking the user which device to connect to. The Bluetooth name of one device may be passed as an argument to the `syncconnect` function on the other device in order to connect directly to that device, bypassing the screen asking the user which device to connect to.

On Windows, you may need to run your application with administrative privileges to successfully change the Bluetooth name.

Return Value

The function returns a logical value of 1 (true) if the Bluetooth name was set and 0 (false) on error.

Example

```
// set the Bluetooth name to the staff ID prior to syncing
setbluetoothname(maketext("NSO Device %v", STAFF_ID));

syncserver(Bluetooth);
```

See also: GetBluetoothName Function, SyncConnect Function (Bluetooth)

Export Attributes

Export Attributes

This page explains the numerous types of connection string attributes available to you when working with files in the CSPro run dialogs. The following matrix presents all string attributes, and what file types they can be used by:

Connection String Attribute	.csdb(e)	.dat	.csv	semi-colon	.tsv	Excel	R	SAS
record	Y	Y	Y	Y	Y	Y		Y
mapped-special-values		Y	Y	Y	Y	Y	Y	Y
encoding			Y	Y	Y			Y
header			Y	Y	Y	Y		
values			Y	Y	Y	Y		
decimal-mark			Y	Y	Y			
vector							Y	
factor							Y	
dictionary	Y	Y						
syntax-file								Y

The attributes listed above are presented below as section headings. When combined with the file name in a run dialog, using connection string attributes (properties) allows you to more tightly control how files are created. As outlined on the connection string page, a connection string is made up of several components:

```
file_name | property_name1=property_value1 & property_name2=property_value2 & property_name3=property_value3
```

After the pipe symbol ("|") has been given, connection strings can be given in any order, they just need to be combined with the ampersand symbol (&).

The following is a list of available connection string attributes:

- Export Attribute: record
- Export Attribute: mapped-special-values
- Export Attribute: encoding
- Export Attribute: header
- Export Attribute: encoding
- Export Attribute: value
- Export Attribute: decimal-mark
- vector goes here, see stub below
- factor goes here, see stub below
- Export Attribute: dictionary
- Export Attribute: syntax-file
- Export Attribute: type

vector

The **vector** attribute is available for R data sources. There are three options: codes, factors, and codes-and-factors.

...describe this attribute

factor

The **factor** attribute is available for R data sources. There are two options: discrete and all.

...describe this attribute

Export Attribute: decimal-mark

When creating a new **.csv**, Semicolon, or **.tsv** data source, you might want to change the way decimal values are written out. In the U.S. and Canada, a decimal point is used to indicate fractional values (41.38), whereas in many European countries the decimal comma is used (41,38). By default, CSPro uses the decimal point; but by including this attribute and option in your connection string for a file:

```
decimal-mark=comma
```

this will be flipped. The two possible values for the decimal-mark attribute are **point** and **comma**.

See also: Export Attributes

Export Attribute: dictionary

When the **type=CSProExport** attribute is used to generate CSPro data files (**.csdb** or **.dat**), a dictionary (**.dcf**) will be automatically created. The **dictionary** attribute allows you to control the file name of the **.dcf**. For example, if the following connection string was used for an output file:

```
OutputData=.\PopstanCensus.csdb|type=CSProExport
```

A dictionary would be generated with the file name **PopstanCensus.csdb.dcf**. If you would rather give it a different name, include the dictionary attribute. For example,

```
OutputData=.\PopstanCensus.csdb|type=CSProExport&dictionary=PopstanCensus.dcf
```

would create a **.dcf** file named PopstanCensus.dcf.

See also: Export Attributes

Export Attribute: encoding

If you wish to create data sources using a particular encoding scheme, you can do so with the **encoding** attribute for **.csv**, semicolon, **.tsv**, and SAS files. The format is:

```
file_name | encoding=encoding_value
```

where ***encoding_value*** can be **ansi**, **utf-8**, **utf-8bom**, or **unicode**. If this attribute is not used, the default encoding is UTF-8BOM.

See also: Export Attributes

Export Attribute: header

The **header** option can be used with **.csv**, semicolon, **.tsv**, and Excel files. As the name implies, it controls the header (first) line written to these files. The available options are:

- **none**—no header will be written
- **names**—the dictionary item's unique name will be used as the column header (the default behavior when this attribute is not used)
- **labels**—the dictionary item's label will be used as the column header

See also: Export Attributes

Export Attribute: mapped-special-values

If any special values have been used in your dictionary (default, missing, or refused), using this attribute will allow you to control how those values will be written to your new data source. It can be used on any file type except CSPro files (i.e., **.csdb** and **.dat**). The format is:

```
file_name|mapped-special-values=mapped_value
```

where ***mapped_value*** can be one of the following:

- **code**—the numeric value associated with the special value will be written to the data source (the default behavior when this attribute is not used)
- **notappl**—the text 'value' associated with the special value will be written (i.e., "default", "missing", "refused")
- **software-missing**—available when working with SAS data sources. SAS has two special values, MISSING and REFUSED. If the **software-missing** option is used, MISSING values will be mapped to .A and REFUSED values will be mapped to .B. If this option is not used and a MISSING value is mapped to, for example, -99, then your SAS file would have -99 instead of .A.

If the **mapped-special-values** attribute is not used, the default behavior will be to write out the numeric values of each dictionary item.

See also: Export Attributes

Export Attribute: record

When working with a datafile that contains 2+ records, this allows you to select a specific record for output. The format is as follows:

```
file_name | record=record_name
```

An example of this attribute being used for the output data file is:

```
.\pop_record.xlsx|record=POP_REC
```

By using this option for each record type, you can create multiple output files. For example, if your dictionary contained a housing and population record, you could each as separate files—and as different output file types—as shown below:

```
".\pop_record.xlsx|record=POP_REC"
".\housing_record.csv|record=HOUSE_REC"
```

Note the first connection string will generate an Excel file, and the second will generate a **.csv** file. While appearing as a single entry in the run dialog, when you look at the **.pff** file, you will see they have been written on separate lines as follows:

```
OutputData=.\pop_record.xlsx|record=POP_REC
OutputData=.\housing_record.csv|record=HOUSE_REC
```

The **record** attribute can not be used more than once when creating a new data source. For example, if your datafile had 3 records (housing, population, and vaccination), and you wanted to export the population and vaccination records into a

new file, you can not do this with this feature. Only a single record type can be named for each new data source. If you wanted to export multiple, but not all, of the datafile's records, you would have to use CSPro's utility CSExport, found in the "Tools" dropdown of most CSPro applications.

If the record attribute is used to generate a CSPro file (either .csdb(e) or .dat), you must include the **type** attribute set to "CSProExport". This is necessary as it tells the exporter that a CSPro data file is being created, which prompts it to create the dictionary first. The exporter then uses that **.dcf** to construct the resultant datafile. Therefore, if the housing record above was being written to a **.csdb** file instead of the **.csv** file, the connection string would change to the following:

```
.\housing_record.csdb|record=HOUSE_REC&type=CSProExport
```

If the **record** attribute is not used, the default behavior will be to write out all records found in the datafile.

See also: Export Attributes

Export Attribute: syntax-file

Similar to the **dictionary** attribute, the **syntax-file** attribute allows you to control a filename—in this case, the command (syntax) file (**.sas**) that accompanies a SAS data source creation (**.xpt**). By default, the command file will take the same name as the SAS file. For example, when SAS is chosen as the output file data source, this file name:

```
OutputData=.\PopstanCensus.xpt
```

would create a command file named PopstanCensus.xpt.sas. However, by using the **syntax-file** attribute, you can change that behavior. For example,

```
OutputData=.\PopstanCensus.xpt|syntax-file=PopstanCensus.sas
```

would create a **.sas** command file named PopstanCensus.sas.

See also: Export Attributes

Export Attribute: type

The **type** attribute can be used with all data sources. However, it has not been included in the chart above. Why? Because it is not normally needed; the **type** attribute is only necessary when CSPro can't determine what file format should be used when creating a data source. How does CSPro determine the file format? By the file extension. If a file extension is used that CSPro has no association for (for example, ".foo"), CSPro defaults to a text file. But by using the **type** attribute, you can control that behavior, should you think there might be any confusion.

There are only two times when the **type** attribute is required.

(1) When you don't want to provide a file name during the run. For example, maybe your application needs a lookup file, but your file isn't quite ready to go. You can add the lookup file to your application, but during your test runs, you can designate "none". In the **.pff** for your batch, you might see:

```
[ExternalFiles]
GeoLookup=|type=None
```

(2) When a CSPro data source is being created via the **record** attribute. In that situation, type=CSProExport must be included. For example, if creating a data source by extracting the vaccination record from the main data file, the following connection syntax would be needed:

```
OutputData=.\VaccRec.csdb|record=VACC_REC&type=CSProExport
```

The full list of acceptable **type** arguments are:

```
type=None
type=Text
type=CSProDB
type=EncryptedCSProDB
type=CSV
type=Semicolon
type=Tab
type=Excel
type=CSProExport
type=R
type=SAS
type=SPSS
type=Stata
```

See also: Export Attributes

Export Attribute: value

The **values** attribute can be used with **.csv**, semicolon, **.tsv**, and Excel files. It controls how the data file's values are written to the file. Available options are:

- **codes**—numeric codes will be written (the default behavior when this attribute is not used)
- **labels**—the value set's label will be used (for example, if Sex=1 or 2, the labels might read Male and Female)
- **codes-and-labels**—two columns will be written for each data point; the codes, followed by the value set label for that item

For example, suppose you want to generate an Excel file that uses the labels in the header and generates codes and labels (so, 2 columns) for each data point. The syntax for the Output Data file connection string would be as follows:

```
OutputData=.\pop_rec.xlsx|record=POP_REC&header=labels&values=codes-and-labels
```

And the resultant data source would appear as follows (a partial screenshot being shown):

Relationship	Relationship	Sex	Sex	Age	Age	Marital Status (age 15+)	Marital Status (age 15+)
1	Head	1	Male	24	24	1	Married/Living Together
2	Spouse	2	Female	22	22	1	Married/Living Together
3	Child	1	Male	4	4		
3	Child	2	Female	2	2		
8	Sibling	1	Male	18	18	4	Never Married/Lived Together

You can see that generating two columns for the relationship, sex, and marital status variables clarified the meaning of those codes; however for other fields, such as age, no additional information is imparted. You may also notice that for some fields (Marital Status), no value is given. Changing the connection string to include the mapped-special-values attribute:

```
OutputData=.\pop_rec.xlsx|record=POP_REC&header=labels&values=codes-and-labels&mapped-special-
values=notappl
```

yielded no changes. What else could be happening? As you can guess from the ages and see from the column heading, the marital status field was not asked (i.e., it was skipped) for persons under 15 years of age. Oftentimes marital status is asked for all persons regardless of age; in that instance, we would have expected to see "Never Married" or "Single".

See also: Export Attributes

Templated Reporting System

Templated Reports

(Prior to CSPro 7.7, a different version of templated reports existed. That version should no longer be used as it will be removed in a future release.)

Overview

CSPro can create HTML reports using case data as well as results from queries into paradata and other data sets. By specifying tags in the templated reports, CSPro logic can be run and CSPro dictionary variables can be accessed to render a dynamic report.

Functionality

Each templated report is a CSPro logic object and the following functions can be called via dot notation:

Function	Description
`save`	Saves the report to a file.
`view`	Displays the report in an embedded web browser.
`write`	Writes text to the report.

In addition to these object functions, templated reports can be used as arguments to the `filename` and `view` functions.

Assignments

When a templated report is used as an argument to a user-defined function, it is passed by reference.

Add, Remove, and Rename

A templated report can be added to the project by selecting Add Files under the **Files** menu. If creating a new HTML file, an additional dialog will be displayed with template options. The templates have an HTML scaffolding which includes popular CSS and JavaScript libraries. A templated report can be removed from the project by selecting Drop Files under the **Files** menu.

The name of the report will be automatically created based on the filename, but can be modified using the Report Properties dialog.

CSPro Logic

CSPro logic can be embedded in the templated report by starting the logic with **<?** and ending the logic with **?>**. The tag can be used to embed inline logic or multiple lines of logic.

Output

There are three ways of writing numeric and string expressions to templated reports:

1. **~~**...**~~** writes the results of the numeric or string expression specified between the tildes.
2. **~~~**...**~~~** is similar to the above version except that the text will not be automatically escaped for HTML.
3. `$.write` writes directly to the report.

Preview

A templated report can be previewed in both data entry and batch by pressing **Ctrl + F5**. This will not run the CSPro logic, so the preview will not necessarily look complete. Use it to preview the HTML and CSS.

Example 1

```
<p>Report generated at ~~timestring()~~</p>
```

Example 2

```
<?
    numeric partial_count = 0;
    numeric complete_count = 0;

    forcase HOUSEHOLD_DICT do
        if ispartial(HOUSEHOLD_DICT) then
            inc(partial_count);
        else
            inc(complete_count);
        endif;
    endfor;
?>
    <p><b>~~getsymbol(HOUSEHOLD_DICT)~~</b> has ~~partial_count~~ partial case(s) and
~~complete_count~~ complete case(s).</p>
```

See also: HTML in CSPro, JavaScript Interface

Report.Save Function

Format

```
b = report_name.save(report_filename);
```

Description

The `report.save` function runs the logic in the templated report and saves the output as a file specified by the string expression ***report_filename***. The function is meant to be called from outside the templated report's logic.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Example

```
HOUSEHOLD_REPORT.save(maketext("household_report_%d.html", timestamp()));
```

See also: Templated Reports, Report.View Function

Report.View Function

Format

```
b = report_name.view([optional_named_arguments]);
```

Description

The `report.view` function runs the logic in the templated report and displays the output in an embedded web browser. The function is meant to be called from outside the templated report's logic.

HTML files or websites are displayed in an embedded window using a Chromium-based web browser (Edge WebView2 on Windows; WebView on Android). In this web browser, it is possible to use the JavaScript interface to run user-defined functions or to execute other CSPro operations.

Optional named arguments can be used to control how the embedded web browser displays on Windows:

- `title :=` string expression giving the title of the window.
- `width :=` numeric expression specifying the width of the window in display units.
- `height :=` numeric expression specifying the height of the window in display units.
- `showCloseButton :=` conditional expression that, if false, hides the window's *Close* button.

If either the height or width is specified, then both values have to be provided. Hiding the *Close* button also removes the margin around the web browser, so the browser will fill the entire embedded window.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Examples

```
HOUSEHOLD_REPORT.view();

HOUSEHOLD_REPORT.view(title := "Household Report",
                      width := 600, height := 400,
                      showCloseButton := false);
```

See also: Templated Reports, Report.Save Function

Report.Write Function

Format 1

```
b = report_name.write(string_exp⌈, argument1, ..., argumentN⌋)
```

Format 2

```
b = report_name.write(msg_num⌈, argument1, ..., argumentN⌋)
```

Description

The `report.write` function writes text directly to a templated report. The function is meant to be called from within the templated report's logic.

Each ***argument*** is sequentially inserted into the ***string_exp*** or ***msg_num***. Arguments can be numeric or string expressions, but the type of the argument must match the type of the receiving field in the message text. See Message Formatting Options for a complete list of formatting options.

Return Value

The function returns a logical value of 1 (true) if successful and 0 (false) otherwise.

Examples

```
<?
    HOUSEHOLD_REPORT.write("The report was generated at: ", timestamp());

    // Alternatively, use $ to refer to HOUSEHOLD_REPORT
    $.write("The report was generated at: ", timestamp());
?>
```

See also: Templated Reports

Templated Reports (Deprecated)

***Feature Upgrade**: Starting with CSPro 7.7, this version of Templated Reports has been superseded by a new version, Templated Reports. This version will be removed in a future version of CSPro.*

Overview

CSPro can create HTML reports using case data as well as results from queries into paradata and other data sets. By specifying tags in your report template, you indicate to a templating engine what areas of your report are dynamic, and then via the `setreportdata` function, CSPro creates an object that the templating engine uses to render a dynamic report. The reporting system also supports the ability to automatically run queries on data sets based on queries specified in the report template.

