

PsyScope

Version 1.0

User Manual

Credits

Design	Jonathan D. Cohen Jefferson Provost	Brian MacWhinney Matthew Flatt
Development	Jefferson Provost Matthew Cushman Erik Selberg Eric Sedlar Jay Gowdy	Matthew Flatt Jonathan D. Cohen Robert Findler Darius Clynes
User Documentation	Matthew Flatt Jefferson Provost	Jonathan D. Cohen
Editing and Typesetting	Matthew Flatt	
Button Box Hardware	Bruce Taylor	
Contributors	Jonathan Cohen Jay McClelland Mark Seidenberg Mark Johnson Cathy Harris Kay Bock Larry Barsalou	Brian MacWhinney Elizabeth Bates Maggie Bruck Jonathan Vaughan Kevin Miller Tom Trabasso
Testing & Feedback	Vijoy Abraham Randy Bruno Andy Edmonds Dan Gallagher Charles Hill Leigh Nystrom Steve Ritter Norman Vinson	Kathryn Brennan Robert Buffington Julia Evans Jolene Gordon Therese Huston Alan Petersen Michael Shapiro

PsyScope was written at Carnegie Mellon University, Department of Psychology.

© 1994 Carnegie Mellon University.

PsyScope is free for non-commercial use. All rights reserved.

The PsyScope application and this manual may be obtained from <http://psyscope.psy.cmu.edu>. Send all bugs and comments to PsyBug@psyscope.psy.cmu.edu. Correspondence regarding the PsyScope Consortium should be sent to Consortium@psyscope.psy.cmu.edu or to Jonathan Cohen or Brian MacWhinney at Department of Psychology, Carnegie Mellon University, Pittsburgh, PA 15123, USA.

This manual was produced with FrameMaker[®] publishing software for the Apple[®] Macintosh[™] using the Helvetica, Times, Courier, Chicago, and Symbol typeface families.

Overview of the PsyScope Manual

- 0.1 Organization 1
- 0.2 System Requirements 1
- 0.3 Conventions 2

Part 1: Introduction to PsyScope

Chapter 1. Introduction 1

Chapter 2. Running Your First Experiment 3

- 2.1 Open the Experiment 3
- 2.2 Run the Experiment 4
- 2.3 Look at the Experiment in the Design Window 4
- 2.4 Change One or Two Things in the Experiment 5

Chapter 3. Designing an Experiment 9

- 3.1 A Word About Scripts and the Graphic Environment 9
- 3.2 Creating a New Experiment 10
 - 3.2.1 Using the Design Window 11
 - 3.2.2 Creating a Trial Template 12
 - 3.2.3 Creating a New Event 14
 - 3.2.4 Setting Attributes Using the Event Dialog 16
 - 3.2.5 Timing and Sequencing Events 19
 - 3.2.5.1 Timing an Event 19
 - 3.2.5.2 Sequencing Events 22
 - 3.2.6 Recording Responses 26
 - 3.2.7 Conditions and Actions 30
 - 3.2.8 Using Factor Tables and Lists 36
 - 3.2.8.1 Creating a Factor Table 36
 - 3.2.8.2 Creating Trials from a Factor Table 41
 - 3.2.8.3 Controlling How Cells are Chosen 49
 - 3.2.8.4 Lists 59
 - 3.2.8.5 Counterbalancing 63
 - 3.2.9 Using Blocks 65
 - 3.2.9.1 Blocking Trials 66
 - 3.2.9.2 Varying by Block 72
 - 3.2.10 Using Subject Info and Groups 74
 - 3.2.10.1 The Subject Info Dialog 75
 - 3.2.10.2 Groups 81
 - 3.2.11 Experiment Attributes 90
 - 3.2.11.1 Instructions and Debriefing Files 91
 - 3.2.11.2 Rest Periods 92
 - 3.2.11.3 Reverse Video 92
 - 3.2.12 Running Trials 92
 - 3.2.12.1 Running the Experiment and Breaking 92

- 3.2.12.2 Practice vs. Run Mode 93
- 3.2.12.3 Previewing Trials and Using the Trial Chooser floating window 94
- 3.2.12.4 The Trial Monitor 95
- 3.3 Where to Go from Here 99

Part 2: Graphic Environment Reference

Chapter 4. Introduction 103

- 4.1 A Word About Scripts 103

Chapter 5. Windows and Dialogs 105

- 5.1 Windows vs. Dialogs 105
 - 5.1.1 Windows 105
 - 5.1.2 Dialogs 106
- 5.2 The Design Window 107
 - 5.2.1 Objects and the Experiment Hierarchy 107
 - 5.2.1.1 Linking Objects 108
 - 5.2.2 Design Window Palettes 109
 - 5.2.2.1 The Tools Palette 110
 - 5.2.2.2 The Events Palette 112
 - 5.2.3 Design Window Work Area 113
 - 5.2.3.1 Cleaning Up 113
 - 5.2.3.2 Trash 113
 - 5.2.4 Design Window Control Area 113
 - 5.2.5 New Object Name Dialog 114
 - 5.2.6 The Palettes Window 114
 - 5.2.7 Object List Dialog 115
 - 5.2.8 Get Object Dialog 115
 - 5.2.9 View Trash Dialog 115
- 5.3 The Experiment Object 116
 - 5.3.1 Connecting Objects to the Experiment 116
 - 5.3.2 Experiment Dialog 116
- 5.4 Groups 118
 - 5.4.1 Connecting Objects to a Group 118
 - 5.4.2 Group dialog 119
- 5.5 Blocks 120
 - 5.5.1 Connecting Objects to Blocks 120
 - 5.5.1.1 Connecting Events to a Block 121
 - 5.5.1.2 Connecting Blocks to a Block 121
 - 5.5.1.3 Connecting Lists to Blocks 121
 - 5.5.2 Block Dialog 121
 - 5.5.3 Superblock Dialog 123
- 5.6 Trials and Templates 124
 - 5.6.1 The Trial Template Window 124
 - 5.6.1.1 Template Name and Buttons 125
 - 5.6.1.2 Palettes 125

5.6.1.3 Event Name Area	126
5.6.1.4 Timeline Area	126
5.6.1.5 Event Status Area	128
5.7 Factors and Lists	129
5.7.1 Definitions	129
5.7.1.1 Factor Table Sets	130
5.7.1.2 Lists	130
5.7.1.3 List Files	131
5.7.1.4 Expanded Lists	132
5.7.1.5 Nested Factors	132
5.7.1.6 Lists in a Factor table	132
5.7.1.7 Level Order and Crossing Types	132
5.7.1.8 Factors in the Hierarchy	135
5.7.2 Factor Table Windows	138
5.7.2.1 The Factor Table Window	138
5.7.2.2 Table Info Dialog	140
5.7.2.3 New/Rename Table Factor Dialog	142
5.7.2.4 New/Rename Table Level Dialog	142
5.7.2.5 Latin Squares Dialog	142
5.7.2.6 Choose Crossing Dialog	143
5.7.2.7 Factor Table Floating Window	144
5.7.3 List Dialogs	145
5.7.3.1 List Dialog	145
5.7.3.2 List File Dialog	146
5.7.3.3 Factor Set Dialog	147
5.7.3.4 Level Dialog	147
5.7.3.5 Connect List Dialog	148
5.8 Attributes	149
5.8.1 Definitions	149
5.8.1.1 Attribute Inheritance	150
5.8.1.2 Factor Table and Attributes	150
5.8.1.3 Attribute Dialogs	150
5.8.2 The Standard Attributes Dialog	151
5.8.2.1 Settings	152
5.8.2.2 Custom Attribute Sets	156
5.8.2.3 Special Keyboard Shortcuts	157
5.8.2.4 New/Rename/Retype Attribute Dialog	157
5.8.3 Experiment Attributes	158
5.8.3.1 Custom Run and Custom Practice Attributes	158
5.8.3.2 Default Stimulus/Event/Trial Attributes	159
5.8.3.3 Experiment Attributes Dialog	160
5.8.3.4 Standard Experiment Attributes	161
5.8.4 Group Attributes	169
5.8.4.1 Custom Group Attributes	169
5.8.4.2 Default Stimulus/Event/Trial Attributes	169
5.8.4.3 Group Attributes Dialog	170
5.8.5 Block Attributes	170
5.8.5.1 Custom Block Attributes	171
5.8.5.2 Default Stimulus/Event/Trial Attributes	171