Basic Functionality

This example will demonstrate how to create a simple report using Mustache, the default templating engine. The basic report template is a standard HTML file to which you add two components. First, you add a div block with the id **cspro_report**. This is the placeholder block where the template engine will place the results of its templating.

```
<div id="cspro_report"></div>
```

Second, you specify the text that you want to pass to the templating engine. Using the script element, you specify the type, **application/vnd.cspro.report-template**, as well as the id, **cspro_report_template**. In between the script tags, you specify the HTML. Using Mustache, replacement parameters are defined in between double brackets {{ ... }}, and # and / are used to start and end processing on elements of an array.

```
<script type="application/vnd.cspro.report-template" id="cspro_report_template">
    <h2>{{title}}</h2>
    <table>
        <tr><th>Name</th><th>Sex</th><th>Age</th></tr>
        {{#PERSON_REC}}
            <tr><td>{{NAME}}</td><td>{{SEX}}</td><td>{{AGE}}</td></tr>
        {{/PERSON_REC}}
    </table>
</script>
```

When CSPro passes this template to Mustache, the templating engine will look to replace {{title}}, and then for each occurrence (array element) of PERSON_REC, it will replace {{NAME}}, {{SEX}}, and {{AGE}}. Once it has done this, it will replace the **cspro_report** block with the dynamic contents.

Within CSPro, we must pass these parameters to the reporting system. Assuming that our template is saved to a file named **simple_report.html**, then our code might be as follows:

```
setreportdata("title","Demographic Report");
setreportdata(PERSON_REC);
report("simple_report.html");
```

The calls to `setreportdata` set the data for the title and copy the information on the person record to an array that can be accessed in the template. The call to `report` specifies the name of the template. CSPro will read the template, pass

in the title and person record data, and display the report in a web browser.

Details for Advanced Users

There are a few components that make up the reporting system, each described in detail below.

Data

The first component is the data itself. Via calls to `setreportdata`, CSPro creates a JSON object and embeds it as a script with the id **cspro_report_data**. If your template has such an id defined, it will be overwritten; otherwise, it will be inserted in the output. If you want to define your own templating engine, it must be able to work with this JSON object.

Report Template

As described above, the report template contains the HTML text that will be passed to the templating engine. That, along with the JSON data, is what turns the template into a dynamic report. The template has a few parameters:

```
<script type="application/vnd.cspro.report-template"
        id="..."
        data-template-engines="..."
        data-queries="...">
    <!-- HTML report -->
</script>
```

The type must always be **application/vnd.cspro.report-template**.

The id uniquely identifies the script block. There are no restrictions on its name, but if you want to use the basic report functionality, then the id must be **cspro_report_template** and there must be a div block with the id **cspro_report**.

The optional data-template-engines specifies the names of the templating engines that are required to create the report. Separate multiple names with whitespace. If not specified, and if the id is **cspro_report_template**, Mustache will be used, as that engine comes installed with CSPro. If you specify a templating engine that CSPro cannot locate, the report generation will result in an error.

The optional data-queries specifies the names of any queries whose execution is required to create the report. Separate multiple names with whitespace. CSPro will check if data with the query name has been set via `setreportdata`. If not, it will search for a query with the proper name, and if found, execute it. If no suitable query can be found, the report generation will result in an error.

Query

Any queries defined in your template will be executed prior to CSPro generating the report. Queries will be passed through the `setreportdata`/`sqlquery` functions with the result set stored under the attribute defined by data-query-name. The query has a few parameters:

```
<script type="application/vnd.cspro.query"
        data-query-name="..."
        data-description="..."
        data-source="...">
    <!-- SQL query -->
</script>
```

The type must always be **application/vnd.cspro.query**.

The data-query-name is the name that CSPro uses to refer to the query. You do not need to specify an id, as CSPro uses this name instead. This is the name that corresponds to entries in a report template's data-queries.

The data-description is optional text that may be used to give a more complete description of the query.

The data-source identifies the source of the query. The reporting system must know on which database to execute the query. The options are either **paradata**, to execute a query on the currently open paradata log, or a dictionary name, in which case the query is executed on the currently open CSPro data source associated with that dictionary.

Template Engine

Most users will not have a reason to add a new templating engine, as Mustache is sufficient for most reports, but some advanced users may want to define their own engine. The engine has a few parameters:

```
<script type="application/vnd.cspro.template-engine"
        data-template-engine-name="..."
        data-script-type="..."
        data-include-src="..."
        data-include-type="...">
    <!-- script code -->
</script>
```

The type must always be **application/vnd.cspro.template-engine**.

The data-template-engine-name is the name that CSPro uses to refer to the template engine. You do not need to specify an id, as CSPro uses this name instead. This is the name that corresponds to entries in a report template's data-template-engines.

The optional data-script-type specifies the type of code located in the script block. If not specified, it defaults to **application/javascript**.

The optional data-include-src identifies the file, generally a JavaScript file, that contains the source for the templating engine.

The optional data-include-type specifies the type of code found in the file identified by data-include-src. If not specified, it defaults to **application/javascript**.

CSPro will include in the HTML report the script block of any templating engine necessary to generate the report, as well as links to any (JavaScript) files required to run the template.

Evaluating Queries and Template Engines

CSPro first reads in any report templates and determines what queries and template engines are required to create the report. It checks any data set via `setreportdata`, and if that data has an attribute with the same name as a query, then it is assumed that that data overrides the query and the query required by the template is assumed to be handled and is ignored. CSPro then looks in the report template for the definition of any queries and template engines. If they are found, then they are processed.

However, if CSPro cannot find the definition of a query or template engine in the report template, it looks in two places for these definitions, which can be included in report script definition files, which have the extension **.csrs**. These files consist of script blocks defining queries and template engines. First, CSPro looks in the directory where the report is being created and loads any .csrs files in that directory. If CSPro cannot find the definition of a query or template engine in those files, it then loads the .csrs files that come with default CSPro installation. If it cannot find them in either of these locations, the report generation will result in an error.

To see an example of how this works, create a basic report, using the Mustache engine, and the output of the report will use the engine as it is defined in "C:\Program Files (x86)\CSPro 7.7\Reports\CSPro Reports.csrs." You could override the default Mustache behavior by adding a template engine with the name "mustache" to your report template or to a .csrs file in the directory where your report will be created.

See also: Report Function (Deprecated), SetReportData Function (Deprecated), SqlQuery Function

Mustache Templated Reports (Deprecated)

***Feature Upgrade**: Starting with CSPro 7.7, the version of Templated Reports that uses Mustache Templated Reports has been superseded by a new version, Templated Reports. This version will be removed in a future version of CSPro.*

Mustache is an open-source templating engine that comes installed with CSPro. You can use Mustache to create simple reports templated reports. More information on Mustache is available on the project's website, and there are many online examples of using Mustache.

Below are some examples of how to use Mustache with data from CSPro:

Example - Text

```
setreportdata("operator_name");
```

```
<h2>Report for {{operator_name}}</h2>
```

Example - Records

```
// each record occurrence is output as an array, with the items
// for each occurrence of the record output within that array
setreportdata(PERSON_REC);
```

```
{{#PERSON_REC}}
    <p><b>{{NAME}}</b> is aged {{AGE}}.</p>
{{/PERSON_REC}}
```

Example - SqlQuery

```
string DeviceQuery =
    "SELECT device_model, COUNT(*) AS device_count "
    "FROM device_info "
    "GROUP BY device_model;";

setreportdata("device_info",sqlquery(paradata,DeviceQuery));
```

```
{{#device_info}}
    <p>{{device_count}} interviewers used {{device_model}}.</p>
{{/device_info}}
```

See also: Templated Reports (Deprecated)

HTML and JavaScript Integration

HTML in CSPro

Overview

While CSPro is not a web-based application, HTML is used in CSPro in a variety of ways:

1. When displaying question text.
2. When displaying logic-based reports written in HTML.
3. When displaying content using the `view` or `htmldialog` functions.
4. When displaying some dialogs for logic functions, as well as other aspects of the CSPro user interface (UI).

HTML files or websites are displayed in an embedded window using a Chromium-based web browser (Edge WebView2 on Windows; WebView on Android). In this web browser, it is possible to use the JavaScript interface to interact with various CSPro features.

Local Web Server and HTML Directory References

When a CSPro application runs, a local web server is launched that serves content stored on the local file system. On Windows, this server uses *localhost* as the host and chooses a dynamic port, checking for one that is not in use.

Relative paths that are not based off the root are evaluated from the location of the displayed HTML content:

- **Question Text**: Relative to the data entry application (.ent) or .pen file.
- **Reports**: Relative to the location of the report (or, when running a .pen file, where it would have existed relative to the data entry application).
- **HTML Files** (shown using `view` or `htmldialog`): Relative to the location of the file.

For example, these references in question text would refer to files in the application directory and in a relative directory located one level above the application directory:

```
<img src="image-in-application-directory.jpg" />
<img src="../shared/image-in-relative-directory.jpg" />
```

HTML, JavaScript, CSS, and other files that CSPro uses for its operations are located in a *html* directory. On Windows, these files are typically found here:

```
C:\Program Files (x86)\CSPro 7.7\html\
```

The `pathname` and other path-related functions can return the name of this directory using the `Html` argument.

If you want to access any of the files in that directory in HTML that you write, you should not reference the file using an absolute path (using a file URL) because the web browser that displays the HTML may not be able to access local files. Instead, you should use the `path.getRelativePath` function to construct a relative path from the location of your HTML content. For example, if trying to use the PFF icon from a report:

```
<!-- will not work when displayed in CSPro: -->
<img src="file:///C:\Program Files (x86)\CSPro 7.7\html\images\pff-icon.png" />

<!-- correct format: -->
<img src="~~path.getRelativePath(path.getDirectoryName(filename($)),
                                 "C:\Program Files (x86)\CSPro 7.7\html\images\pff-
icon.png")~~" />
```

Fortunately, the local web server treats the *html* directory as the root of the server. The above example can instead be coded as:

```
<img src="/images/pff-icon.png" />
```

JavaScript Libraries Included With CSPro

Some popular JavaScript libaries are included in the *html* directory:

- Bootstrap: "Quickly design and customize responsive mobile-first sites with Bootstrap, the world's most popular front-end open source toolkit, featuring Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful JavaScript plugins."
- jQuery: "jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers".
- Leaflet (Windows only): "Leaflet is the leading open-source JavaScript library for mobile-friendly interactive maps."
- mustache.js: "Mustache is a logic-less template syntax. It works by expanding tags in a template using values provided in a hash or object."

To use these libraries, use the following code:

```
<!-- Bootstrap -->
<link rel="stylesheet" href="/external/bootstrap/bootstrap.min.css" />
<link rel="stylesheet" href="/external/bootstrap/bootstrap-icons.css" />
<script src="/external/bootstrap/bootstrap.bundle.min.js"></script>

<!-- jQuery -->
<script src="/external/jquery/jquery.min.js"></script>

<!-- Leaflet (Windows only) -->
<link rel="stylesheet" href="/external/leaflet/leaflet.css" />
<script src="/external/leaflet/leaflet.js"></script>

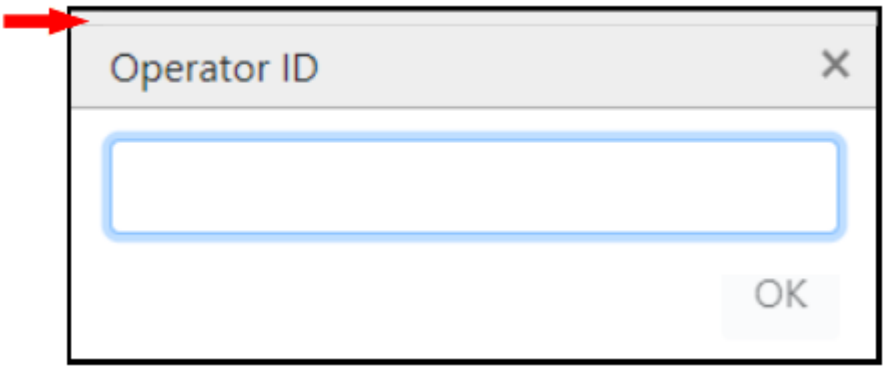
<!-- Mustache -->
<script src="/external/mustache/mustache.min.js"></script>
```

See also: JavaScript Interface

HTML Dialog Boxes

HTML is used in CSPro in a variety of ways, including for displaying some dialogs for logic functions, as well as other aspects of the CSPro user interface (UI).

On Windows, a HTML dialog looks a bit different from the standard Windows dialogs. Normally a user can use the cursor to grab the title bar of a dialog to move it around the screen. Now, the user must grab the small sliver of area at the top of the dialog, demarcated by a thin horizontal band just below the main dialog frame:



In addition, the dialog box has a close button (X) in the upper right corner that can be used to dismiss the dialog when appropriate.

Overriding CSPro Dialogs

The HTML files for the dialogs that CSPro uses are typically found here:

```
C:\Program Files (x86)\CSPro 7.7\html\dialogs\
```

To override the look or behavior of the HTML dialogs, it is possible to modify the files in that directory, but any changes will only take effect on the device where the changes are made, and will not affect applications run on Android. Instead, if you want to customize the dialogs, create a copy of the dialogs and reference it by specifying the HtmlDialogs directory override in the PFF file.

See also: HTML in CSPro, HtmlDialog Function

JavaScript Interface

Overview

CSPro displays HTML in an embedded browser when showing question text, when displaying logic-based reports written in HTML, and when displaying HTML content using the `view` or `htmldialog` functions. The HTML content is shown in a Chromium-based web browser (Edge WebView2 on Windows, and WebView on Android). There is a limited API that facilitates communication between JavaScript and CSPro.

By using JavaScript in question text, you can include links in your question text that perform actions such as opening a manual or skipping to a different question. By using JavaScript in reports or in HTML shown using the `view` or `htmldialog` functions, you can add dynamic interactive features to your application.

When using the JavaScript interface, an error will display if CSPro cannot properly interpret the call. When calling a JavaScript method, if a movement statement (e.g., `reenter`) or application halt (e.g., `stop`) occurs, the embedded browser will close automatically.

Interface

The interface is accessed using several methods belonging to the CSPro object. Some JavaScript interface methods are called synchronously (sequentially) while others are called asynchronously (concurrently). When a synchronous method is called, the JavaScript instruction following the call will not be executed until the method returns the result from CSPro. When the asynchronous method is called, the JavaScript instruction following this call will be executed immediately without waiting for the return value from CSPro.

These methods are always available:

```
CSPro.getMaxDisplayWidth()
```

Returns the maximum display width. This method executes synchronously.

`CSPro.getMaxDisplayHeight()`

Returns the maximum display height. This method executes synchronously.

`CSPro.runLogic(logic)`
`CSPro.runLogicAsync(logic, callback = undefined)`

Runs a user-defined function and returns the result of the function as a string. The parameter `logic` contains the CSPro logic to call the function. The logic will be compiled at runtime, which means the full compiler is not available and thus only numeric constants and string literals can be passed as arguments to the CSPro function. (This limitation is due to the absence of the full CSPro compiler when running CSEntry on Android.) In the asynchronous version of this method, the optional parameter `callback` is JavaScript code to execute following the running of the CSPro logic. Due to threading issues with the embedded browser, if the user-defined function you call may display UI elements as part of its operations, you must use the asynchronous version.

`CSPro.invoke(functionName, arguments = undefined)`
`CSPro.invokeAsync(functionName, arguments = undefined, callback = undefined)`

Runs the `invoke` function to execute a user-defined function using runtime binding. The parameter `functionName` is a string containing the user-defined function name. The optional parameter `arguments` is a JSON string containing the arguments to the function. Only numeric constants, string literals, and arrays of numerics and strings can be provided as arguments. (Arrays will be transformed into a CSPro `list`.) In the asynchronous version of this method, the optional parameter `callback` is JavaScript code to execute following the running of the CSPro logic. Due to threading issues with the embedded browser, if the user-defined function you call may display UI elements as part of its operations, you must use the asynchronous version.

`CSPro.getAsyncResult()`

Returns the result of the last completed asynchronous method (CSPro.runLogicAsync or CSPro.invokeAsync). This method executes synchronously.

These methods are available when showing a HTML dialog using the `htmldialog` function:

`CSPro.getInputData()`

Returns the string passed to the `htmldialog` function as the `inputData` argument. This method executes synchronously.

`CSPro.setDisplayOptions(displayOptions)`

Sets the display options used by the `htmldialog` function. The parameter `displayOptions` is a JSON string containing one or more of the display options used by the function. This method executes asynchronously.

`CSPro.returnData(result)`

Sets the result of the `htmldialog` function and closes the dialog. The parameter `result` is a string that will be returned to CSPro logic as the result of the `htmldialog` function. This method executes asynchronously.

This method is occasionally available to run operations that only exist in some contexts:

`CSPro.do(action, token = undefined)`

Runs an operation with an optional token and optionally returns a result. The parameter `action` is a string identifying the operation to run. This method executes synchronously.

At present there is only one command, which is recognized by the embedded browser used by `view`:

- `"close"`: Closes the browser (as though the user hit the OK button on Windows or the back button on Android).

Example (Question Text)

Question text HTML:

```
<p>What is ~~NAME~~'s occupation?</p>
<p><a href="javascript:CSPro.runLogicAsync('ViewOccupationList();');">(view full list of
occupations)</a></p>
```

CSPro logic:

```
function ViewOccupationList()
    view("ISCO.pdf");
end;
```

Example (HTML Shown Using View)

HTML:

```
<h1>District Reports</h1>

<p>Select a province or district report to view:</p>

<ul>
  <li><a href="javascript:CSPro.runLogicAsync('ShowDistrictReport(1);');">Artesia</a>
    <ul>
      <li><a href="javascript:CSPro.runLogicAsync('ShowDistrictReport(1, 1);');">Dongo</a></li>
      <li><a href="javascript:CSPro.runLogicAsync('ShowDistrictReport(1, 2);');">Varda</a></li>
    </ul>
  </li>
</ul>

<p><a href="javascript:CSPro.do('close');">Return to Program</a></p>
```

CSPro logic:

```
function ShowDistrictReport(numeric province, numeric district = notappl)
    // generate report...
end;
```

An alternate way to call the function using a JavaScript function:

```
<script>
    function showDistrictReport(province, district) {
        let input = {
            "province": province,
            "district": district
        };
        CSPro.invokeAsync("ShowDistrictReport", JSON.stringify(input));
    }
</script>

<a href="javascript:showDistrictReport(1, 2);">Show District Report: Varna</a>
```

See also: HTML in CSPro

JSON Format

JSON, an acronym for JavaScript Object Notation, is a format for representing objects in text format using property/value pairs. The format is widely used by web-based applications, and several CSPro functions take JSON strings as inputs to functions.Details about the JSON format can be readily found online. A general overview and an example will demonstrate basic features:

- Objects are surrounded by curly brackets { }.
- Property names appear in quoted strings.
- Subsequent property/value pairs are separated by a comma.
- Numeric values can be specified directly.
- String values are specified using a properly constructed string with certain characters escaped. (The `encode` function can create such strings by using the `JsonString` option.)
- Arrays are surrounded by square brackets [ ].

JSON Example

```
{
  "csproVersion": 7.7,
  "forumUrl": "http://www.csprousers.org"
  "applicationTypes": [
    "Data Entry",
    "Batch Edit",
    "Tabulation"
  ]
}
```

See also: JavaScript Interface

Appendix

Appendix A - Installation

Hardware and Software Requirements

The minimal configuration for CSPro is:

- Pentium processor
- 512MB of RAM
- SVGA monitor
- Mouse
- 100MB of free hard drive space
- Microsoft Windows Vista, 7, 8, 10, or 11

Installing CSPro

The CSPro installer has the file name **cspro77.exe**. To install CSPro:

1. Double-click on the file.
2. Read and accept the U.S. Census Bureau's license agreement
3. Select the components that you want to install. You will generally want to install all of the components.
4. Select the folder where you want to install CSPro. You will generally want to install CSPro in the suggested directory.

After CSPro has been installed on your computer, you will have the option to run the program and/or view the release notes.

You will find a CSPro 7.7 icon on your desktop that you can double-click on when you want to run CSPro in the future.

Installing a Newer Version

Upgrading to CSPro 7.7 From CSPro 7.6 or Earlier

If you have an old version of CSPro installed on your computer, you can install CSPro 7.7 without affecting the previous version. When you have finished your conversion of applications to CSPro 7.0, you can then uninstall the previous version.

Due to internal changes within CSPro 7.7, once files have been loaded in CSPro 7.7, you may no longer be able to load them in previous versions of CSPro.

Updating a Previous Version of CSPro 7.7

If you are updating a previous version of CSPro 7.7, the installer will remove the previous version and replace it with the

new version. It will not change any applications or other files that you have created.

Uninstalling CSPro

There are two ways to uninstall CSPro. The uninstaller will remove all registry entries and CSPro system files. It will not remove any applications or other files that you have created.

You can uninstall the program using the Windows Control Panel:

1. Using the Windows search functionality (**Windows Key+S**), type **Add or remove programs**.
2. Select CSPro from the list of programs.
3. Follow the prompts to uninstall the program.

Alternatively, you can:

1. Use Windows Explorer to browse to the CSPro application folder, which will likely be: **C:\Program Files (x86)\CSPro 7.7**.
2. Run the program **uninstall.exe**.
3. Follow the prompts to uninstall the program.

Installing Data Entry Applications

Install CSPro

Run the CSPro installation program on each computer. The installation setup can be run from removable media such as a CD or USB (flash) drive, or it can be placed on a networked drive.

Install the CSPro Data Entry Application

Once the data entry application has been tested and finalized, use the Pack Application tool to create a single file that contains all the essential application files. This file can be placed in a folder either on a network drive for multiple users or on each data entry operator's local hard drive. The application files then need to be extracted (unzipped) from the packed file.

If the application is on a network drive, multiple users can access it at the same time. Normally, only one person has access to an application on a local hard drive. The data file for an application can also be placed on a network drive or the operator's local hard drive, though two or more data entry operators cannot access the same data (or lookup) file at the same time.

Another way to distribute a data entry application is to compile it into a single combined file that contains all elements of the program. This file cannot be edited and provides security during the data entry operation. For this approach, read about binary data entry applications.

For information about data entry applications in general, see the Data Entry User's Guide.

Appendix B - Keys Summary

Data Dictionary Keys

Shortcuts Common Throughout CSPro

Shortcut Key	Description
Ctrl + R	Run the application.
Ctrl + Shift + R	Run the application without prompting for file associations.
Ctrl + J	Hide the tree (view full screen).
Ctrl + T	Show names instead of labels in tree.
Ctrl + Shift + T	Cycle between tabbed views in the tree.

Shortcuts Specific to the Dictionary Editor

Shortcut Key	Description
Ctrl + A	Add a new level, record, item, value set, or value.
Ctrl + I	Insert a new object above the selected object.
Ins	Insert a new object above the selected object.
Del	Delete the selected object.
Esc	Cancel edits for the object (if in editing mode).
Ctrl + Enter	Finish edits for the object and move on to the next object.
Ctrl + M	Modify the selected object.
Ctrl + D	Add notes about the selected object.
Ctrl + G	Generate a value set for an item.
Ctrl + L	Show or hide the data file layout view.
Ctrl + Alt + V	Pastes a value set as a linked value set.
Alt + Enter	Show or hide the dictionary property panel.

Shortcuts Common Throughout Windows

Shortcut Key	Description
F1	Launch the help system.
Ctrl + N	Create a new document.
Ctrl + O	Open an existing document.
Ctrl + S	Save the active document.
Ctrl + C	Copy the selected object or text to the clipboard.
Ctrl + X	Cut the selected object or text to the clipboard.
Ctrl + V	Paste into the document the object or text on the clipboard.
Ctrl + Z	Undo the last action.
Ctrl + Y	Redo the previously undone action.
Ctrl + F4	Close the active document.
Alt + F4	Exit the application.
Ctrl + Tab	Change focus to the next window.
Alt + F6	Change focus to the next window.
Shift + Alt + F6	Change focus to the previous window.

See also: Data Entry Keys, Batch Edit Keys, Tabulation Keys

Data Entry Keys

Shortcuts Common Throughout CSPro

Shortcut Key	Description
Ctrl + R	Run the application.
Ctrl + Shift + R	Run the application without prompting for file associations.
Ctrl + J	Hide the tree (view full screen).
Ctrl + T	Show names instead of labels in tree.
Ctrl + Shift + T	Cycle between tabbed views in the tree.

Shortcuts Specific to the Forms Designer

Shortcut Key	Description
Ctrl + L	View the logic for the currently selected item. Switches to logic view if no item selected.
Ctrl + M	View the form for the currently selected item. Switches to form view if no item selected.
Ctrl + Q	View the CAPI question text for the currently selected item. Switches to CAPI text view if no item selected.
Ctrl + G	Generate a set of forms based on the dictionary.
Ctrl + A	Add a form to the current level.
Alt + Enter	Modify the field properties for the currently selected item(s).
Arrow Key	Move selected item(s) on a form.
Ctrl + Arrow Key	Move selected item(s) on a form at a faster rate.
Shift + Spacebar	Clicking on a field or a field's text within the form selects both. Ensures field and text belong to one another.
Del	Delete currently selected item(s).
Ctrl + Del	Delete the currently displayed form.
F7	Publishes an entry application (.pen).

Shortcuts Specific to the Logic Editor

Shortcut Key	Description
Ctrl + K	Compile the currently selected item's logic.
Ctrl + Space	Display autocomplete suggestions for a partially entered word.
Ctrl + 4	Insert the name of the procedure or function that is currently being edited. 4 is the key for this item ($) on many keyboards.
Ctrl + G	Go to a specific line number.
Ctrl + F	Find the specified text.
F3	Find the next occurrence of the specified text.
Shift + F3	Find the previous occurrence of the specified text.
Ctrl + H	Replace the specified text with different text.
Ctrl + A	Select all text.
Ctrl + Shift + L	View the PROC_GLOBAL logic.
Ctrl + /	Comment or uncomment the current line or block of logic.
Ctrl + D	Duplicate the current line or block of logic.
Ctrl + L	Delete the current line.
Ctrl + Del	Delete the current word.
Ctrl + Shift + F	Copy the selected logic in a format for using on the CSPro Users Forum.

Alt + F12	Display the reference window with information on the selected word.
Ctrl + Alt + Click	Display the reference window with information on the selected word.
F12	Go to the declaration of the selected word.
Ctrl + Click	Go to the declaration of the selected word.
Ctrl + Plus (+)	Increase the text font size.
Ctrl + Minus (-)	Decrease the text font size.
Alt + 0	Folds all logic.
Alt + Shift + 0	Unfolds all logic.
Ctrl + Alt + F	Toggles the current code fold level.

Shortcuts Specific to the CAPI Question Text Editor

Shortcut Key	Description
Ctrl + B	Bold text.
Ctrl + I	Italicize text.
Ctrl + U	Underline text.

Shortcuts Common Throughout Windows

Shortcut Key	Description
F1	Launch the help system.
Ctrl + N	Create a new document.
Ctrl + O	Open an existing document.
Ctrl + S	Save the active document.
Ctrl + C	Copy the selected object or text to the clipboard.
Ctrl + X	Cut the selected object or text to the clipboard.
Ctrl + V	Paste into the document the object or text on the clipboard.
Ctrl + Z	Undo the last action.
Ctrl + Y	Redo the previously undone action.
Ctrl + F4	Close the active document.
Alt + F4	Exit the application.
Ctrl + Tab	Change focus to the next window.
Alt + F6	Change focus to the next window.
Shift + Alt + F6	Change focus to the previous window.

See also: Data Dictionary Keys, Batch Edit Keys, Tabulation Keys

Batch Edit Keys

Shortcuts Common Throughout CSPro

Shortcut Key	Description
Ctrl + R	Run the application.
Ctrl + Shift + R	Run the application without prompting for file associations.
Ctrl + J	Hide the tree (view full screen).
Ctrl + T	Show names instead of labels in tree.
Ctrl + Shift + T	Cycle between tabbed views in the tree.

Shortcuts Specific to the Logic Editor

Shortcut Key	Description
Ctrl + K	Compile the currently selected item's logic.
Ctrl + Space	Display autocomplete suggestions for a partially entered word.
Ctrl + 4	Insert the name of the procedure or function that is currently being edited. 4 is the key for this item ($) on many keyboards.
Ctrl + G	Go to a specific line number.
Ctrl + F	Find the specified text.
F3	Find the next occurrence of the specified text.
Shift + F3	Find the previous occurrence of the specified text.
Ctrl + H	Replace the specified text with different text.
Ctrl + A	Select all text.
Ctrl + Shift + L	View the PROC_GLOBAL logic.
Ctrl + /	Comment or uncomment the current line or block of logic.
Ctrl + D	Duplicate the current line or block of logic.
Ctrl + L	Delete the current line.
Ctrl + Del	Delete the current word.
Ctrl + Shift + F	Copy the selected logic in a format for using on the CSPro Users Forum.
Alt + F12	Display the reference window with information on the selected word.
Ctrl + Alt + Click	Display the reference window with information on the selected word.
F12	Go to the declaration of the selected word.
Ctrl + Click	Go to the declaration of the selected word.
Ctrl + Plus (+)	Increase the text font size.
Ctrl + Minus (-)	Decrease the text font size.
Alt + 0	Folds all logic.
Alt + Shift + 0	Unfolds all logic.
Ctrl + Alt + F	Toggles the current code fold level.

Shortcuts Common Throughout Windows

Shortcut Key	Description
F1	Launch the help system.
Ctrl + N	Create a new document.
Ctrl + O	Open an existing document.
Ctrl + S	Save the active document.
Ctrl + C	Copy the selected object or text to the clipboard.
Ctrl + X	Cut the selected object or text to the clipboard.
Ctrl + V	Paste into the document the object or text on the clipboard.
Ctrl + Z	Undo the last action.
Ctrl + Y	Redo the previously undone action.
Ctrl + F4	Close the active document.
Alt + F4	Exit the application.
Ctrl + Tab	Change focus to the next window.
Alt + F6	Change focus to the next window.
Shift + Alt + F6	Change focus to the previous window.

See also: Data Dictionary Keys, Data Entry Keys, Tabulation Keys

Tabulation Keys

Shortcuts Common Throughout CSPro

Shortcut Key	Description
Ctrl + R	Run the application.
Ctrl + Shift + R	Run the application without prompting for file associations.
Ctrl + J	Hide the tree (view full screen).
Ctrl + T	Show names instead of labels in tree.
Ctrl + Shift + T	Cycle between tabbed views in the tree.

Shortcuts Specific to the Table Designer

Shortcut Key	Description
Ctrl + B	View the table's tally attributes.
Ctrl + D	Show or hide the table's hidden parts.
Alt + A	Edit the area name options.
Ctrl + A	Select all cells.
Ctrl + Shift + C	Fast copy a populated table (for Excel).
Alt + Shift + Left	Go to the previous table.
Alt + Shift + Right	Go to the next table.
Ctrl + L	View the logic for the tables.
Ctrl + M	View the tables (from logic view).