- 5.8.5.3 Block Attributes Dialog 172
- 5.8.6 Trial Attributes 172
 - 5.8.6.1 Custom Template Attributes 173
 - 5.8.6.2 Default Stimulus/Event Attributes 173
 - 5.8.6.3 Trial Attributes Dialog 174
 - 5.8.6.4 Standard Trial Attributes 175
- 5.8.7 Event Attributes 176
 - 5.8.7.1 Event Types 177
 - 5.8.7.2 Event Attributes 179
 - 5.8.7.3 Stimulus Attributes 181
- 5.8.8 Stimulus Attribute Dialogs 185
 - 5.8.8.1 Stimulus Dialog 185
 - 5.8.8.2 Stimuli Dialog 185
 - 5.8.8.3 Style Dialog 185
 - 5.8.8.4 Ports and Positions Dialogs 187
- 5.9 Conditions and Actions 192
 - 5.9.1 Conditions and Actions Dialog 193
 - 5.9.2 Conditions Dialog 194
 - 5.9.3 Condition Parameter Dialogs 195
 - 5.9.3.1 Button Box Parameter Dialog 195
 - 5.9.3.2 Key Parameter Dialog 196
 - 5.9.3.3 Mouse Parameter Dialog 197
 - 5.9.3.4 Start/End Parameter Dialog 197
 - 5.9.3.5 When Parameter Dialog 198
 - 5.9.3.6 ScriptWhen Parameter Dialog 198
 - 5.9.4 Actions List Dialog 198
 - 5.9.4.1 Available Actions 198
 - 5.9.5 Parameters dialogs 205
- 5.10 Trial Manager Variables 205
 - 5.10.1 How Trial Manager Variables Work 206
 - 5.10.2 Trial Manager Variable Expressions 206
 - 5.10.3 Built-in Variables 207
 - 5.10.4 Linking to Variable Values 207
 - 5.10.5 Trial Manager Variables Dialog 208
- 5.11 Trial Chooser Floating Window 209
- 5.12 Additional Concepts 210
 - 5.12.1 List Ordering 210
 - 5.12.1.1 Weights 210
 - 5.12.1.2 Other Modifiers 210
 - 5.12.2 Trial Counting 211
 - 5.12.2.1 Experiments without Blocks 211
 - 5.12.2.2 Experiment with Blocks 211
 - 5.12.2.3 Block Scaling 211
 - 5.12.2.4 Superblocks 212
 - 5.12.2.5 Trial Counts and Crossing Factors 212
 - 5.12.2.6 Trial Counts Reported in the Trial Monitor 212

Chapter 6. Running and Managing Experiments 213

6.1 File System	213
6.1.1 Using Projects	213
6.1.1.1 Creating a Project	213
6.1.1.2 The Scripts Dialog	214
6.1.2 Path Names	215
6.1.2.1 Relative Paths	215
6.1.2.2 Reverse Notation	215
6.1.3 Resources	216
6.1.3.1 PsyScope Extensions	216
6.1.4 The Data File	217
6.1.4.1 Specifying the Data File	217
6.1.4.2 Information in the Data File	217
6.1.4.3 Formatting the Data File	222
6.1.5 The Log File	222
6.1.5.1 Specifying the Log File	222
6.1.5.2 Viewing and Editing the Log File	223
6.1.5.3 Information in the Log File	223
6.1.6 Safe Saves	224
6.1.7 Start-up Shortcuts	224
6.2 Subject Info	224
6.2.1 Subject Info Items	225
6.2.1.1 Special Items	225
6.2.1.2 Subject Info Dialog	226
6.2.2 Subject Info and the Log File	230
6.2.3 Subject Number Calculation	230
6.2.3.1 Logging and Scheduling Correctly	231
6.2.3.2 How the Subject Number Calculations are Performed	232
6.2.4 Automatic Grouping	233
6.2.4.1 Automatic Grouping Dialog	234
6.2.4.2 Group Criteria Dialog	235
6.2.5 Data File Dialog	236
6.2.6 Subject Info Schedule Dialog	237
6.3 The Trial Monitor	238
6.3.1 Trial Compilation Statistics	241
6.4 The Event Monitor and Variable Monitor	241
6.4.1 The Event Monitor	242
6.4.1.1 Trial Information	242
6.4.1.2 Event Information	243
6.4.1.3 Action Information	243
6.4.2 Event Monitor Operation	244
6.4.2.1 Perceived Times	244
6.4.2.2 Step Mode	245
6.4.3 The Variable Monitor	245
6.5 Space and Speed	246
6.5.1 Precompiling	246
6.5.1.1 Problems with Precompiling	247
6.5.2 Loading Stimuli	247
6.5.2.1 Load Time	247

Chapter 7. User Environment 253

- 7.1 Menus Reference 253
 - 7.1.1 File Menu 253
 - 7.1.2 Edit Menu 254
 - 7.1.3 Run Menu 254
 - 7.1.4 Utilities Menu 254
 - 7.1.5 Design Menu 256
 - 7.1.6 Script-Specific Menus 256
 - 7.1.7 Windows Menu 256
- 7.2 The Console 257
- 7.3 The Editor 258
 - 7.3.1 Editor Menu Items 258
 - 7.3.2 Keyboard Commands 260
 - 7.3.3 Action Bar 261
 - 7.3.4 Interactive Mode 261
 - 7.3.5 The Find Dialog 262
- 7.4 The Evaluator 263
- 7.5 The Help System 264
 - 7.5.1 The Help Search Dialog 264
 - 7.5.2 Help Action bar Buttons 265
- 7.6 Options 265
 - 7.6.1 General Options 266
 - 7.6.2 Run Options 266
 - 7.6.3 Editor Options 267
 - 7.6.4 Design Options 268
 - 7.6.5 Display Options 269
 - 7.6.6 Custom Options 269

Part 3: Scripting User Manual

Chapter 8. Introduction 273

Chapter 9. Scripting Overview 275

- 9.1 A PsyScope Script 275
- 9.2 Entries 275
 - 9.2.1 Entry Name 276
 - 9.2.2 Content 276
 - 9.2.3 Attribute Blocks 276
 - 9.2.4 How Entries Are Used 277
 - 9.2.5 Spaces, Blanks, and Quotes 277
- 9.3 Comments 278
- 9.4 Entry References 279
- 9.5 Lists 279

9.6 Function Calls 280

Chapter 10. Scripting an Experiment 281

- 10.1 Scripting a New Experiment 281
 - 10.1.1 The Standard Script Template 281
 - 10.1.2 Using the Interactive Editor 284
 - 10.1.3 Scripting a New Event 285
 - 10.1.3.1 Timing and Sequencing Events 287
 - 10.1.3.2 The 'Duration' Attribute 287
 - 10.1.3.3 The 'StartRef' Attribute 289
 - 10.1.4 Scripting Conditions and Actions 290
 - 10.1.5 Scripting Templates 292
 - 10.1.5.1 Scripting Trial Actions 295
 - 10.1.5.2 Attribute Inheritance 296
 - 10.1.5.3 TrialAttrib() 297
 - 10.1.6 Scripting Experiment Attributes 299
- 10.2 Scripting Factors 299
 - 10.2.1 Scripting the Acuity Experiment 299
 - 10.2.2 Scripting Free Factors 301
 - 10.2.2.1 Compact Factors 303
 - 10.2.2.2 Factor Interactions 304
 - 10.2.2.3 Factor Sets 306
 - 10.2.2.4 Nested Factors 307
 - 10.2.3 Scripting Factor Tables 309
- 10.3 Scripting Blocks and Groups 309
 - 10.3.1 Scripting Blocks and BlockAttrib() 309
 - 10.3.2 Scripting Groups 311
 - 10.3.2.1 GroupAttrib() 312
- 10.4 Advanced Topics 312
 - 10.4.1 Linking to the PsyScope Environment 312