Shortcuts Specific to the Table Print View

Shortcut Key	Description
Ctrl + G	Go to a specific page or table.
Ctrl + Plus (+)	Zoom in.
Ctrl + Minus (-)	Zoom out.
Alt + Home	Go to the first page of the first table.
Alt + Left	Go to the previous page.
Alt + Right	Go to the next page.
Alt + End	Go to the last page of the last table.

Shortcuts Specific to the Logic Editor

Shortcut Key	Description
Ctrl + K	Compile the currently selected item's logic.
Ctrl + Space	Display autocomplete suggestions for a partially entered word.
Ctrl + 4	Insert the name of the procedure or function that is currently being edited. 4 is the key for this item ($) on many keyboards.
Ctrl + G	Go to a specific line number.
Ctrl + F	Find the specified text.
F3	Find the next occurrence of the specified text.
Shift + F3	Find the previous occurrence of the specified text.
Ctrl + H	Replace the specified text with different text.
Ctrl + A	Select all text.
Ctrl + Shift + L	View the PROC GLOBAL logic.
Ctrl + /	Comment or uncomment the current line or block of logic.
Ctrl + D	Duplicate the current line or block of logic.
Ctrl + L	Delete the current line.

Ctrl + Del	Delete the current word.
Ctrl + Shift + F	Copy the selected logic in a format for using on the CSPro Users Forum.
Alt + F12	Display the reference window with information on the selected word.
Ctrl + Alt + Click	Display the reference window with information on the selected word.
F12	Go to the declaration of the selected word.
Ctrl + Click	Go to the declaration of the selected word.
Ctrl + Plus (+)	Increase the text font size.
Ctrl + Minus (-)	Decrease the text font size.
Alt + 0	Folds all logic.
Alt + Shift + 0	Unfolds all logic.
Ctrl + Alt + F	Toggles the current code fold level.

Shortcuts Common Throughout Windows

Shortcut Key	Description
F1	Launch the help system.
Ctrl + N	Create a new document.
Ctrl + O	Open an existing document.
Ctrl + S	Save the active document.
Ctrl + C	Copy the selected object or text to the clipboard.
Ctrl + X	Cut the selected object or text to the clipboard.
Ctrl + V	Paste into the document the object or text on the clipboard.
Ctrl + Z	Undo the last action.
Ctrl + Y	Redo the previously undone action.
Ctrl + F4	Close the active document.
Alt + F4	Exit the application.
Ctrl + Tab	Change focus to the next window.
Alt + F6	Change focus to the next window.
Shift + Alt + F6	Change focus to the previous window.

See also: Data Dictionary Keys, Data Entry Keys, Batch Edit Keys

Appendix C - Menu Summary

CSPro Menu

The following is a summary of the menu options available throughout CSPro:

File

New	Create a new application.
Open	Open an existing application.

Tools

Data Viewer	View data files.
Text Viewer	View text or data files.
Table Viewer	View CSPro tables.
Table Retrieval	Retrieve tables from a data set.
Tabulate Frequencies	Tabulate frequency distributions for file contents.
Sort Data	Sort cases or records.
Export Data	Export data in various formats.
Reformat Data	Reformat data using two dictionaries.
Compare Data	Compare contents of two similar data files.
Concatenate Data	Join text or data files one after the other.
Table Retrieval Setup	Setup table retrieval database.
Pack Application	Pack entire CSPro application into a ZIP file.
Index Data	Identifies duplicate cases in a data file.
Excel to CSPro	Convert data from Excel to CSPro.
PFF Editor	Edit a PFF file in an interactive environment.
Production Runner	Automate the running of multiple CSPro applications.
Operator Statistics Viewer	View operator statistics files.
Save Array Viewer	View and modify saved arrays.
Font Preferences	Change the font preferences for the CSPro Designer.

Window

Cascade	Arrange windows in an overlapping fashion.
Tile Top to Bottom	Arrange windows one above the other.
Tile Side by Side	Arrange windows one beside the other.

Help

Help Topics	Get help on current application.
Examples Folder	Open the folder containing some CSPro examples.
Mailing List Signup	Sign up to the mailing list to receive CSPro news updates.
Technical Support	Get information about receiving technical support.
Android CSEntry Application	Open Google Play to download the Android application.
About	Get information about the software.

Data Dictionary Menu

The following is a summary of the menu options available while editing dictionaries:

File

New	Create a new application.
Open	Open an existing application.
Close	Close an application.
Save	Save an application.
Save As	Save the current dictionary to a new file name.
Add Files	Insert a file in an existing application.
Drop Files	Drop a file from an existing application.
CSPro Settings	Clears saved credentials.
Page Setup	Change headers, footers, and margins for printed pages.
Print Setup	Change orientation and paper size for printed pages.
Print Preview	Preview the printed pages.
Print	Print all or part of a document.

Edit

Undo	Undo dictionary changes.
Redo	Redo dictionary changes.
Cut	Copy selected dictionary element to clipboard and delete it.
Copy	Copy selected dictionary element to clipboard.
Paste	Paste dictionary element on clipboard to selected location.
Modify	Edit the selected dictionary element.
Add	Add a dictionary element at the end of the list.
Insert	Insert a dictionary element at the selected location.
Delete	Delete selected dictionary element.
Languages	Define the set of languages used in the dictionary.
Security Options	Handle settings for data modification, exportation, and password storage.
Relations	Define relations between items and records.
Notes	Edit notes for selected dictionary element.
Find	Find a label or name with the specified text.
Convert to Subitems	Convert selected items to subitems and insert the item that contains them.
Flatten Occurrences	Convert a multiply occurring item to many singly occurring items.
Occurrence Labels	Edit the occurrence labels of a multiply occurring record or item.
Generate Value Set	Generate a value set of intervals of a numeric data item.
Dictionary Macros	Access special functionality for the dictionary.

View

Names in Trees	Show names instead of labels in trees.
Append Labels to Names in Tree	Show both labels and names in trees.
Full Screen	Hide the trees and show full screen view.
Aliases	View all aliases created within the dictionary.
Dictionary Analysis	Generate analyses about the dictionary contents.
Layout	Show record layout of file in the window.
Property Panel	Access and modify additional dictionary properties.

Options

Relative Positions	Select whether items stay next to each other with no gaps.
ZeroFill Default 'Yes'	Select whether numeric data items will have leading zeros.

DecChar Default 'Yes'	Specifies whether the item should be stored in the data file with an explicit decimal character.

See also: CSPro Menu

Data Entry Menu

The following is a summary of the menu options available while editing data entry applications:

File

New	Create a new application.
Open	Open an existing application.
Close	Close an application.
Save	Save an application.
Save As	Save the active application with a new name.
Add Files	Add a dictionary or form to an existing application.
Drop Files	Drop a dictionary or form from an existing application.
CSPro Settings	Clear cached credentials from your computer.
Compile	Compile the logic in the application.
Run	Run the application.
Run as Batch	Run the application after finishing keying data for a file.
Publish Entry Application	Generate a binary (compiled) version of the application.
Publish and Deploy	Generate a binary (compiled) version of the application and publish it to the location cited in the deploy script.

Edit

Undo	Undo the most recent change.
Redo	Redo the last undo.
Cut	Copy selected element to clipboard and delete it.
Copy	Copy selected element to clipboard.
Paste	Paste element on clipboard to selected location.
Add Form	Add a form to the application.
Delete Form	Delete a form from the application.
Generate Forms	Generate forms using the dictionary.
Delete	Delete selected objects.
Find	Find text in the procedures.
Find Next	Find the next occurrence of text in the procedures.
Replace	Replace text with new text in the procedures.
Symbol Analysis	View information about where each dictionary symbol, form symbol, and user-defined function is used in an application.
Format Logic	Format the logic using a defined CSPro code formatting style.

View

Box Toolbar	Show or hide the box drawing toolbar.
Names in Trees	Show names instead of labels in trees.
Append Labels to Names in Tree	Show both labels and names in trees.
Full Screen	Hide the trees and show full screen view.
Form	Show forms in the right-hand window.
Logic	Show logic procedures in the right-hand window.

CAPI Questions	Show the screen to enter question text.
Code Folding	Options to fold the application's logic.
OnKey Character Map	Calculate `OnKey` and `OnChar` codes.
User and Configuration Settings	Open a dialog that allows you to edit the user and configuration settings.

Options

Application Properties	Modify a variety of application properties.
Data Entry	Change the data entry options.
Drag	Change the drag options.
Field Properties	Change the field properties for two or more fields at a time.
Data Sources	Change the data source options.
Synchronization	Set the simple synchronization parameters.
Map Case Listing	Allow already-captured cases to be displayed on a map using the case's latitude and longitude coordinates.
Default Text Font	Change the default text font settings.
Field Font	Change the field font settings.
Deprecation Warnings	Modify the setting for the kind of deprecation warnings to display during compilation.

Align

Left	Position to left-most item.
Center	Center items as a group.
Right	Position to right-most item.
Top	Position to top-most item.
Mid	Align mid-points of items as a group.
Bottom	Position to bottom-most item.
Evenly Horizontal	Space evenly left to right.
Evenly Vertical	Space evenly top to bottom.

CAPI

CAPI Languages	Define the set of CAPI languages used in application.
Edit Styles	Define custom styles to use in the question window.
Bold	Switch between **bold** and non-bold text.
Italic	Switch between *italicized* and non-italicized text.
Underline	Switch between <u>underlined</u> and non-underlined text.
Color	Change text color.
Left	Align text to left of question window.
Center	Align text to center of question window.
Right	Align text to right of question window.
Bullets	Convert text to a bulleted list.
Numbering	Convert text to a numbered list.
Insert Image	Insert an image into the question text.
Text Direction LTR	Mark the paragraph as for left-to-right text.
Text Direction RTL	Mark the paragraph as for right-to-left text.
Question Text Macros	Access special functionality for the question text.

See also: CSPro Menu, Data Dictionary Menu

Batch Editing Menu

The following is a summary of the menu options available while editing batch edit applications:

File

New	Create a new application.
Open	Open an existing application.
Close	Close an application.
Save	Save an application.
Save As	Save the active application with a new name.
Add Files	Add a dictionary to an existing application.
Drop Files	Drop a dictionary from an existing application.
CSPro Settings	Clear cached credentials from your computer.
Compile	Compile the logic in the application.
Run	Run the application.

Edit

Undo	Undo latest cut/copy/paste operations.
Redo	Redo the latest undo operations.
Cut	Cut logic and place it on the clipboard.
Copy	Copy logic and place it on the clipboard.
Paste	Paste logic from the clipboard.
Find	Find text in the logic.
Find Next	Find the next occurrence of text in the logic.
Replace	Replace text with new text in the logic.
Symbol Analysis	View information about where each dictionary symbol, form symbol, and user-defined function is used in an application.
Format Logic	Format the logic using a defined CSPro code formatting style.

View

Names in Trees	Show names instead of labels in trees.
Append Labels to Names in Tree	Show both labels and names in trees.
Full Screen	Hide the trees and show full screen view.
Code Folding	Options to fold the application's logic.
User and Configuration Settings	Open a dialog that allows you to edit the user and configuration settings.

Options

Application Properties	Modify a variety of application properties.
Custom Order	Allow user defined order of editing.
Deprecation Warnings	Modify the setting for the kind of deprecation warnings to display during compilation.

See also: CSPro Menu, Data Dictionary Menu

Tabulation Menu

The following is a summary of the menu options available while editing tabulation applications:

File

New	Create a new application.
Open	Open an existing application.

Close	Close an application.
Save	Save an application.
Save As	Save the active application with a new name.
Save Tables	Save current table results in a file.
Add Files	Add a dictionary to an existing application.
Drop Files	Drop a dictionary from an existing application.
CSPro Settings	Clear cached credentials from your computer.
Load Preferences	Load stored table formatting preferences.
Save Preferences	Save table formatting preferences.
Run	Run the application.
Run in Parts	Run Tabulate, Consolidate, or Format process.
Page Setup	Change headers, footers, and margins for printed pages.
Print Setup	Change orientation and paper size for printed pages.
Print Preview	Preview the printed pages.
Print	Print all or part of a document.

Edit

Add Table	Add a table at the end.
Insert Table	Insert a table at the current location.
Delete Table	Delete the current table.
Tally Attributes Table	Edit the universe or filter of tabulation.
Area	Enable or change tabulation by geographic area.
Format Table	Edit formats that apply to the entire table.
Format Application	Edit formats that apply to the entire application.
Format Print Table	Edit formats for the printed tables.
Go To	Go to a particular page, table, or area.
Copy	Copy selected tables cells and headings to the clipboard.
Copy Cells Only	Copy only the selected table cells to the clipboard.
Copy Table Spec	Copy table specification to the clipboard.
Paste Table Spec	Paste table specification into application.
Select All	Select the entire table.
Cancel Selection	Cancel the currently selected cells.
Exclude Table from Run	Exclude the selected table from the run.
Exclude All But This	Exclude all tables but the selected table from the run.
Include All Tables	Include all tables in the run.
Preferences	Change application format and tally preference.

View

Names in Trees	Show names instead of labels in trees.
Append Labels to Names in Tree	Show both labels and names in trees.
Full Screen	Hide the trees and show full screen view.
Hidden Parts	Show or hide hidden parts of a table.
Previous Table	Show the previous table.
Next Table	Show the next table.
Zoom In	Zoom in on a print preview.
Zoom Out	Zoom out on a print preview.
Facing Pages	Show facing pages (two) on a print preview.
Multiple Pages	Show selected number of pages in print preview.
First Page	Show first print page of first table.

Previous page	Show previous print page.
Next page	Show next print page.
Last page	Show last print page of last table.

Options

Application Properties	Modify a variety of application properties.

See also: CSPro Menu, Data Dictionary Menu

Appendix D - Toolbar Summary

CSPro Toolbar

The CSPro toolbar is displayed across the top of the window, below the menu bar. It provides quick mouse access to many features used in CSPro.

Click To

Create a new application.
Open an existing application
Get help.

The CSPro toolbar only appears when CSPro is opened without specifying an application or file. When applications or files are open, the toolbar corresponding to the contents of the right-hand screen appears.

Data Dictionary Toolbar

The Data Dictionary toolbar is displayed across the top of the window, below the menu bar. It provides quick mouse access to many features used in the Data Dictionary. It is available whenever the right-hand screen is displaying dictionary items

Click To

Create a new dictionary.
Open a dictionary.
Save a dictionary.

Set up page margins and headings for printing.
Preview contents of the dictionary.
Print contents of the dictionary.

Undo the last change to dictionary.
Redo last undo.
Cut the selected records, items, or values to the clipboard.
Copy the selected records, items, or values to the clipboard.
Paste the contents of the clipboard to the current position.

Add levels, records, items, values sets, or values.
Insert levels, records, items, values sets, or values.
Delete levels, records, items, value sets, or values.

Edit Notes for dictionary, level, record, item, value set, or value.
Find a label or a name in the dictionary.

Show the Layout window.

Show last Dictionary window.
Show last Forms window.
Show last Batch Edit window.
Show last Tabulation window.

Get Help.

Data Entry Toolbar

The Forms Designer toolbar is displayed across the top of the window, immediately below the menu bar. The toolbar provides quick mouse access to many of the often-used features found in the Forms Designer.

Click To

Create a New application.
Open an application.
Save an application.
Compile the logic (code) of your data entry application.
Run the current data entry application (i.e., start up CSEntry).

Undo the latest changes.
Redo the latest changes.

Cut the selected elements to the clipboard.
Copy the selected elements to clipboard.
Paste the contents of the clipboard to the form.

Delete the currently selected item(s).
Find text in logic.