Part 4: Scripting Reference

Chapter 11. Introduction 317

Chapter 12. PsyScript Reference 319

- 12.1 Components of a Script 319
- 12.2 Entries 320
 - 12.2.1 Entry Content and Expressions 320
 - 12.2.2 Attributes 321
 - 12.2.3 Entry Syntax 321
 - 12.2.4 References 323
 - 12.2.4.1 THIS and OWNER 324
- 12.3 Comments 324
- 12.4 Modifiers 324
 - 12.4.1 #PsyScope 324

12.4.2 #include and #winclude	325
12.4.3 #inherit and #noinherit	325
12.4.4 #NoIncludeStdLib	326
12.5 Section Markers	326
12.6 Operators and Functions	326
12.6.1 Literals	326
12.6.2 Function Calls	327
12.6.2.1 Exceptions to the Rules	327
12.6.3 Operation Sentences	328
12.6.3.1 Distributivity	328
12.7 Attribute Block Reference	329
12.8 Lists	329
12.8.1 Accessing a List	330
12.8.2 Access Type	331
12.8.3 Linking	332
12.8.4 Weights, Multiple, Grip	332
12.8.5 Offsets	333
12.8.6 SaveCurrents	334
12.8.7 Sublisting	334
12.9 Inline Entries	335
12.9.1 Inline Entries vs. Regular Entries	336
12.9.2 Attributes of Inline Entries	337
12.9.3 Incorporating a Global Entry	338
12.9.4 Inline Entries and Lists	338
12.9.5 Token Reference Inline Entries	339
12.9.6 Function Definitions	340
12.9.6.1 THIS and Inline Entries	341
12.9.6.2 Using THIS to Define a Function	341
12.9.6.3 Parameter Tags	342
12.10 Using a File as an Entry	343
12.11 Inherited Attributes	344
12.11.1 Inheritance and Token Reference Inline Entries	346
12.12 Crossing Lists	347
12.12.1 Mapped Crossings	348
12.12.2 Checklist Storage	349
12.13 Optimizations	349
12.14 Script Operators and Functions Summary	349
12.14.1 Operators and Functions	350
12.14.1.1 Reference Operations	350
12.14.1.2 Math Operations	351
12.14.1.3 String Operations	352
12.14.1.4 List Operations	352
12.14.1.5 Other Operations	354
12.14.2 Operator Precedence	356
12.15 The Evaluator	356

Chapter 13. Experiment Scripting Reference 357

13.1	Experiment Scripting Basics	357
13.1.1	Introduction	357
13.1.2	Script Interpretation	357
13.1.2.1	Self-Modifying Scripts	358
13.1.3	Script Formats	358
13.1.4	The 'Experiments' Entry	359
13.1.5	Experiment Entries	359
13.1.6	Standard Attributes	360
13.1.6.1	Standard Experiment Attributes	360
13.1.6.2	Standard Trial Attributes	366
13.1.6.3	Standard Event Attributes	367
13.2	StimList and EventList Formats	368
13.2.1	StimList Format	368
13.2.1.1	Trial Attributes in StimList Format	370
13.2.1.2	Block Mode in StimList Format	370
13.2.2	EventList Format	371
13.2.3	Attribute Inheritance in StimList/EventList Format	372
13.2.4	StimList/EventList Event Names	373
13.2.5	StimList/EventList Optimization	373
13.2.6	Summary of Attributes for EventList and StimList Formats	374
13.2.7	StimList/EventList Compilation Details	375
13.3	Factor Format	376
13.3.1	Scripting the Experiment Hierarchy	376
13.3.2	Scripting the Factor Format Experiment Entry	378
13.3.2.1	Factoring and Linking Experiment Attributes	379
13.3.3	Scripting Groups	379
13.3.3.1	Factoring and Linking Group Attributes	380
13.3.4	Scripting Blocks	380
13.3.4.1	Factoring and Linking Block Attributes	381
13.3.5	Scripting Templates	381
13.3.5.1	Factoring and Linking Template Attributes	382
13.3.6	Scripting Events	383
13.3.6.1	Factoring Event Attributes	384
13.3.6.2	Linking Event Attributes to Template, Block, and Group Attributes	385
13.3.6.3	Linking Event Attributes to the Run Mode	385
13.3.6.4	Factor Format Tags	386
13.3.6.5	Constant Events in Factor Format	387
13.3.7	Scripting Factors	387
13.3.7.1	Scripting Factor Sets	388
13.3.7.2	Scripting Crossing Types	389
13.3.7.3	Scripting Access Types	391
13.3.7.4	Scripting Cell Weights	393
13.3.7.5	Scripting Nested Factors	393
13.3.8	Scripting Compact Factors	394
13.3.9	Scripting Factor Tables	395
13.3.9.1	Scripting Factor Table Structures	395
13.3.9.2	Scripting Factor Table Crossing Values	396
13.3.9.3	Nested Factor Values in a Factor Table	399

- 13.3.9.4 Scripting the Factor Set Scope 400
- 13.3.10 Scripting Factor Format Trial Counts 401
- 13.3.11 Technical Details of Factor Format Scripting 402
 - 13.3.11.1 Structural vs. Non-structural Attributes 402
 - 13.3.11.2 Attribute Inheritance in Factor Format 403
 - 13.3.11.3 Factor Format Optimization 404
 - 13.3.11.4 Factor Format Compilation Order 404
- 13.3.12 Summary of Factor Format 405
 - 13.3.12.1 Factor Format Entry Types Summary 405
 - 13.3.12.2 Factor Format Summary Object Attribute Groups 407
 - 13.3.12.3 Factor Format Summary Description Attribute Groups 408
- 13.4 Complex Attribute Formats 409
 - 13.4.1 Action Lists 409
 - 13.4.1.1 Specifying Action Lists 409
 - 13.4.1.2 Instances and ActiveUntil 410
 - 13.4.2 Start Reference 411
 - 13.4.2.1 Start Reference Format 411
 - 13.4.3 Duration 411
- 13.5 Trial Manager Variables 412
 - 13.5.1 Declaring Variables 412
 - 13.5.1.1 Variable Types 412
 - 13.5.1.2 Variable Declaration Entries 412
 - 13.5.1.3 Composite Types 413
 - 13.5.1.4 Type Declarations 415
 - 13.5.1.5 Built-in Variables 417
 - 13.5.2 Using Trial Manager Variables 417
 - 13.5.3 Variable Expression Syntax 417

Chapter 14. Actions and Devices Reference 419

- 14.1 Actions Reference 419
 - 14.1.1 Standard Actions 419
 - 14.1.1.1 Trial Termination Actions 419
 - 14.1.1.2 Event Scheduling Actions 419
 - 14.1.1.3 Unscheduled Stimulus Display Actions 420
 - 14.1.1.4 Miscellaneous Actions 421
 - 14.1.1.5 Trial Variable Actions 422
 - 14.1.1.6 Factor Format Actions 423
 - 14.1.2 Type-specific Actions 424
- 14.2 Stimulus Types Reference 424
 - 14.2.1 Text 424
 - 14.2.1.1 Text and Screen Attributes 424
 - 14.2.1.2 Text and Screen Experiment Attributes 429
 - 14.2.1.3 Text and Screen Actions 430
 - 14.2.2 Document 431
 - 14.2.2.1 Formatting characters 431
 - 14.2.2.2 Document Attributes 431
 - 14.2.3 Paragraph 432
 - 14.2.3.1 Paragraph Attributes 432