Toggle between selecting item(s) or drawing boxes .
View the forms
View the logic
View the CAPI question

Show last Dictionary window.
Show last Forms window.
Show last Batch Edit window.
Show last Tabulation window.

Get Help.

Batch Editing Toolbar

The Edit Designer toolbar is displayed across the top of the window, immediately below the menu bar. The toolbar provides quick mouse access to many of the often-used features found in the Edit Designer.

Click To

Create a New application
Open an application
Save an application

Compile the logic (code) of your application
Run the current batch edit application

Undo the latest text changes.
Redo the latest text changes.

Cut the selected logic to the clipboard.
Copy the selected logic to clipboard.
Paste the contents of the clipboard to the logic.
Find text in the logic

Launch Text Viewer

Show last Dictionary window.
Show last Forms window.
Show last Batch Edit window.
Show last Tabulation window.

Get Help.

Tabulation Toolbar

The Tabulation toolbar is displayed across the top of the window, immediately below the menu bar. The toolbar provides quick mouse access to many of the more frequently-used features found in Tabulation.

Click To

Create a New application.
Open an existing application.
Save an application.
Save tables.

Setup page margins and headings for printing.
Preview printing of tables.

Print tables.

Run a tabulation.

Add a table.
Insert a table.
Delete the current table.

Specify Tally Attributes for the table.
Define Area Processing variables.

Copy a selection from a table.

Show Previous Table.
Show Next Table.

Show last Dictionary window.
Show last Forms window.
Show last Batch Edit window.
Show last Tabulation window.

Get Help.

Appendix E - Application Properties

Application Properties

Data entry, batch edit, and tabulation applications have a variety of properties that can be modified. To view or modify these properties, from the **Options** menu, select **Application Properties**. A dialog shows the properties organized by topic. The property sections are listed on the left, with the various options for that section on the right. The button **Reset Section Properties** resets the properties for the current section to their default settings.

Application properties are saved to a file with the extension .csprops. When a new application is created, with all properties at their default state, no application properties file is associated with the application. When you modify any properties away from their default state, you will be prompted to associate an application properties file with your application. CSPro will suggest a default filename based on your application filename. You can use this filename, select an existing application properties file, or you can provide a different filename.

Each properties section is documented in a separate help page:

Section

Paradata

Mapping

- Esri Tile Provider
- Mapbox Tile Provider

Advanced Features

Paradata Properties

To view options related to paradata collection, select Application Properties from the **Options** menu of a data entry, batch edit, or tabulation application, and then select the **Paradata** section.

Event Collection Options

A paradata event contains information about some aspect of an application's processing. You have three options for paradata collection:

- **All events**: This enables paradata collection for all events.
- **Some events**: This enables paradata collection for some events. You can select the events that you would like to collect by checking or unchecking the checkbox next to the name of each event. You may want to disable some events to reduce the size of the paradata log or if you are concerned about collecting sensitive information.
- **No events**: This disables paradata collection.

The following is a summary of each kind of event that CSPro can collect:

- **Data Source**: Information about interactions with CSPro data files, including what files were used, what cases were loaded and saved, etc.
- **Device State**: Details about the data collection device, including indicators about whether GPS, Bluetooth, and Wi-Fi were enabled; what mobile network the device was using; etc.
- **External Application**: Information about external applications (or CSPro data entry applications) launched from

CSPro.

- **Field Entry**: Details about each field entered, including how long the interviewer spent answering the question, how the program reached that field, etc.
- **Field Movement**: Information about each field movement, including how the program reached and exited each field.
- **Field Validation**: Details about CSPro's validation of each entered field, including information on which value set was used and whether or not the field was successfully validated.
- **GPS**: Information about GPS collection, including the duration to take fresh GPS readings.
- **Impute**: Details about which items were imputed, including the initial and imputed values. Only numeric imputations are recorded.
- **Language Change**: Information about which language was initially used and records of language changes during the course of data collection.
- **Message**: Details about any error message triggered during data collection.
- **Note**: Information about notes left by interviewers (or via logic) during the execution of the program.
- **Operator Selection**: Details about selections made by the operator as a result of requests coming from functions such as `accept` or the `userbar`.
- **Property**: Information about what CSPro settings were used, or changed, during the program's run.

You can explore these events in more detail using the Paradata Viewer's Table Metadata feature.

For advanced users writing SQL queries on the paradata tables, note that the base "event" table links to each of these event tables, and that some of these events link to other tables with names that end in "_info" or "_instance."

Partial Collection Options

Some of the events listed above can store sensitive information in the paradata log, which is why you can disable the collection of this information while still storing the non-sensitive portions of the event:

- **Field values**: The Field Validation event can store each value entered into a field. This can be useful if you want to look at the frequency of valid and invalid values, but if you do not want such data stored in the paradata log, you should disable this. The Note event also uses this option to determine if note text should be stored in the paradata log.
- **GPS coordinates**: The GPS event can store the coordinates (latitude and longitude) and altitude of the interviewers. If you do not want this data stored in the paradata log, you should disable the storage of these three values, though other information (e.g., accuracy) will still be stored.
- **Cases loaded via logic loops**: This is not an option related to sensitive information, but if your application loops through external data files in logic (using `forcase` or `loadcase`), the paradata log will become very large quickly due to a preponderance of Data Source events. You can disable this information if you do not plan to analyze it.
- **Initial property values**: If this option is enabled, the initial property settings will be stored as Property events in the paradata log when your application begins. This can lead to a large amount of paradata that is potentially not useful. If disabled, only information about changes to properties will be stored.

The default setting is for all of these options to be disabled.

Background Events

Most events are triggered by some action (e.g., an error message appearing), but two events can occur in the background, even if the interviewer is not doing anything. These background events only occur in data entry applications:

- **Device state**: Information about the device as described above in the Device State event.
- **GPS location**: Unless a GPS reading is in progress, the device's GPS unit will be turned on and CSEntry will attempt to take a GPS reading. Once a GPS reading with accuracy 10 meters or less is taken, or once two minutes have elapsed, the GPS unit will be turned off.

The value listed indicates the number of minutes between events occurring, or 0 if the event should be disabled. For example, a value of 30 for GPS location means that CSEntry will attempt to store the GPS coordinates of the interviewer every half hour.

See also: Paradata

Mapping Properties

To view options related to mapping, select Application Properties from the **Options** menu of a data entry, batch edit, or tabulation application, and then select the **Mapping** section.

Mapping

The mapping properties allow control of the following mapping options:

- The **Coordinate Display** allows the toggling of the coordinates between decimal and DMS (degrees, minutes, seconds).
- The **Default Base Map** allows the selection of the base map (normal, hybrid, satellite, terrain, or none). Otherwise, local files can be selected to display offline maps.
- The **Tile Provider** allows the selection of the source of base maps (Esri or Mapbox).

Esri Tile Provider

If Esri is selected as the tile provider then the base maps will be served from their ArcGIS platform. An API key allows access to the ArcGIS platform. The API key is created once an ArcGIS Developer account has been created. The **Default Base Map** option consists of five different properties. Excluding None, each property is a descriptive label that is associated with a tile layer. By default the associations are:

Name	Tile Layer
Normal	Streets
Hybrid	NationalGeographic
Satellite	Imagery
Terrain	Topographic

Press the **Preview Map** button to view changes.

Mapbox Tile Provider

If Mapbox is selected as the tile provider then the base maps will be served from the Mapbox servers. An access token allows access to Mapbox's tools, APIs, or SDKs. Access tokens can be managed from the access tokens page once a Mapbox account has been created. The **Default Base Map** option consists of five different properties. Excluding None, each property is a descriptive label that is associated with a tile layer. By default the associations are:

Name	Tile Layer
Normal	mapbox/streets-v11
Hybrid	mapbox/satellite-streets-v11
Satellite	mapbox/satellite-v9
Terrain	mapbox/outdoors-v11

Press the **Preview Map** button to view changes.

Advanced Features Properties

To view options related to CSPro advanced features, select Application Properties from the **Options** menu of a data entry, batch edit, or tabulation application, and then select the **Advanced Features** section.

These options allow customization of advanced CSPro features. These features may not be available in a future release. Options include:

Use HTML dialogs: If checked, CSPro will use HTML dialogs for displaying logic functions (such as `errmsg` and `accept`) as well as other aspects of the CSPro user interface (UI). If unchecked, CSPro will use the style of dialogs used prior to CSPro 7.7. In a future release of CSPro, the old style of dialogs will be removed.

Appendix F - Converting Within IMPS or ISSA

Converting a Data Dictionary

CSPro will convert your existing IMPS 3.1, IMPS 4.1, or ISSA dictionary (any version) to the CSPro dictionary format. You can also save your CSPro dictionary file out as an IMPS 3.1 or ISSA dictionary. Either way, just do the following:

1. Click the **Tools** menu, and then click **Convert Dictionary**.
2. Choose whether you are converting between CSPro and IMPS, or CSPro and ISSA, then press **Nex**t.
3. Choose the type conversion you are performing and press **Next**.
4. Choose the file name of the dictionary you are converting.
5. Specify the file name of the dictionary you are generating.
6. Press **Finish** when ready.

Note: The Convert Dictionary tool is no longer distributed with CSPro but is instead available on www.csprousers.org.

Converting within IMPS

Convert an IMPS 3.1 (DOS) data dictionary to CSPro

- Hypens (dashes) within record and item names are changed to underlines.
- All common items in IMPS are changed to ID items in CSPro.
- As CSPro does not allow the use of subitems in the ID section, any subitems found in the IMPS common section will be converted to ID items. This could increase the length of the ID section. Therefore we advise you to check your CSPro dictionary after conversion if you receive a message about "subitems found in common, promoted to ID items". Similarly, if you receive any error messages during the conversion about overlapping items, this is most likely due to the use of subitems in the common block of IMPS.
- If you have used "NR" as a value name in your IMPS dictionary, it will be converted to the special value `missing`.
- If you have used "NA" as a value name in your IMPS dictionary, it will be converted to the special value `notappl`.

Convert an IMPS 4.1 (Windows) data dictionary to CSPro

- All hierarchy information is lost.
- All other comments as listed under *IMPS 3.1 to CSPro* apply.

Convert a CSPro data dictionary to IMPS 3.1 (DOS)

- Underlines within record and item names are changed to hyphens (dashes).
- All level record, item, and value set labels are lost.
- All value labels are truncated to 16 characters.
- Value sets, after the first, are converted to subitems.

- All notes are lost.
- All level information is lost.
- All special values (`missing`, `refused`, `notappl`, and `default`) are lost.

CSPro allows the use of long names even though IMPS only allows 16 characters. Therefore, if any of your unique names are longer than 16 characters, you will receive a message for each variable that its name will be truncated to 16.

CSPro does not support the use of short names, as described in IMPS. Therefore, if the first 8 characters of any item name in CSPro are not unique, the converted IMPS dictionary will have duplicate shorts names. You should check the converted IMPS dictionary to ensure that there are no duplicate short names.

Converting within ISSA

In an ISSA data entry application, a dictionary file contains both descriptive information about each data item (item type, item length, etc), as well as information about the form layout. However, in CSPro, we break this information into two separate files. Therefore, if you:

- Convert an ISSA data dictionary to CSPro data dictionary only, you will receive a CSPro dictionary (.dcf) only. You can choose to convert either a regular data dictionary (.dic) or a working storage dictionary (.wst).
- Convert an ISSA data dictionary to CSPro form and data dictionary, you will receive both a CSPro dictionary (.dcf) and a CSPro form file (.fmf). You can choose to convert either a regular data dictionary (.dic) or a working storage dictionary (.wst).

Converting an IMPS Data Entry Application

CSPro provides a utility to convert existing IMPS data dictionaries to CSPro. However, there is no automated tool to convert CENTRY application files. You must create the forms again in CSPro.

At any time in CSPro, you can go to the **Tools** menu option. Select **Convert Dictionary** from the menu, then choose to convert **Between CSPro and IMPS** in the opening dialog box.

CSPro is much less constrained than CENTRY in the relationship between dictionary records and data entry forms. In CSPro you can mix dictionary items from different records on the same form. See Issues to Consider When Designing a Form for more details.

In CSPro you can make the equivalent of the CENTRY "Batch," "Questionnaire," and "Record" screens. If you use this approach, you must be sure to make all the fields on the "Batch" screen persistent.

Note: The Convert Dictionary tool is no longer distributed with CSPro but is instead available on www.csprousers.org.

Converting an ISSA Data Entry Application

If you have an existing ISSA Dictionary that contains forms, CSPro provides a utility to convert it to a CSPro form file (a dictionary will be generated as well).

At any time in CSPro, you can go to the **Tools** menu option. Select **Convert Dictionary** from the menu, then choose to convert **Between CSPro and ISSA** in the opening dialog box.

Next, state that you'd like to convert from **ISSA to a CSPro Forms and Data Dictionary**. Provide the name of the original ISSA dictionary file, and the name you would like to call the CSPro form file to be generated (a CSPro dictionary file will also be created, and its name will be based on the form file name). Press **OK** when ready and the files will be created for you. You are then ready to fine-tune the layout of the forms as desired.

Note that this creates a stand-alone dictionary and form file; it does not create a data entry application. Until there exists

a data entry application, you cannot write logic for the form variables, nor can you enter data based on this form file. If you would like to generate a data entry application, proceed as you would for creating a new data entry application. When you are asked to provide the name of the form file, simply use the same form file name that was used during the conversion, and CSPro will complete the task.

Note: The Convert Dictionary tool is no longer distributed with CSPro but is instead available on www.csprousers.org.

Appendix G - Errors in Censuses and Surveys

Nature of Census and Survey Data

A census is a complete count of a given entity, whether population, housing, agricultural holdings, businesses, or other, in a given geographic area at a given time. Because of financial and time constraints, the information collected in a census is usually basic and limited. A survey is admin istered to a subset of a population to obtain detailed information about certain groups, such as school-age children or mothers, or to obtain other information, on a sample basis, about the entity in question (households, businesses, agricultural establishments, etc.).

A survey can provide valuable infor ma tion about the population being studied, but because of the limitations of samples, the results can be generalized only for relatively high-level geographic areas, such as the country as a whole, or (depending on the way in which the sample was selected) for specific regions or areas of the country (e.g., urban vs. rural). A census, however, attempts to cover the entire geographic area of the subject population, so it can provide reliable information at very low levels of geography. In addition to counts (e.g., number of people, number of housing units, number of farms, number of businesses, etc.), a census usually provides a profile of additional related characteristics such as fertility, housing quality, acreage, number of employees, and so on.

Census and survey data are used to plan for education, health facilities, administration, and other needs. In order to implement programs and activities, statistics are needed by government administrators and by private users, including businesses, industries, research organizations, and the general public. These statistics also may serve as measures of existing conditions for small areas, providing a basis for planning development programs, and perhaps establishing a basis for action.

In order to obtain an accurate census or survey, the data must be as free as possible from errors and inconsistencies. Statistics derived from 'dirty data' (that is, data which still contain errors) may produce an inaccurate profile of the country or geographic area. Therefore, before any tabulation programs are run, the data should be checked for errors and changed so that important data items are valid and consistent. This is not to say that correction of data after they are collected can compensate for poorly collected data. It is not practical (if not impossible) to produce a data file which is 100 percent error free. Every effort at accuracy should be made in all stages of the census or survey.

Errors in Censuses and Surveys

Type of Errors

Editing is the process of maximizing the quality of data, in the shortest time possible, while minimizing the introduction of new errors. The process involves a number of sequential, interrelated activities as shown below. During each activity errors may occur.