- 14.2.4 KeySequence 432
 - 14.2.4.1 KeySequence Attributes 433
- 14.2.5 PICT 433
 - 14.2.5.1 PICT Attributes 433
- 14.2.6 Pasteboard 434
 - 14.2.6.1 Pasteboard Attributes 434
 - 14.2.6.2 Pasteboard Experiment Attributes 435
- 14.2.7 SoundLabel 435
 - 14.2.7.1 SoundLabel Attributes 435
- 14.2.8 BBox 436
 - 14.2.8.1 .BBox Attributes 436
 - 14.2.8.2 BBox Experiment Attributes 437
- 14.3 Conditions and Inputs 437

Chapter 15. Trial Manager Technical Reference 441

- 15.1 Running Trials 441
 - 15.1.1 Loading Stimuli 441
 - 15.1.2 Running Events and Actions 442
 - 15.1.2.1 The START Event 442
 - 15.1.2.2 The Life of an Event 442
 - 15.1.2.3 Running an Event More than Once Per Trial 444
 - 15.1.2.4 Event Statistics 444
 - 15.1.2.5 The Life of an Action 444
 - 15.1.2.6 Ending a Trial 445
- 15.2 Screen Stimulus Display 445
 - 15.2.1 Screen Stimulus Loading 445
 - 15.2.2 How a Screen Stimulus is Drawn and Cleared 445
 - 15.2.2.1 Screen Timing 446
- 15.3 Playing Sound Stimuli 447
 - 15.3.1 Loading Sounds 447
 - 15.3.2 Sound Timing 447
 - 15.3.2.1 Actual Duration vs. Recorded Duration 447

Chapter 16. Configuring the User Environment 449

- 16.1 Setting up the Menus 449
 - 16.1.1 Item Entries 450
 - 16.1.1.1 Checkmarks 452
 - 16.1.1.2 Range Checking 452
 - 16.1.1.3 Open/Close Alert 453
 - 16.1.1.4 Menu Disabling 453
 - 16.1.1.5 'MenuName' and 'ItemName' 453
 - 16.1.1.6 Title 454
 - 16.1.2 Submenus 455
- 16.2 The Console 455
- 16.3 Custom Options 456
- 16.4 File Names 456
- 16.5 Log File 458
 - 16.5.1 Log File Format 458

16.5.2	Logging script information	458
16.6	Special Entries	459
16.6.1	Experiments	459
16.6.2	Execution Entries	460
16.6.3	Resources	461
16.7	PsyScopeStdLib	461
16.7.1	CurrentExperiment	461
16.7.2	Standard Menu Items	461
16.7.2.1	UserLevelMenuItem	461
16.7.2.2	SettingsMenuItem	462
16.7.2.3	DataFieldsMenuItem	462
16.7.2.4	ReverseVideoMenuItem	462
16.7.2.5	InputDevicesMenuItem	462
16.7.2.6	TimerMenuItem	463
16.7.2.7	OptimizeMenuItem	463
16.7.2.8	Test BBox	463
16.7.2.9	TurnOffBBox	463
16.7.3	Standard Menus	464
16.8	SubjectInfoLib	464
16.8.1	The Subject Menu	464

Chapter 17. Dialog and Function Extensions 467

17.1	Calling Sequence for Dialogs	468
17.2	Standard Dialogs and Functions Reference	469
17.2.1	Standard Configurable Dialogs	469
17.2.1.1	The Standard Attributes	469
17.2.1.2	Messages to the Standard Dialogs	471
17.2.1.3	Standard Dialog Descriptions	472
17.2.2	Miscellaneous Dialogs and Functions	476
17.2.3	Input Device Dialogs	477
17.2.4	Stimulus Attribute Dialogs	478
17.2.5	File Name Dialogs	480
17.2.6	Log File Related Functions	482

Part 5: Appendices

Chapter 18. Error Messages 489

18.1	Error Numbers	489
18.2	Global and Memory Errors	489
18.3	PsyScript Errors	490
18.4	User Environment Errors	493
18.5	Graphic Environment Errors	495
18.6	Factor Format Errors	499
18.7	Trial Manager Errors	502
18.8	Screen Manager Errors	507
18.9	Sound Manager Errors	510

18.10 Button Box Errors 511

Chapter 19. Configuring the Button Box 513

Chapter 20. Creating Picture Resources 517

Chapter 21. Creating SoundEdit™ Sound Files 519



Overview of the PsyScope Manual

0.1 Organization

The PsyScope user documentation is divided into five parts:

Part 1: Introduction to PsyScope

A step-by-step tutorial which explores the basic features of PsyScope and takes the user through the process of creating a working experiment in the graphic environment.

Part 2: Graphic Environment Reference

A reference for the basic experiment concepts, the dialogs in the graphic environment, the features for running and managing an experiment, and all of the menus and windows in the user environment.

Part 3: Scripting User Manual

An introduction to using PsyScript to define an experiment, including a scripting tutorial.

Part 4: Scripting Reference

An exhaustive reference for PsyScript, experiment description formats, user environment configuration, and dialog extensions.

Part 5: Appendices

A list of PsyScope's error messages, details about configuring the CMU button box, and instructions for creating PICT and digitized sound stimuli.

0.2 System Requirements

PsyScope will run on any Macintosh from the Mac Plus upward running System 6.0.5 or later.

0.3 Conventions

This manual can be used for learning PsyScope or as a reference for experienced users. We assume that those reading this manual have a basic knowledge of the Macintosh — how to click on things with the mouse, and how to use the menus.

For instance, when we say,

“Choose **Paste** from the **Edit** menu,”

we assume that you will know what me mean, without our having to say, “Move the cursor up to the **Edit** menu in the menu bar, push the mouse button down and hold it down while moving the cursor until the word **Paste** is hilited,” etc.

Key terms or concepts are *italicized and boldfaced* where they are first defined. References to items that actually appear on the screen, in windows or dialogs, are generally printed in **Chicago** font. The generic name for windows or dialogs of given type are capitalized (e.g., the Positions dialog). Text that is seen in the script is printed in *Courier*.

In examples, steps that should be followed in order are numbered; bulleted steps can be carried out in any order desired.

Part 1:

Introduction to PsyScope

Chapter 1. Introduction 1

Chapter 2. Running Your First Experiment 3

Chapter 3. Designing an Experiment 9



Chapter 1. Introduction

The laboratory microcomputer has become a crucial tool for conducting experiments in psychology. But students who wish to use this tool to construct new experiments typically find it out of their grasp, either because they are not expert programmers or because they cannot devote the hundreds of hours needed to program new experiments from the ground up. In this respect, psychology students are much like chemists without bunsen burners or geologists without compasses. Denied access to the basic tools of their trade, they are blocked from first-hand access to laboratory work in their science. The basic goal of PsyScope is to remove this technological barrier.

PsyScope is designed to eliminate the need for programming skills on the part of the user. Instead, the user can work within the graphic environment, which provides a direct visual representation of the design of the experiment. This will allow the student to focus on understanding the principles of experimental psychology, instead of the mechanics of computer programming. The crucial design features of PsyScope are:

1. **No programming** — The PsyScope system does not require the user to be a programmer or to learn a complex symbolic scripting language. These aspects of the system are made transparent to the user.
2. **Graphic environment** — To create this transparency, we have used the standard Macintosh interface of pop-up menus, buttons, scrollable windows, and icons to construct fully visual psychological experiments.
3. **Complete coverage of design types** — The PsyScope system is designed to support the full range of design types. Included are Latin square, Graeco-Latin square, blocked, embedded, linked, matched, random factor, fixed factor, and all other major design types.
4. **Accuracy checking** — To guarantee the accuracy of the program the student produces, the PsyScope graphic environment produces an “event schedule” that expresses in great detail the contents of each experimental trial. The student does not need to type in this list, since it is automatically compiled by PsyScope. It is made available to allow the student to make sure that the intended experimental design is actually being implemented on a particular run of the experiment. For example, if the student wants to examine the exact composition and shape of trial number 88, even before running the experiment, this can be easily done.
5. **A variety of input and output formats** — PsyScope supports all of the major output formats that computers can control. It is capable of playing digitized sound or speech. It can present words in Roman or non-Roman alphabets. It can display any form of computer graphics and now has primitive abilities to display animations. Currently,

simple experiment forms such as Stroop, ESP, Signal Detection, Picture-Word Naming, Mental Rotation, and so on are easy to design. Even more complex paradigms such as Moving Window or RSVP can be designed within PsyScope. Although PsyScope can support many input and output types, there are still others that must be implemented as separate extensions.

6. **Precise timing** — For students who need full experimental timing precision, PsyScope can be run in conjunction with a piece of external hardware called the “button box” which provides timing accurate to one millisecond. The low-level timing routines required for this box and for the precise control of the screen have been worked out in detail and have been thoroughly tested on all machines in the Macintosh line. When developing this system, we paid close attention to software timing routines (Rensink, 1990; Westall, Perkey, & Chute, 1989). However, we found that a general solution to the problem required an external device.
7. **Clear representation of experimental designs** — The graphic environment provides a clear representation of the structure of the experiment. This representation can then be used as a reference point for instruction on the principles of experimental design, such as Shaughnessy and Zechmeister (1990).



Chapter 2. Running Your First Experiment

Let's begin our look at PsyScope by running a simple demonstration experiment called the "Acuity Experiment". To run the Acuity Experiment, you need to have a copy of PsyScope installed on your hard disk. PsyScope can be anywhere on your disk, but it is probably best to keep your experiments and PsyScope in the same general folder. The other thing you need to run the Acuity Experiment is a copy of the script which should be called simply "Acuity Experiment script".

2.1 Open the Experiment

When you have PsyScope and "Acuity Experiment script" in place, double click on the icon for "Acuity Experiment script". This should launch PsyScope. First you will see a window with the PsyScope logo. Next, the logo will disappear and you will see two new windows. The first is the PsyScope Console window which looks like this:

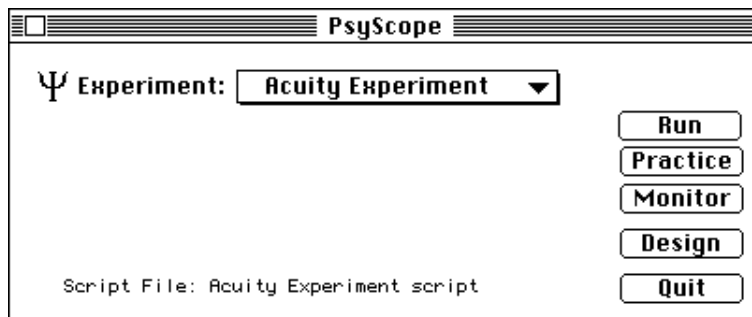


Figure 1 – PsyScope Console

The **Experiment** pop-up menu lists all of the experiments you currently have available. At this point the only available experiment is called "Acuity Experiment". The five buttons on the right allow you to control various high-level functions in PsyScope. If you click on the **Quit** button or the close box at the top left of the console window, you will quit PsyScope or close the script. Don't do this quite yet.

The second window is the Design window, which has a set of icons connected with lines. If you close the Design window, you can re-open it from the Console by clicking on the **Design** button. Try closing the Design window (by clicking on its close box) and then re-opening it with the **Design** button in the Console.

2.2 Run the Experiment

Next we are ready to run the experiment. This can be done by typing Command-R at any time. Alternatively, you can click on the **Run** button in the PsyScope console window or you can choose **Run** under the **Run** pulldown menu at the top of the screen. Each of these three ways of issuing the command leads to the same result.

As soon as the experiment is loaded, an instructions screen will come up telling you about the operation of the three keys. The basic idea is that you should wait for the asterisk to come on. Then you should press the “2” key as soon as you are ready for the next word. The word or non-word will come on immediately and you will have to decide whether it was a word or a non-word. If you think it was a word you should type “1”. If you think it was a non-word, you should type “3”. The experiment will run through 8 trials. Give it a try. If you want to interrupt the experiment, hold down Command-. (Command-Period).

After each trial, data are written to the output file. You can view this output file by typing Command-O to open a file, or choosing **View Data File** under the **Utilities** menu at the top of the screen. You can perform the normal Macintosh operations of printing and editing this file. If you break from the experiment (using Command-.) your data for completed trials will still be saved. The format of the data file, and how to customize this is discussed in below, and in detail in “Part 2: Graphic Environment Reference, 6.1.4 The Data File”, p217.

Another file that is automatically generated whenever you run PsyScope is the “PsyScope.log” file. Unless you specify otherwise, this file is stored in the folder which contains the PsyScope application. The log file lists the times at which you started experiments, the numbers of trials you ran, error messages, and so on.

2.3 Look at the Experiment in the Design Window

There are three factors in the Acuity Experiment. One is the position of the stimulus on the screen. There are five positions: far left, near left, center, near right, and far right. The second factor is the size (font pitch) of the stimulus which is either 12, 18, or 24. Finally, the stimulus is either a word or a non-word. From the viewpoint of experimental design, this is a fully-crossed three factorial within-subject design. You can see this design more clearly by double-clicking on the Factor Table icon (see “Figure 2 – Factor table icon”) in the Design window to open the Factor Table window.



Figure 2 – Factor table icon

		Small	Middle	Large
LeftFar	Word	1	1	1
	NonWord	1	1	1
LeftNear	Word	1	1	1
	NonWord	1	1	1
Center	Word	1	1	1
	NonWord	1	1	1
RightNear	Word	1	1	1
	NonWord	1	1	1
RightFar	Word	1	1	1
	NonWord	1	1	1

Figure 3 – Factor Table window

This window specifies a within-subject design in the full version of the experiment because each subject sees stimuli in each of the cells of the design when the number of trials is set to 30 (that’s how many cells there are). We have set the number of trials to 8 rather than 30 to keep your introduction to PsyScope brief (that is, to keep you from hitting the **Quit** button).

2.4 Change One or Two Things in the Experiment

To get a sense of how to work with an experiment in PsyScope, let’s change a couple of things in the Acuity Experiment. If you found the experiment a bit too easy, you can make it harder by cutting down the exposure time for the stimulus. Do this by going to the Design window (the one titled “Acuity Experiment” containing the various icons connected by lines), and double-clicking on the icon called “Stimulus” (if you don’t see this, look to see if the **Show Events** checkbox just below the window title is checked, and if it isn’t, click

on it to check it; a number of icons should appear, including the one for the Stimulus event). This brings up the Event Attributes dialog, which looks like this:

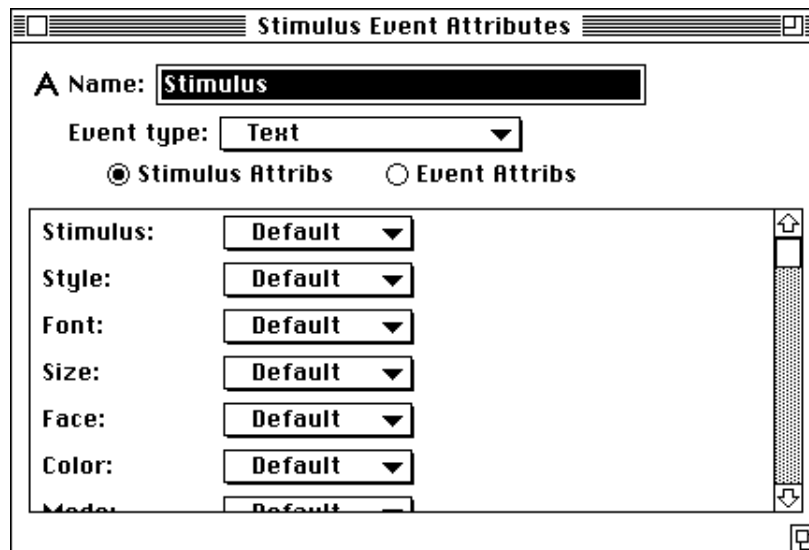


Figure 4 – Event Attributes dialog

To change the duration of the stimulus, click on the **Event Attribs** radio button. The window dialog will change to show a different set of attributes, including the **Duration** attribute with “100” written next to it, indicating that the current duration is 100 milliseconds. Clicking on the 100 will open up a new dialog, where you can change the duration by typing in some other value, say “50”.