Activity	Possible Errors
Enumeration	Respondent Errors, Enumerator Errors
Field Editing	Field Checking, Office Checking
Office Coding	Miscodes
Data Capture	Miskeys, Incorrectly Scanned
Computer Editing	Logic Errors, Misallocation, Miscorrection
Tabulation	Distribution of Unknowns
Publication	Misprints

Errors in Enumeration

Two types of errors occur at the enumeration stage: the respondent sometimes errs when giving information to the enumerator either by offering what the respondent believes to be the "proper response" (as opposed to a truthful response) to the questions; or by misunderstanding the question; and the enumerator, in asking the questions, recording the responses, and reviewing entries at the conclusion of the interview, also may add errors to the data.

Little can be done to improve the quality of responses from individuals, except through publicity for the census and well-trained enumerators who explain the purpose of the census and reasons for asking the various questions. The quality of enumerators and enumerator training often can be the crucial factors in census processing. Enumerators must be properly trained in all relevant aspects of census procedures and made to understand why their part of the census is important, and how the enumeration fits in with the other stages of the census. Pretesting should be used to eliminate problems in the questionnaires and materials, and to help enumerators obtain the data and complete the enumeration in the allotted time. Also, since enumerators come from many different backgrounds and have varying levels of education and training, training must be developed to make certain the enumerators know how to ask the questions to obtain an unbiased response.

Errors in Field Editing

A general rule for editing could be: the closer that corrections are made to the source of the data the higher the quality of the final statistics. Field editing is, therefore, perhaps the most important stage in the census processing. Supervisors, in addition to training enumerators, must also be able to collect the data themselves and to correct enumerator errors while the forms, the respondents, and the enumerators are still available. Questions that arise can then be answered before the questionnaires are sent to the central census office.

Once the forms leave the enumeration areas, changes can no longer easily be made with the knowledge and help of the respondents and enumerators, so other procedures must be used to cope with inaccurate, incomplete, or inconsistent data. During preliminary census office editing, checks of crucial entries must be carried out quickly to determine completeness and consistency in the collected data. Highly-aberrant forms may be sent back to the field, if time and money permit. Place codes must be checked for validity, and relationships between numbers of expected individuals as recorded on the household form, and the actual numbers of individual forms (if individual forms are used) must agree.

Errors in Coding

Precise, detailed instructions for coding in preparation for manual and computer editing must be determined after the tabulation plans are developed, but before the enumeration is actually undertaken. Back in the census office, it is no longer possible to make corrections while in contact with the respondents, so editing must be determined on the basis of assumptions about what the most probable response would be. If computer editing is possible, manual office editing should be minimized to checking for completeness.

If census data are collected on precoded questionnaires (coded either by respondents or enumerators), and machines are used to convert the information to computer-readable data, then except for the introduction of errors due to stray marks or physical problems with the questionnaires, the errors found should be minimal.

Errors may occur when the data are coded, since the coder may miscode some piece of information. If the miscode is invalid, it should be caught during the computer editing; if the code is valid but incorrect (for example, if two digits are reversed for the entry for birthplace), the computer will not note the errors, and the information will remain incorrect for the tabulations. Coders must be trained to edit according to the edit specifications, and efforts must be made to obtain and maintain quality of coders, 'weeding out' inefficient inaccurate coders. Spot checks and verification of samples from each coder can help to identify persistent coding errors.

Errors in Data Capture

Data capture is the process of converting data to a form which the computer can use. The most common method used to convert the data is keying; scanning technologies like optical character readers (OCR) and optical mark readers (OMR) are increasingly used, and while these methodologies appear to offer lower costs and reduced time when compared with operator-based entry, in fact each presents a separate set of difficulties, different from those associated with manual keying, that must be overcome, and which can nullify any perceived gains.

With keyer-based entry, errors are introduced into the data through miskeying. Verification (rekeying or double-keying) can reduce these errors. A system called "intelligent data entry" (IDE) may be used to prevent invalid entries from ever getting into the system. An IDE system ensures that the value for each field or data item is within the permissible range of values for that item. Such a system increases the chance that the data entry operator will key in reasonable data and

relieves some of the burden on later stages of the data preparation process.

With scanning technologies, errors can be more insidious, and proper verification is more time-consuming, because it will require either manual comparison between the information captured by the scanner and the forms, or the establishment of a separate keying operation so that keyed output can be compared with scanned output. It is extremely important not to assume that the use of "advanced" technology reduces or eliminates the need for verification; errors can and will occur, so they must be caught early in the cycle of capture so that corrective measures (technological or manual) may be applied. If this is not done, systemic error can corrupt the data beyond repair.

Errors in Computer Editing

The high degree of accuracy and uniformity obtainable with computer editing cannot be achieved through manual editing. In computer editing, range checks and within-record consistency checks can easily be made; between-record edits can also be done if the computer programs have this capability; and unknown information can be allocated automatically. If an allocation method is used, as much of the original information as possible must be retained.

Computer edit checks have been used in almost all censuses carried out since the 1980 round. For proper implementation of this tool, there must be good communication between the subject-matter specialists and the programmers. Subject-matter specialists should write complete and clear edit specifications. Programmers should review these specifications and work closely with subject-matter specialists to resolve questions or difficulties in implementing the specifications. Programmers also should make sure that subject-matter specialists are involved in testing the edit programs, that is, in providing test data and reviewing the outputs to insure that all the necessary edits were included in the specifications. It is the programmers' responsibility to produce an edit program free of errors. If these programs are inadequately thought out or not completely tested, existing errors in the data may not be corrected and even more errors may be introduced.

Errors in Tabulation

Errors can occur at the tabulation stage due to improper programming or use of unknown information. Errors at this stage are difficult to correct without introducing new errors.

Errors in Publication

Errors can occur at the publication stage through lack of inter-tabulation checking, or through printing errors. If errors are carried through all stages of the process to publication, they will be apparent and the results will be of questionable value. Most importantly, obvious errors at this stage diminish the credibility of the organization presenting the data. Finally, it is very important that error analysis be done to help in interpreting the extent and kind of errors in the census and to aid in preparing for future censuses and surveys.

Appendix H - File Types

File Types

It is highly recommended that you enable the viewing of file extensions in Windows when using CSPro. By default, Windows hides the extensions of known types, and this can be confusing when working with CSPro files. To view the extensions, open **Explorer**, select the **View** tab and then the **Options** button. On the **View** tab, uncheck **Hide extensions for known file types**.

Dictionary

.dcf	Data dictionary

Program Information

.pff	Parameters describing how to run any CSPro application
.pffRunner	Compilation of PFF files used by the Production Runner

Data Entry Application

.ent	Data entry application
.dcf	Data dictionary
.fmf	Forms
.ent.apc	Logic code
.ent.mgf	User-defined messages
.ent.qsf	CAPI question text
.csprops	Application properties (optional)
.pen	Compiled data entry application to run on mobile devices

Data File

.csdb	CSPro DB data file
.csdbe	Encrypted CSPro DB data file
.???	Text data file
.csidx	Index for text data files (in binary format)
.csnot	Notes (entered by the operator) for text data files
.sts	Status file, listing information about what cases have been verified or partially entered, for text data files
.log	Operator statistics
.lst	Listing of data entry activity

Batch Edit Application

.bch	Batch edit application
.dcf	Data dictionary
.ord	Order of edits
.bch.apc	Logic code
.bch.mgf	User-defined messages

.csprops	Application properties (optional)
.lst / .html / .xlsx / .csv / .csdb / .dat	Listing of batch edit results
.tbw / .lst / .html / .xlsx	Frequencies
.impute_freq.lst	Imputation frequencies
.impute_stat.csdb	Imputation statistics
.sva	Contents of saved arrays

Tabulation Application

.xtb	Tabulation application
.dcf	Data dictionary
.xts	Table specifications
.xtb.apc	Logic code
.xtb.mgf	User-defined messages
.csprops	Application properties (optional)
.anm	Area names
.lst	Listing of tabulation results
.tbw	Output tables for the Table Viewer
.tab	Table matrices file
.tabidx	Table matrices index file

Other

.cmp	Comparison specifications for the Sort Data tool
.csds	Deployment specifications for the Deploy Application tool
.cslog	Log with paradata events stored during an application's run
.csrs	Report script for templated reports
.exf	Export specifications for the Export Data tool
.fqf	Frequency specifications for the Tabulate Frequencies tool
.m4a	Recorded audio files (AAC audio in an MPEG-4 container)
.ssf	Sort specifications for the Sort Data tool
.trs	Table retrieval file
.xl2cs	Conversion specifications for the Excel to CSPro tool

Some of the file extensions were different in previous versions of CSPro:

Previous Extension	Current Extension
.app	.apc
.enc	.pen
.idx	.csidx
.not	.csnot
.tai	.tabidx

Importing Data to CSPro Format

There are occasions, such as when using lookup files, when you will want to convert data created in another application to a format that CSPro can read. This is a two-step process:

1. Convert the data to a flat file with one record per line and all records for a case organized together. (Read more about data requirements or CSPro's file structure.)
2. Create a CSPro dictionary that describes this data file.

There are many formats from which data can be converted, but converting data to CSPro format from Microsoft Excel is the most common task. There is a tool, Excel to CSPro, that can do this automatically for you. It is the preferred way of converting data. It is also possible to create a flat file yourself from within Excel. To convert data to CSPro format:

1. Open a file in Excel. We will demonstrate this conversion using the following file:

	A	B	C	D	E
1	**Date**	**Establishment**	**Amount**	**Exchange Rate**	**Dollar Amount**
2					
3	11/10/2009	Hotel Intercontinental	KES 1,500.00	0.013387	$20.08
4	11/12/2009	China Anhui Huaren Restaurant	KES 1,000.00	0.013387	$13.39
5	11/13/2009	Alliance Française	KES 100.00	0.013387	$1.34
6	11/15/2009	Kenya Forest Service	KES 150.00	0.013387	$2.01
7	11/19/2009	Waamo Pizza & Fries	KES 750.00	0.013388	$10.04

2. Remove any header information in the file. CSPro data files should contain only data and thus any metadata (e.g., column headings) should be removed:

	A	B	C	D	E
1	11/10/2009	Hotel Intercontinental	KES 1,500.00	0.013387	$20.08
2	11/12/2009	China Anhui Huaren Restaurant	KES 1,000.00	0.013387	$13.39
3	11/13/2009	Alliance Française	KES 100.00	0.013387	$1.34
4	11/15/2009	Kenya Forest Service	KES 150.00	0.013387	$2.01
5	11/19/2009	Waamo Pizza & Fries	KES 750.00	0.013388	$10.04

3. All fields must be right-justified:

	A	B	C	D	E
1	11/10/2009	Hotel Intercontinental	KES 1,500.00	0.013387	$20.08
2	11/12/2009	China Anhui Huaren Restaurant	KES 1,000.00	0.013387	$13.39
3	11/13/2009	Alliance Française	KES 100.00	0.013387	$1.34
4	11/15/2009	Kenya Forest Service	KES 150.00	0.013387	$2.01
5	11/19/2009	Waamo Pizza & Fries	KES 750.00	0.013388	$10.04

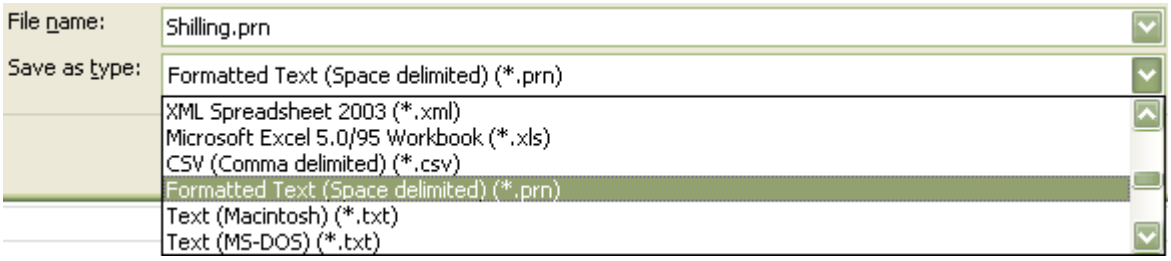
4. Remove any specialized formatting attached to cells (unless that formatting is desired in the final output):

	A	B	C	D	E
1	11/10/2009	Hotel Intercontinental	1500.00	0.013387	20.08
2	11/12/2009	China Anhui Huaren Restaurant	1000.00	0.013387	13.39
3	11/13/2009	Alliance Française	100.00	0.013387	1.34
4	11/15/2009	Kenya Forest Service	150.00	0.013387	2.01
5	11/19/2009	Waamo Pizza & Fries	750.00	0.013388	10.04

5. Set the column widths to appropriate values. When this information is finally saved, the width of each column determines the length of each item in your CSPro dictionary.

	A	B	C	D	E
1	11/10/2009	Hotel Intercontinental	1500.00	0.013387	20.08
2	11/12/2009	China Anhui Huaren Restaurant	1000.00	0.013387	13.39
3	11/13/2009	Alliance Française	100.00	0.013387	1.34
4	11/15/2009	Kenya Forest Service	150.00	0.013387	2.01
5	11/19/2009	Waamo Pizza & Fries	750.00	0.013388	10.04

6. Save As, choosing "Formatted Text (Space delimited) (*.prn)" as the output format:



Using the above example, the resulting file looks like this:

```
11/10/2009       Hotel Intercontinental  1500.00  0.013387 20.08
11/12/2009China Anhui Huaren Restaurant  1000.00  0.013387 13.39
11/13/2009           Alliance Française   100.00  0.013387  1.34
11/15/2009         Kenya Forest Service   150.00  0.013387  2.01
11/19/2009          Waamo Pizza & Fries   750.00  0.013388 10.04
```

7. Create a CSPro dictionary that describes this data. The length of each item is determined by the column width specified in Excel. Other options, such as whether a decimal character should exist, or whether numeric values should be zero-filled, can also be specified in Excel.

It is a good idea to run frequencies on each item to make sure that the dictionary you created properly describes the .prn file.

The above example is a rather simple one, but more complex imports, including importing cases with multiple records (and thus a record type), are also possible.

Locking Application Files

If distributing a CSPro application to users, one simple, though not rigorous, way of preventing the users from modifying the applications is to add the command [NoEdit] at the top of either of these files:

- .ent
- .bch
- .xtb
- .dcf
- .fmf

For instance:

```
[NoEdit]
[CSPro Application]
Version=CSPro 7.7
```

When the user tries to open any such data file, the CSPro Designer will give an error message. However, the user can open the files in the context of running an application, whether that is running a batch program or exporting data. This setting only affects the CSPro Designer. It is not particularly robust protection, as a knowledgeable user can remove [NoEdit] from the file, but it will protect against most users modifying application files. With data entry applications, .pen files can be distributed for more protection against editing.

Temporary Data File

Most CSPro applications sensibly require data files to operate but there may be times when the contents of a data file are unimportant. Examples of this include a menu program that launches another program based on a user's selection. Such a program generally has no output and thus the contents of the file are meaningless. For batch programs it may be the case that you want to use CSPro functions to perform some task, such as creating content using the `write` statement.

Before CSPro 7.0, it was possible to write "<none>" in the Define File Associations dialog to have CSPro create a temporary data file. Now you can use a none data source.