For a second change, try reducing the smallest font from 12 to 8 point. To do this, you first need to select the “Small” level of the Size factor, so that you are working only with trials that will use the smallest words. Do this by going back to the Factor Table window, and clicking on the `Small` column heading. This will highlight all of the cells in that column,

corresponding to all of the trials in which the smallest size will be used for the stimulus word.

		Size	Small	Middle	Large
Position	Type				
LeftFar	Word		1	1	1
	NonWord		1	1	1
LeftNear	Word		1	1	1
	NonWord		1	1	1
Center	Word		1	1	1
	NonWord		1	1	1
RightNear	Word		1	1	1
	NonWord		1	1	1
RightFar	Word		1	1	1
	NonWord		1	1	1

Figure 5 – Clicking on a level

Once this column is highlighted, double-click on one of the cells within it, or click on the **Open Cells** button, and the Trial Template window will come up:

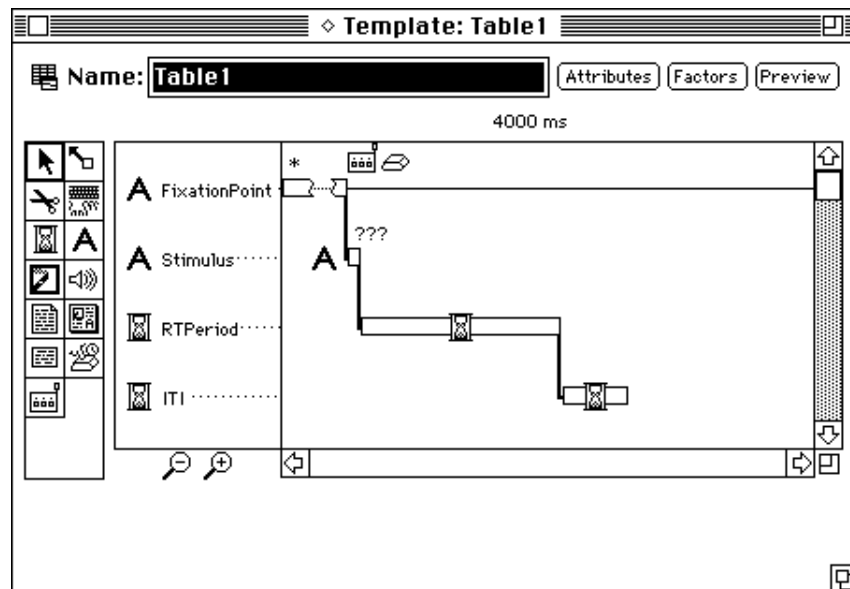


Figure 6 – Template window

Here, double-click on the icon for the Stimulus event, which will re-open the Event Attributes dialog. **Size** is about halfway down in the list of attributes. Open the pop-up menu next to this attribute, and choose **Set To**. This will open a dialog that will allow you to change the size of the stimulus from 12 to 8.

Try making both of these changes, and then running the experiment again. It should now be more difficult, particularly for the smallest stimuli.



Chapter 3. Designing an Experiment

This is an introduction to the structure of an experiment in PsyScope, and how to build experiments interactively within the graphic environment. If you are an experienced Macintosh user, this section should provide enough information to get you started “snooping around”. More detailed descriptions of the components of an experiment, and the tools and dialogs used to build them, are provided in “Part 2: Graphic Environment Reference”.

3.1 A Word About Scripts and the Graphic Environment

Information about each experiment in PsyScope is stored in a script. You can build experiments in one of two ways: by using PsyScript (the PsyScope scripting language) to write a script directly, or by using PsyScope’s point-and-click graphic environment to build the experiment interactively. When you design an experiment interactively, PsyScope actually writes the script for you. You can view this script at any time, and edit it directly. In fact, anything you do interactively is reflected immediately in the script, and most changes made directly in the script are reflected immediately in PsyScope’s interactive windows and dialogs.

PsyScript is a powerful scripting language that will allow you to design just about any type of experiment you can imagine. Sooner or later, you will want to make use of PsyScript to take full advantage of the power of PsyScope. However, learning PsyScript may take more time or energy than you have immediately available.

The graphic environment provides an intuitive, easy-to-use environment that will allow you to get most experiments up and running quickly, without having to learn PsyScript. This environment is also organized around the basic concepts of experimental design, so that it should seem natural to experienced psychologists, and will help introduce new students to these concepts. Even once you are familiar with PsyScript, you may still find it convenient to outline an experiment graphically, and then customize it using PsyScript.

This chapter will introduce you to the basic concepts and techniques needed to design an experiment graphically. In Part 2, you will find a complete reference to the structure and construction of a PsyScope experiment using the graphic environment. Part 4 is a reference for the PsyScript scripting language.

3.2 Creating a New Experiment

The first thing you will need to do to create a new experiment using the graphic environment is to create the script in which it will be stored.

Begin by double-clicking on the PsyScope application to open it. Once the application has opened, the menu bar should contain the following menus:



Figure 7 – Standard menu bar

Create a new script by doing the following:

1. Go to the **Design** menu, and choose **New Experiment...** A dialog asking you for the name of the new experiment will appear.
2. Type in the name you wish to give your experiment (let's say "Example Experiment").
3. Click on the **OK** button.

*Note: If another experiment is already open and you have made changes to it, a dialog will appear asking if you want to save the changes to the current experiment. Choose **Yes** if you want to save the changes, or **No** if you do not. If you are not sure, choose **Cancel** to review the changes before continuing. When you are ready, go back to step 1 above.*

4. A standard Macintosh file dialog will appear, asking you to name the file in which the script for your experiment will be saved. Enter the name of the file (say, "Example Experiment Script") and, if you wish, locate the folder in which you want to save it. Then click on the **Save** button.

All of your work will now be stored in the script file. It is a good idea to save this file to disk periodically by choosing **Save Script** from the **File** menu, or typing Command-S. If the **Save Script** menu item is grayed out, then there are no new changes to save. You can also save your script to a new file, by choosing **Save Script As...** from the **File** menu. A copy of the old file will remain on disk, but any new changes you make to the experiment will be stored in the new file. **Save a Copy as...** lets you save a copy of your script to a new file, while you continue to work with the old file.

Once you have named the experiment and created the script file, the PsyScope Console will appear, with the name of your experiment in the Experiment pop-up menu, and the name of your script listed as the *Script File* at the bottom. (If your version of PsyScope is configured to automatically open the Design window, the Console window will be behind it.

Click on the Console window to see it, or choose **Console** from the **Windows** menu, or type Command-1).