Data entry applications run as they would with a file specified. Batch applications, however, do not run any logic associated with elements of the dictionary. Instead only the application `preproc` and `postproc` are executed.

```
PROC GLOBAL

// this function concatenates all the .dat files in a directory by order of date
function ConcatenateDatFiles()

    string file_listing_filename = "file_listing.txt";
    string output_filename = "Output/CombinedFiles.dat";

    // use the DOS dir command to create a file with the list of all .dat files
    string listing_call = maketext('cmd /c "dir /b /od *.dat > %s"', file_listing_filename);
    execsystem(listing_call, wait);

    // open that file and read all of the contents
    file file_listing_file;
    list string file_list;

    file_listing_file.open(file_listing_filename);
    file_listing_file.read(file_list);
    file_listing_file.close();

    filedelete(file_listing_filename);

    // create and run a pff to concatenate these files
    pff concat_pff;

    concat_pff.setproperty("AppType", "Concatenate");
    concat_pff.setproperty("InputData", file_list);
    concat_pff.setproperty("OutputData", output_filename);
    concat_pff.setproperty("Listing", "ConcatenateDatFiles.lst");
    concat_pff.setproperty("ViewListing", "OnError");
    concat_pff.setproperty("ViewResults", "No");

    concat_pff.exec();

end;

PROC CONCATENATE_FF

    ConcatenateDatFiles();
```

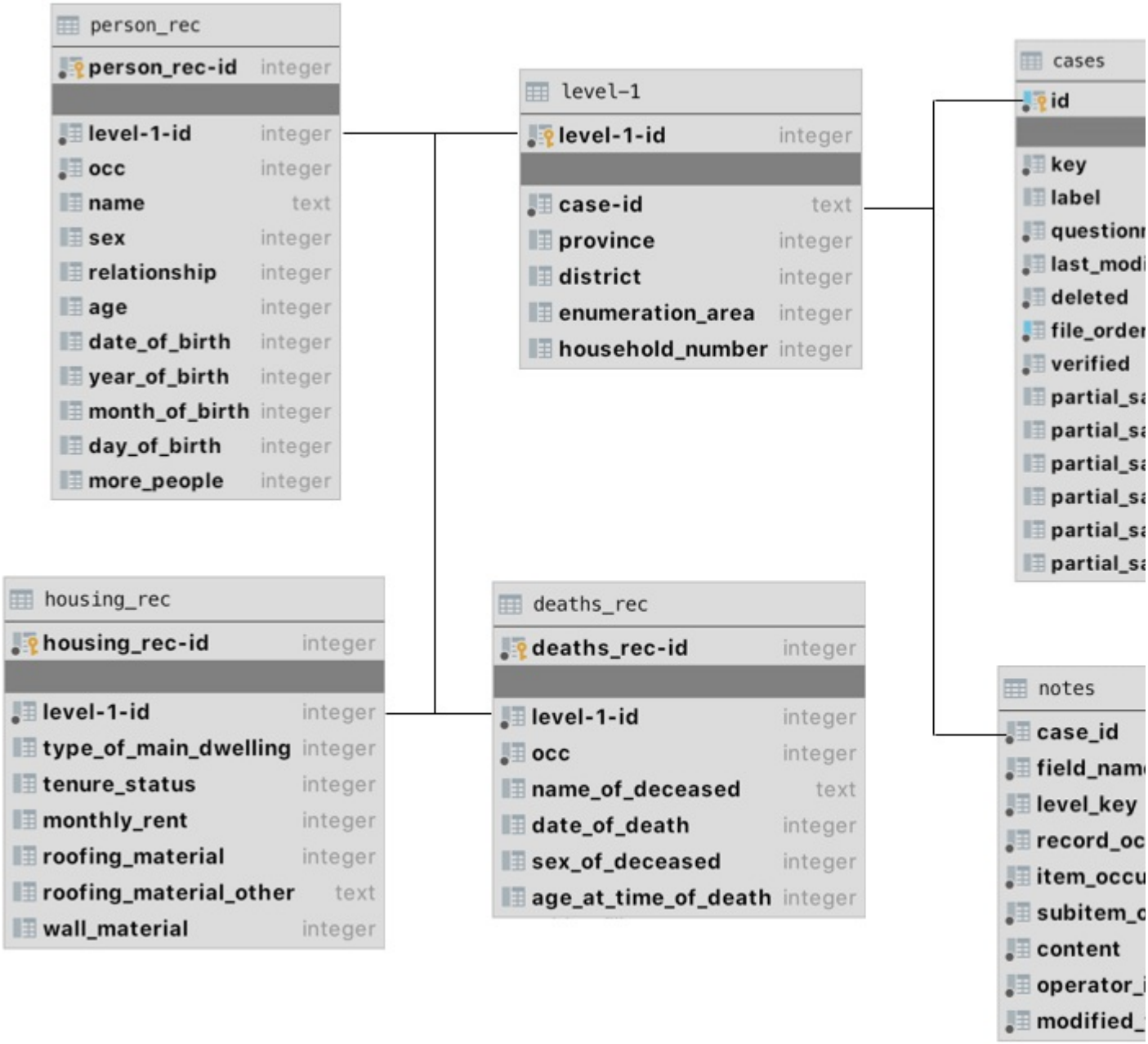
CSPro DB File Format

CSPro DB files are SQLite database files. SQLite is a file-based relational database which means that it is possible to use SQL commands to query the CSPro data. SQL queries can be run from a CSPro program using the `sqlquery` function. In addition, there are also many free viewers for SQLite that allow you to access the data outside of CSPro.

Note that this information applies only to CSPro DB files created in version 7.4 or later. Data files created in earlier versions store all the data from a case in a single database column so it is not possible to access individual items using SQL.

Structure of the Database

The structure of the database is based on the data dictionary. Each record in the data dictionary corresponds to a table in the database. The database tables have the same name as the record in the dictionary. Each item in the record corresponds to a column in the table with the same name as the dictionary item. The ID items from the dictionary are stored in a table named **level-1** (two level dictionaries will have a second table named **level-2** for the second level id-items). The tables for each record are linked to the table for the level by the primary key of the level table which is named **level 1-id**.



In the example above, the dictionary has three records: **PERSON_REC**, **HOUSING_REC** and **DEATHS_REC**. The database therefore has the corresponding records **person_rec**, **housing_rec** and **deaths_rec**. The dictionary ID items are in the table **level-1** which is linked to the record tables by the column **level-1-id**.

Each record table has an auto-increment integer primary key column named *<record name>-id*. This column does not correspond to any dictionary item and its value is set automatically by SQLite.

Records with multiple occurrences have an additional column named **occ** which represents the occurrence number. If a case in the data file has, for example, three occurrences of a particular record then the values of the **occ** column for the rows in the record table for that case will be 1, 2 and 3.

All CSPro DB files also have a table named **cases** that contains additional information about each case in the data file such as the case label, whether or not the case was deleted, and whether or not it was partially saved. The ID column of

the **cases** table is a UUID stored as a string. The database table for the first level in the dictionary is linked to the cases table by this ID column.

CSPro DB files also contain additional tables that store field notes as well as information required for data synchronization.

Querying the Database

To find the rows in the record tables that correspond to a particular case in the data file, join the record table with the level table using the linking column and filter on the ID items. For example, the following query on the example database above lists all the individuals in province 1, district 1, enumeration area 3, household 1:

```
SELECT * FROM person_rec
JOIN `level-1` ON `level-1`.`level-1-id` = person_rec.`level-1-id`
WHERE province = 1 AND district = 1 AND enumeration_area = 3 AND household_number = 1
```

Note the use of ` around **level-1-id**. This is required so that SQLite does not treat the "-" character in the name as a minus sign which would cause a syntax error.

The query above could potentially include data from deleted cases. To exclude deleted cases, join with the cases table and filter on the deleted column:

```
SELECT * FROM person_rec
JOIN `level-1` ON `level-1`.`level-1-id` = person_rec.`level-1-id`
JOIN cases ON cases.id = `level-1`.`case-id`
WHERE deleted = 0 AND province = 1 AND district = 1 AND enumeration_area = 3 AND household_numbe
r = 1
```

Using **GROUP BY**, the following query gets the number of households by enumeration area:

```
SELECT province, district, enumeration_area, COUNT(*) AS `number of households`
FROM `level-1`
JOIN cases ON cases.id = `level-1`.`case-id`
WHERE deleted = 0
GROUP BY province, district, enumeration_area
```

Modifying the Data

Modifying a CSPro DB file using SQL commands is strongly discouraged. Editing the schema or using update and insert queries to modify the data in a CSPro DB file can leave the file in a state where it can no longer be read by CSPro. In addition, any modifications to cases in the data file that do not correctly update the **file_revisions** and **vector_clock** tables will leave the file in a state where data synchronization will no longer work correctly. All modifications to the CSPro DB file should be done using CSPro data entry or batch edit applications or CSPro tools.

See also: SqlQuery Function, CSEntry Data Source File Types

Files Description

Data Dictionary File (.dcf)

Each data file manipulated by CSPro must be described by a data dictionary. The data dictionary file contains information defining the layout of a data file, including levels, records, items, value sets and values.

CSPro allows you to explicitly open, close and save data dictionary files independently of other application files. You must be careful when you do so if more than one application uses the data dictionary.

CSPro applications may optionally contain data dictionaries which represent secondary files, such as lookup files, which are opened during data entry.

The data dictionary file is a text file which may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment.

Data Entry Application File (.ent)

The data entry application file is the master file for data entry application. This file specifies all other files contained in the application, along with other information. CSPro allows you to open, close and save data entry application files. When you do so, all other files associated with the application are also opened, closed or saved.

The data entry application file is a text file which may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment. Advanced users might do so, however, to change the names of associated files from the CSPro assigned defaults.

Forms File (.fmf)

The forms file contains information about forms, their fields, text and rosters. The forms file also contains the name of the associated data dictionary file. Fields and rosters have links into the data dictionary. The flow during data entry, that is, the order in which forms and fields are entered, is defined in the forms file, not in the data dictionary.

CSPro allows you to explicitly open, close and save forms files independently of the application file. When you do so, the associated data dictionary file is also opened, closed or saved. Note that if you open a forms file you will not have access to its application's logic. Generally, only advanced users open forms files explicitly.

The forms file is a text file which may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment. Advanced users might do so, however, for example to change the name of the associated data dictionary file.

Logic File (.apc)

The logic file contains all the CSPro language statements which control the application. There is a logic file associated with each application. CSPro does not allow you to explicitly open the logic file. It is opened only when you open its associated application.

By default, the logic file has the same name as the application file, but with a different extension, **.apc**. This is not a requirement, however. Advanced users who change the name of this file must also remember to change the corresponding name in the application file.

You can associate multiple logic files with an application, which can be useful if you have a library of functions that you would like to share across several applications. To do this, select File -> Add Files and then choose the external logic

file to add to the application. The names of any variables or functions defined in that logic file cannot clash with the names used in the application's primary logic file. If you no longer want to use the external logic file, you can remove it by selecting File -> Drop Files.

The logic file is a text file which may be viewed with any text editor, such as Text Viewer or Notepad. While you may make changes to this file outside the CSPro environment, CSPro provides a powerful text editor which is integrated with the CSPro compiler.

Prior to CSPro 6.0, logic files had the extension .app. Although the default extension has changed, CSPro can still read applications that have logic files with the .app extension.

Message File (.mgf)

The message file for a data entry application stores the error message text that is displayed during data entry. The message file for a batch application stores the error message text that will be included in the execution report. The message file has the extension **.mgf**. While creating your application, you can view the message text in a window at the bottom of the logic screen.

You can associate multiple message files with an application, which can be useful if you have messages defined in multiple languages. To do this, select File -> Add Files and then choose the external message file to add to the application. If you no longer want to use the external message file, you can remove it by selecting File -> Drop Files.

If the name of an external message file begins with "CSProRuntime," then the messages in that file will override the CSPro system messages. See Collaborating on CSPro Development for more information about this file.

Basic Messages

Each line in the message file contains one message. A message consists of a message number followed by text. The message text can be up to 240 characters long. It is displayed on the entry screen when an `errmsg` function with the message number is executed in a data entry application. It is written to the execution report when `errmsg` function with the message number is executed in a batch application. The messages can also be accessed with other functions (including `maketext`).

For example, suppose a message file contains the following lines:

```
1 This is the first message
2 This is the second message
```

When an `errmsg(1)` function is executed in a data entry application, the message "This is the first message" is displayed on screen. When an `errmsg(2)` function is executed, the message "This is the second message" is displayed.

Messages with Arguments

Arguments can be specified in the `errmsg` function. These arguments can be numeric expressions or string expressions. String arguments in the error message text are indicated by **%s**. Integer numeric arguments are indicated with **%d**. Decimal numeric arguments are indicated with **%f**. For example, a message file might contain the following:

```
1 The month of %s has only %d days. You entered %d!
```

The application could use this as follows:

```
errmsg(1, "June", 30, 31);
```

When the function is executed, it knows to use error message 1 and substitute the word "June" for **%s** in the message text, the number 30 for the first **%d**, and the number 31 for the second **%d**. The message "The month of June has only 30 days. You entered 31!" will be displayed on the screen.

The more general the arguments of the message, the more flexible the message. In the example below, the value of the

variable HHDAY is used as a argument. The error message will use the value of HHDAY if the errmsg function is executed.

```
PROC HHDAY

    if HHMONTH = 6 and HHDAY > 30 then
        errmsg(1, "June", 30, HHDAY);
        reenter;
    endif;
```

Messages in Multiple Languages

When designing an application that will be used in multiple languages, you can define multiple messages for a given message number. Use the language name, followed by the message number in parentheses, as shown in this example:

```
101     Hello
ZH(101) 你好
FR(101) Bonjour
```

CSPro will automatically display the correct message based on the current language. If no translated message exists for the current language, the default message is shown.

Using the Language directive declares that all subsequent error messages are in a given language. For example, the previous example could be rewritten as:

```
Language=EN
101 Hello

Language=ZH
101 你好

Language=FR
101 Bonjour
```

If you would like to define messages for multiple languages at the same time, separate the name of each language with a comma:

```
Language=HI, PA
101 नमस्ते
```

You can also use the message file to translate string literals that are used along with the tr function. For example:

```
ZH("Hello") 你好
FR("Hello") Bonjour
```

Calling tr("Hello") in logic will display the correct string based on the current language.

See also: Multiple Language Applications, Tr Function

Question File (.qsf)

The question file contains information related to **CAPI** (**C**omputer-**A**ssisted **P**ersonal **I**nterviewing) data entry applications. Such information includes question text to appear on the screen with each field and help screens to appear when the operator presses the help (**F2**) key on desktop or the **?** icon on mobile.

File Format

Most users do not need to understand the internal format of the .qsf file, but for advanced users who wish to use their own tools to read or generate question files the following section describes the file format.

You should avoid editing the question text file by hand, since small changes to the content or indentation can easily render the file unreadable by CSPro.

The .qsf file is in YAML format with UTF-8 encoding.

At the top-level the .qsf contains a mapping (associative array) with the following keys:

- **fileType**: always set to "Question Text"
- **version**: version of CSPro that created the question text file; for example "CSPro 7.7"
- **languages**: list of languages used in the question text file
- **styles**: list of styles used in the question text file
- **questions**: list of questions, with one entry for each field or block in the data entry application that contains non-blank question or help text

Languages

Each language is a mapping with two keys, **name** and **label**, which contain the name and label defined in the languages dialog available from the **CAPI** menu.

```
languages:
 - name: EN
  label: English
 - name: FR
  label: French
```

Styles

The list of styles is used to populate the styles dropdown in the question text editor which can be customized using the **Edit Styles** dialog available from the **CAPI** menu.

Styles are implemented using CSS (Cascading Style Sheets) classes. Since the question text itself is in HTML format, styles are applied to the question text using CSS classes just as one would in a standard HTML web page. Each style in the list of styles in the question text file is implemented using a CSS class that describes the style using standard CSS style rules.

Each style in the question text file is a mapping with three keys:

- **name**: name displayed in the style dropdown and in the Edit Styles dialog
- **className**: CSS class name used for the style in the HTML question text
- **css**: CSS style definition

```
styles:
 - name: Normal
  className: normal
  css: |
   font-family: Arial;font-size: 16px;
 - name: Instruction
  className: instruction
  css: |
   font-family: Arial;font-size: 14px;color: #0000FF;
```

If you were defining the above classes in a CSS file rather in CSPro you would have:

```
.normal { font-family: Arial;font-size: 16px; }
.instruction { font-family: Arial;font-size: 14px;color: #0000FF; }
```

If in the question text editor in CSPro you set the style for a block of text, the editor will modify the HTML to apply the

selected style; e.g.:

```
<p>A question with <span class="instruction">a style applied to this part</span>.</p>
```

Any HTML that does not have a class specified will have the normal style applied. This is done by applying the normal class to the body element of the HTML document.

Note: Prior to version 7.6, the format of the question text file was Rich Text Format (RTF). Opening an older data entry application in version 7.6 onwards will automatically convert the QSF file from RTF to HTML—though you must edit the QSF file to force the conversion. The original RTF will be retained as xxx.qsf.backup. Data entry applications with question text created or edited in version 7.6 can no longer be opened in older versions of CSPro.

Questions

The bulk of the question text is made up of a list of questions where each question is a mapping with a name and a list of conditions.

The name is the fully qualified name of the dictionary item that the question text is displayed for. This is name of the dictionary and the name of the item separated by period; e.g., **SIMPLECAPI_DICT.AGE**. For question text that is applied to blocks, the name is simply the name of the block with no dictionary name or period.

The conditions correspond to the entries in the conditions grid that appears underneath the question text editor. Even questions that have no conditions defined in the conditions grid still contain a condition in the .qsf file, although in such cases the condition logic is empty. Each condition has three keys: **logic**, **questionText**, and **helpText**. The logic is the CSPro expression from the conditions grid. The questionText and helpText are mappings from language code (the codes from the languages section described above) to the HTML for the question or help text.

```
questions:
  - name: SIMPLECAPI_DICT.RELATIONSHIP
    conditions:
      - logic: headKnown = 0
        questionText:
          EN: |
            <p><span class="name">~~PersonName()~~:</span> Are you the head of household
          FR: |
            <p><span class="name">~~PersonName()~~:</span> Êtes-vous le chef de ménage? S

        helpText:
          EN: |
          FR: |

      - logic: headKnown = 1
        questionText:
          EN: |
            <p><span class="name">~~PersonName()~~:</span> What is your relationship to
          FR: |
            <p><span class="name">~~PersonName()~~:</span> Quel est votre lien de parenté

        helpText:
          EN: |
          FR: |
```

See also: HTML Question Text

Application Properties File (.csprops)

A data entry, batch edit, or tabulation application can contain a file that stores application properties. This optional file has the extension **.csprops**. The file will be associated with an application when a user changes properties away from their default settings. With these property settings externalized in a file, it is possible to share properties across multiple applications.

See also: Application Properties

Program Information File (.pff)

Program Information Files (.pff) are used to run CSPro applications or tools in production mode. A PFF file stores the name of the application or tool, the data file(s) to be used, and any runtime parameters specific to the program. You can use a PFF file as a command line parameter for most CSPro applications and tools. PFF files can be edited with a text editor, with the PFF Editor, or can be manipulated in logic with the pff object.

Application-specific PFF information can be viewed in the following help topics:

- Batch Edits
- Compares
- Concatenates
- Deployments
- Data Entry
- Excel to CSPro Conversions
- Exports
- Frequencies
- Indexing
- Packs
- Paradata Concatenates
- Reformats
- Sorts
- Synchronizations
- Tabulations

See also: Pff Object

Binary Data Entry Application File (.pen)

- A binary data entry application file is an alternative to distributing the entire suite of the data entry application files. It is a single binary file that contains all the information from your data entry application's dictionaries, forms, logic, message file, and CAPI question text.
- While this file is generated within the CSPro Designer, it cannot be read by the CSPro Designer. It is read by CSEntry, on either desktop or mobile devices. A .pen file cannot be viewed with a text editor.
- Binary data entry application files provide security during a data entry operation. If you install a .pen file to your data entry operators' workstations, the operators cannot change the application, even if they have the full CSPro installed on their machines.
- **.pen** files are the only type of file you can install on an Android device.
- Binary data entry applications created in versions of CSPro prior to version 6.0 used the extension .enc, not .pen. These files cannot be read by CSEntry versions 6.0 or higher.

Index File (.csidx)

The index file stores the case IDs and their physical location in the data file. This helps CSPro applications manipulate the data file more quickly. The file is a binary file and should not be modified file outside of CSPro.

CSPro automatically generates an index file when creating a new data file. CSPro then coordinates between both files as cases are added, modified, or deleted.

When CSPro opens a data file, it looks for the corresponding index file. If it does not find one, CSPro automatically generates a new one. If you suspect, for any reason, that the index file has been corrupted, or does not match the data file, simply delete the index file and let CSPro generate a new one.

An index is only created for text files. Prior to CSPro 7.0, the data file index had the extension .idx.

Notes File (.csnot)

The notes file is a text file that contains all notes processed by a data entry application. The file contains the case IDs, the name of the data item to which each note belongs, the name of the operator, the time the note was left, and the note text. There may be multiple note records for a particular field.

The format of each note record is as follows:

- The first N characters are the case IDs.
- The next 32 characters are the field name as given in the dictionary (left justified). If you use field names longer than 32 characters, the field name will be encoded in a way so that it fits in 32 characters. You can use the `NOTES_FIELD_VS` value set of the notes dictionary to figure out which note is which.
- The next 32 characters are the name of the operator ID (left justified).
- The next 8 characters are the date the note was last modified (YYYYMMDD).
- The next 6 characters are the time the note was last modified (HHMMSS).
- The next 5 characters are the record occurrence number if applicable (right justified).
- The next 5 characters are the item occurrence number if applicable (right justified).
- The next 5 characters are the subitem occurrence number if applicable (right justified).
- The remaining characters are the note text.

Note text can contain "\n" characters, which indicate a newline (Enter key) character. If the note text is automatically wrapped in the note text box, there will be no newline characters.

The .csnot file can be processed by another CSPro application by creating a data dictionary for it. You can use the Dictionary Macros function to automatically create this dictionary.

Notes can be created, edited, and deleted using the `getnote`, `putnote`, and `editnote` functions.

Notes for non-text files (such as CSPro DB files) are stored within the file itself and not in a .csnot file. Prior to CSPro 7.0, the notes file had the extension .not.

Status File (.sts)

The status file stores information about which cases are partially entered and the last field entered. It also keeps track of the cases that have been verified. CSEntry uses this information to:

- Show which cases in the data file are partially entered.
- Restart partially entered cases at the last field location entered.
- Show the number and percent of verified cases in the operator statistics.

The .sts file is a text file and you can view this file in Text Viewer.

If necessary, you can modify this file in a text editor. However, you should manipulate this file with extreme caution. If

you modify it incorrectly, you can lose information about partially entered cases or which cases have been verified.

Status information for non-text files (such as CSPro DB files) is stored within the file itself and not in a .sts file.

Paradata Log File (.cslog)

Paradata log files contain information about paradata events stored during an application's run. Files created using this type have the extension **.cslog** and are not text files so they cannot be viewed in Text Viewer.

You can use the Paradata Viewer to view the contents of the file. Advanced users can also query paradata logs by using the `paradata` logic function or by generating a templated report. Paradata can be synchronized between devices or servers by using the `syncparadata` function.

You can combine multiple paradata logs into a single log by using the Paradata Concatenator tool.

Operator Statistics File (.log)

The log file stores operator statistics generated by CSEntry for a corresponding data file. CSEntry creates a log file when it creates a new data file. The log file is a comma-delimited text file with a fixed format. It is designed to be easily imported into other software packages for custom processing. The log file can also be processed by CSPro application with the appropriate data dictionary.

Each record in the log file represents one data entry session. The record layout is as follows:

Position	Contents
1 - 3	Mode (**ADD**, **MOD**, or **VER**)
4	*comma*
5 - 36	Operator ID (as entered)
37	*comma*
38 - 47	Start date (mm/dd/yyyy)
48	*comma*
49 - 56	Start time (hh:mm:ss)
57	*comma*
58 - 65	End time (hh:mm:ss)
66	*comma*
67 - 74	Total time (End time - Start time) (seconds)
75	*comma*
76 - 83	Pause time (seconds)
84	*comma*
85 - 92	Number of cases written
93	*comma*
94 - 101	Number of records written
102	*comma*
103 - 110	Number of keystrokes
111	*comma*
112 - 119	Number of bad keystrokes
120	*comma*
121 - 128	Number of fields with errors attributed to keyer
129	*comma*
130 - 137	Number of fields with errors attributed to verifyer
138	*comma*

139 - 146	Total number of fields verified

Listing File

The listing file contains information about how an application was run and contains the results of the program's operations. It gives the following information:

- The names of the files used.
- The start and end time of the run.
- Any error messages, both user and system messages, which occurred during the run.

The listing file associated with a batch edit application is often the primary output of the application. For most other applications and tools, the listing file is a secondary output. For data entry applications, this file can be useful for monitoring the work of keyers in training and identifying problem keyers.

Listing File Formats

The traditional CSPro listing file is a text file with the extension **.lst**. However, you may find it useful to change the listing file output based on the kind of analysis you are conducting. There are five listing file formats. CSPro will output to a format based on the extension of the listing file name.

Format	Extension	Results Viewer
Text	.lst	Text Viewer
HTML	.html / .htm	Default Internet browser
Excel	.xlsx	Microsoft Excel (or other application associated with Excel files)
CSV	.csv	Application associated with.csv files (often Microsoft Excel)
CSPro Data	.csdb / .dat	Data Viewer

Using any extension not in the table above will result in a text listing file.

The Excel, CSV (comma-separated values), and CSPro Data listing files can only be used with applications that contain an input dictionary so they are not suitable for some tabulation applications.

Whereas the text and HTML listing files display keys of cases that had errors in text format, the Excel, CSV, and CSPro Data listing files output these keys as separate columns/items (e.g., "0901" vs. PROVINCE: 9 / DISTRICT: 1).

Error messages that occur outside of a case (for example during the application procedure) are written to all listing file formats except for CSPro Data, which only contains information about case-related errors. When using the CSPro Data listing file, a dictionary that describes the data file will be output using the name of the listing file with the added extension .dcf.

A data entry application's listing file is appended to any existing listing file, whereas listing files for batch and tabulation applications are created from scratch for the program's run. Excel listing files do not support append mode so they may not be a suitable format for data entry applications.

Frequencies File

The frequencies file contains the frequency tables generated by using the `freq` statement or when calling the `freq.save` function. When using frequencies in an application, you can specify the name of the file in the File Associations dialog. You can also manually specify the name using the Freqs attribute in a PFF file.

If no frequencies file is specified, then frequency tables will be saved to the listing file if possible. Only text and HTML listing files support the saving of frequencies.

Frequency File Formats

CSPro can write frequency tables in four formats. CSPro will save tables in a format based on the extension of the frequencies filename.

Format	Extension	Results Viewer
Table	.tbw	Table Viewer
Text	.lst	Text Viewer
HTML	.html / .htm	Default Internet browser
Excel	.xlsx	Microsoft Excel (or other application associated with Excel files)

Using any extension not in the table above will result in a text frequencies file. When writing text frequencies, you can use the ListingWidth PFF attribute to modify the width of the tables.

Frequency Table

A frequency table contains between five and eight columns. The order of the columns depends on the file format.

- **Value**: The value (code) of the row.
- **Label**: The label of the row, taken from an item's value set.
- **Frequency**: The number of times this row's value was tallied.
- **Cumulative Frequency**: The number of times this row's value was tallied added with the previous rows' tallies.
- **Frequency Percent**: The frequency divided by the total of all tallied values.
- **Cumulative Frequency Percent**: The cumulative frequency divided by the total of all tallied values.
- **Frequency Net Percent**: The frequency divided by the total of all non-blank tallied values.
- **Cumulative Frequency Net Percent**: The cumulative frequency divided by the total of all non-blank tallied values.

Based on the frequency generation settings, sometimes the value and label columns will be combined. The net percent columns will be suppressed If no non-blank values are tallied or if the `nonetpercents` command is used.

Batch Edit Application File (.bch)

The batch edit application file is the master file for a batch edit application. This file specifies all other files contained in the application, along with other information. CSPro allows you to open, close and save batch edit application files. When you do so, all other files associated with the application are also opened, closed or saved.

The batch edit application file is a text file that may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment. Advanced users might do so, however, to change the names of associated files from the CSPro assigned defaults.

Edit Order File (.ord)

The Edit Order File is a text file that contains information about the sequence in which fields defined in the associated data dictionary are edited during batch editing. Each item in the associated CSPro dictionary is listed in the Edit Order File in the sequence in which the procedures for that item will be executed.

Saved Arrays File (.sva)

The saved arrays file stores values from arrays in an application marked with the `save` keyword. At the end of program execution, the current values of any saved arrays are written to the file. When the program is run, the saved array file is

read and the array is initialized with the values from the saved array file.

CSPro creates the saved arrays file automatically when one or more arrays in an application are marked as a saved array. The name of the saved arrays file is generally the same as that of the application with the additional .sva file extension added (e.g., application_name.bch.sva), but it can be modified using the PFF SaveArray attribute.

The saved arrays file is a text file that may be viewed with any text editor, such as Text Viewer or Notepad, or using the Save Array Viewer. It is not recommended to make changes to this file outside the CSPro environment. Advanced users might do so, however, to set initial values for arrays.

If you want to prevent the file from being updated after a run, which may be desirable if you want to produce the same results from run to run, you can disable the update via logic:

```
setproperty("UpdateSaveArrayFile", "No");
```

Tabulation Application File (.xtb)

The tabulation application file is the master file for a tabulation application. This file specifies all other files contained in the application, along with other information.

CSPro allows you to open, close and save tabulation application files. When you do so, all other files associated with the application are also opened, closed or saved.

The tabulation application file is a text file which may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment. Advanced users might do so, however, to change the names of associated files from the CSPro assigned defaults.

Table Specifications File (.xts)

The tabulation specifications file contains tables, dictionary items/value sets and other information which defines a set of tables. The file also contains the name of the associated data dictionary file. Items and value sets have links into the data dictionary.

The tabulation specifications file is a text file that may be viewed with any text editor, such as Text Viewer or Notepad. It is not recommended to make changes to this file outside the CSPro environment.

Tables File (.tbw)

The tables file (.tbw) is a text file that contains information about a table layout such as stubs, the headers, column size, etc. It also contains the table data. This file is read by the Table Viewer to produce a "published" table.

See also: Save Tables for the Table Viewer

Area Names File (.anm)

The Area Names file (.anm) is a text file that you can create using any text editor. Be sure that you save this file with the extension .anm.

The Area Names file defines the hierarchical levels of geography and assigns text names to the numeric codes for each geographic unit. The items must be defined in the common part of the data dictionary and should be listed in order from major to minor division.

See also: Create an Area Names File

Table Matrices File (.tab)

A table matrices file (.tab) contains the output of tabulated numbers from CSPro's tabulation engine. They are row and column contents of the tabulated tables, without any surrounding text.

A table matrices file may also contain the output of the Consolidate process of CSPro, where table matrices for one geographic level are added together to produce table matrices for a higher geographic level.

These files are used as output and input when tabulations are run in parts.

See also: Table Matrices Index File (.tabidx), Introduction to Run in Parts

Table Matrices Index File (.tabidx)

A table matrices index file (.tabidx) contains indices for a table matrices file with the same name. While it is not necessary to save these files, because they can be recreated, having them around makes tabulation processing go faster.

See also: Table Matrices File (.tab), Introduction to Run in Parts

Frequency Specification File (.fqf)

Tabulate Frequencies is a CSPro tool that allows you to display a frequency distribution of a dictionary variable(s) based on its value set(s). For example, you could have defined several value sets such as "age in 5 year groups," "level of education," or "type of occupation." Tabulate Frequencies gives you both the numeric count and percentage distribution for the selected variables. It also gives the cumulative distributions.