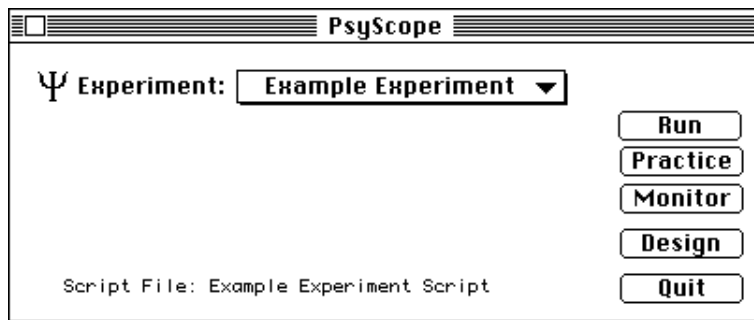


Figure 8 – The Console window for a new script

You are now ready to begin designing an experiment. Whenever you create a new experiment, PsyScope provides you with some initial building blocks. You can see these represented graphically in the Design window.

3.2.1 Using the Design Window

The Design window shows a graphic view of all the components of the current experiment. This is your “home base”. You use this get an overview of the experiment, to create new components and interconnect them, and to access windows and dialogs for customizing these components. “Part 2: Graphic Environment Reference, 5.2 The Design Window”, p107 provides a complete guide to all of the tools available in the Design window, and the techniques for using these.

If the Design window is not already opened, open it by doing one of the following:

- Click on the **Design** button in the Console window.
- Go to the **Window** menu and choose **Design**.
- Type Command-2.

The Design window will appear, and will look like this:

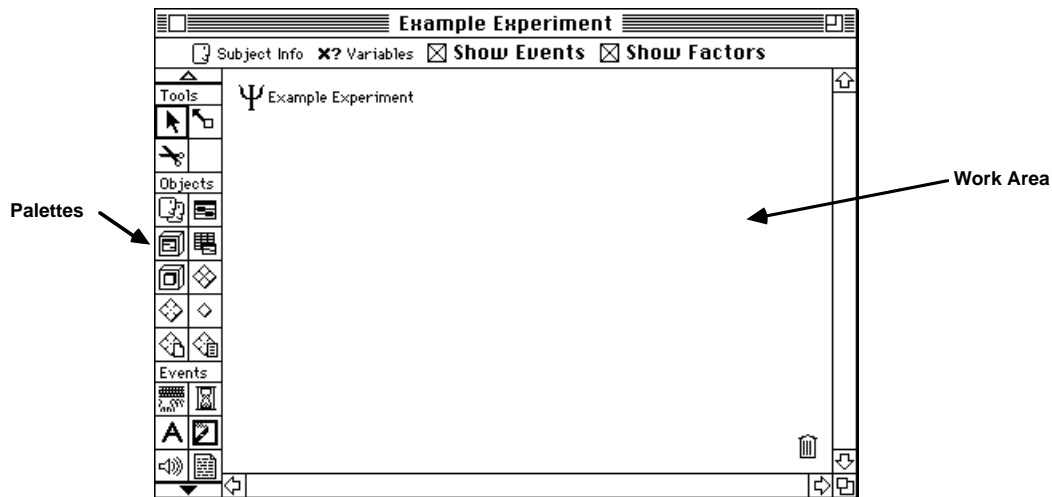


Figure 9 – Design window for a new script

When you create a new experiment, PsyScope starts you out with the experiment object, which is the first component that you need to get going. The icon for the experiment (Ψ) appears in the work area of the Design window.

Notice that there is also a palette on the left side of the window, which contains a number of other icons. These represent the other components that can be included in an experiment, and tools for linking them. In this chapter, we will focus on only those components that are needed to get simple experiments up and running. In the sections that follow, we will introduce you briefly to each of these components, show you how you can create them, and demonstrate some of the ways in which you can use them to design an experiment. The information in this chapter should help you get started quickly. You will find more detailed information about each component in “Part 2: Graphic Environment Reference”. This information will be useful to you when you begin to design more complex experiments.

To get going immediately, let’s build a simple experiment that presents two stimuli on the screen, one after the other, and then records your reaction time to the second stimulus. The components that we will need for this little experiment are *trial templates*, *events*, and *attributes*. Later in the chapter we will also be interested in *factor tables*.

3.2.2 Creating a Trial Template

The most basic element of a PsyScope experiment is a *trial*. A trial is simply a sequence of events during which one or more stimuli are presented, and responses from the subject are collected. An experiment may consist of several different types of trials (for example, trials during which stimuli are presented tachistoscopically and reaction times are measured, and others in which questions are presented and choices are recorded).

In PsyScope, you use a *trial template* (or simply *template*) to design each different type of trial that you want to include in the experiment. The idea here is simple: usually you will want to run a number of trials of the same type, which vary in their details (such as the stimulus actually presented, or how long to wait for a response), but all of which share a common overall structure (present the stimulus, wait a specific amount of time, and then record the response). You would define the structure for this type of trial using the trial template object. You would design a different trial template for each set of other trials that have a different structure. When you run the experiment, each actual trial is an instance of a particular trial template.

A trial template consists of a sequence of events that make up the trials of that type. These events, in turn, determine which stimuli to present, how and when to present them, when to wait for delay intervals, and what input to look for from the subject. You design and edit trial templates using the Trial Template window. First, however, you must create the trial template, and link it to the experiment.

To create a new trial template do the following:

1. Click on the trial template icon (see below), drag it to the work area of the Design window, and then release the mouse.



Figure 10 – The Template icon

2. When you release the mouse, a dialog will appear asking you to name the new trial template. Enter the name you want to use, and then click on **OK**, or press Return. (See “Part 2: Graphic Environment Reference, 5.2.5 New Object Name Dialog”, p114 about legal names.)

*Note: Whenever you create a new object in PsyScope, it will ask you to name that object. If you find this feature annoying, you can turn it off through the **Always ask for new object names** Design options (see “Part 2: Graphic Environment Reference, 7.6.4 Design Options”, p268). If you do this, PsyScope will automatically give names to objects for you, which you can later change at your convenience.*

A new trial template will now appear in the work area, along with the experiment. Also, the cursor should have changed to the link tool icon (see below), unless the automatic tool-switching option has been turned off (See “Part 2: Graphic Environment Reference, 7.6.4 Design Options”, p268). You can always select the link tool from the palette at the left of the Design window.



Figure 11 – The link tool

Using the link tool, link the trial template to the experiment:

3. Click on the trial template, and then drag to the experiment icon. As you drag, you should see a link line stretch from the trial template to the cursor.
4. Release the mouse button when the experiment icon is hilited. A link should now appear between the experiment and the trial template, indicating that the template is part of the experiment.

To work with the trial template, you will need to open it.

To open the Trial Template window:

- Double-click on the trial template icon in the Design window.

The following window will appear on your screen:

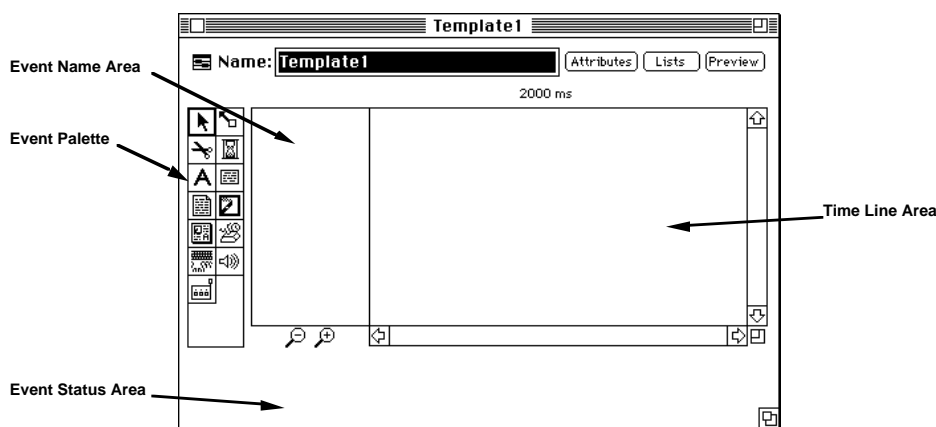


Figure 12 – Anatomy of the Trial Template window

Notice that the three areas of this window are empty. To begin designing the trial template, you will need to create some new events.

3.2.3 Creating a New Event

Trials are made up of events. You can think of an event as a period of time during which you want something specific to happen. You create a separate event for each thing that you want to happen during the trial, and then order the events according to the sequence in which you want them to occur.

There are separate *event types*, for each thing that you might want to happen during the trial. For example, there are event types for different kinds of stimuli — such as text, sounds and pictures — as well as special event types for time delays and subject input. Each type of event is represented by its own icon in the Event palette, at the left of the Trial Template window.

You design a trial template by creating new events, and sequencing them in the Time Line area.

To create a new event:

- Click on the icon in the Event palette for the type of event you wish to create, and then click in the Event Name area.

The different types of events are described fully in “Part 2: Graphic Environment Reference, 5.8.7.1 Event Types”, p177. For now, we will stick with **Text** events, which display a line of text on the screen. After choosing the **Text** event type (the icon, pictured below, is in the Events palette), the following dialog will appear asking you to enter the name of the new event:

A

Figure 13 – The **Text** event icon

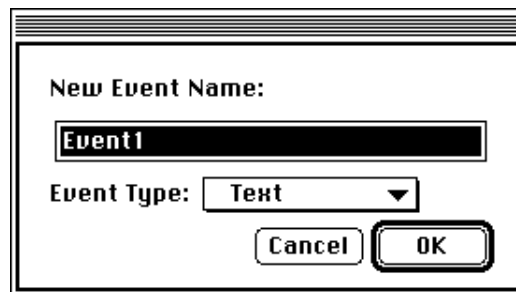


Figure 14 – New event name dialog

*Note: This dialog will not appear if the **Always ask for new object names** item is not checked in Design Options dialog. See also “Part 2: Graphic Environment Reference, 7.6.4 Design Options”, p268*

If you chose a different type of event, a comparable dialog will appear, listing its type in the **Event Type** pop-up menu.

- Type the name you want to give the event in the hilited area, and then click **OK**, or press Return.

The Trial Template window should now look like this:

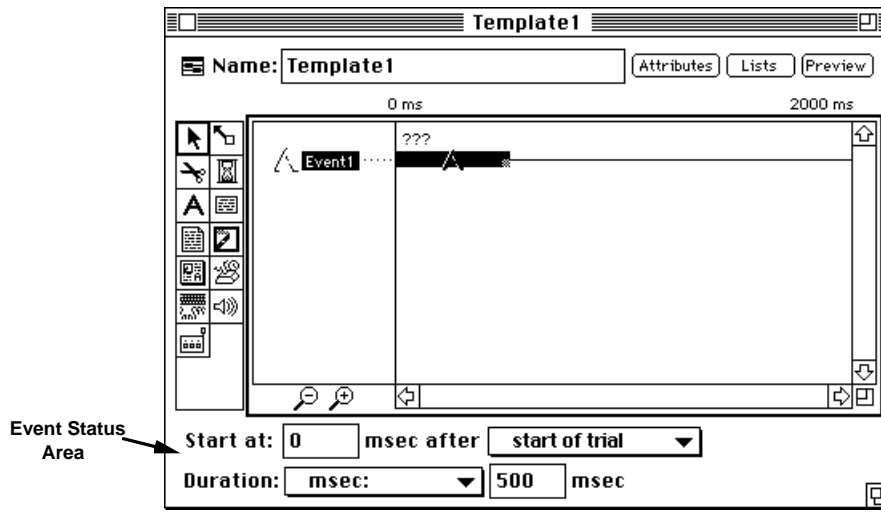


Figure 15 – The Template window with a new event

Note that both the name of an event and an icon for it have been added to the Event Name area, while an *event bar* for it appears in the Time Line area. You can select the event by clicking on any of these. Information about the selected event appears in the Event Status area. In the figure above, Event1 is selected, so information about when it starts and its duration appear in the Event Status area.

The trial template now contains a single event, but we have not yet specified anything about this event, other than the fact that it will be used to present text. To specify the details of an event (the specific stimulus to be presented, or how it will be presented), you must set its attributes.

3.2.4 Setting Attributes Using the Event Dialog

The attributes of an event determine all of the features of the event. You set these using the *Event Attributes dialog*.

- Open the Event Attributes dialog for Event1 by double-clicking on its name or icon in the Event Name area, or its event bar in the Time Line area. The following dialog will appear:

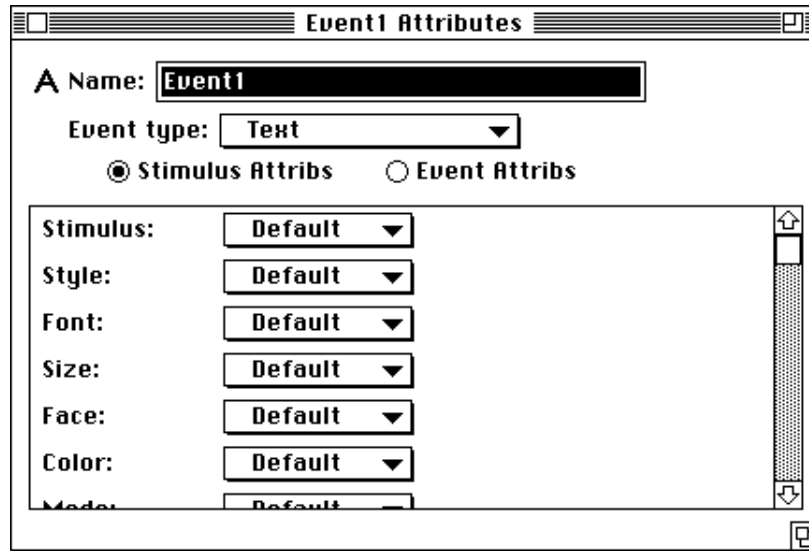


Figure 16 – The Event Attribute dialog

At the top of the Event Attributes dialog are a field for changing the name of the event, a pop-up menu for changing its type, and radio buttons to choose which set of attributes you want to work with. Each event is associated with two sets of attributes: *stimulus attributes* and *event attributes*. **Stimulus attributes** control aspects of the event that are specific to its type (e.g., the font or size of a text stimulus, or the volume of a sound stimulus). **Event attributes** control more general properties of the event that are common to all event types (e.g., their duration). The list area of the Event Attribute dialog displays the list of attributes belonging to the set you have chosen with the **Stimulus Attribs** and **Event Attribs** radio buttons. When you first open the Event Attribute dialog for an event, it displays the list of stimulus attributes for that event. You can switch to the event attributes by clicking the **Event Attribs** radio button.

Each attribute in the list area has a pop-up menu that lets you decide how you want to assign the value for that attribute. If you do nothing, or choose **Default**, PsyScope will give it a reasonable value. If you choose **Set To:**, the attribute will be given the value that you assigned to it; this will be the same on every trial. You can vary the value of the attribute from trial to trial by choosing one of the items under **Vary By:**; We will discuss this technique below (“3.2.8.4 Lists”, p59, “3.2.9.2 Varying by Block”, p72, “The Group Object and Varying By Group”, p83, and “3.2.12.2 Practice vs. Run Mode”, p93).

*Note: The only attribute that you must set is the **Stimulus** attribute. This attribute determines the stimulus that will be presented during the event. You will be given an error message if you try to run an experiment that contains any events whose stimulus attribute has not been set.*

Assign the text to be displayed during Event1 by setting its **Stimulus** attribute:

1. Go to the pop-up menu next to **Stimulus** in the list of attributes, and choose **Set To:**, as shown below. (If you cannot find the stimulus attribute, check to be sure that **Stimulus Attribs** radio button is selected.)

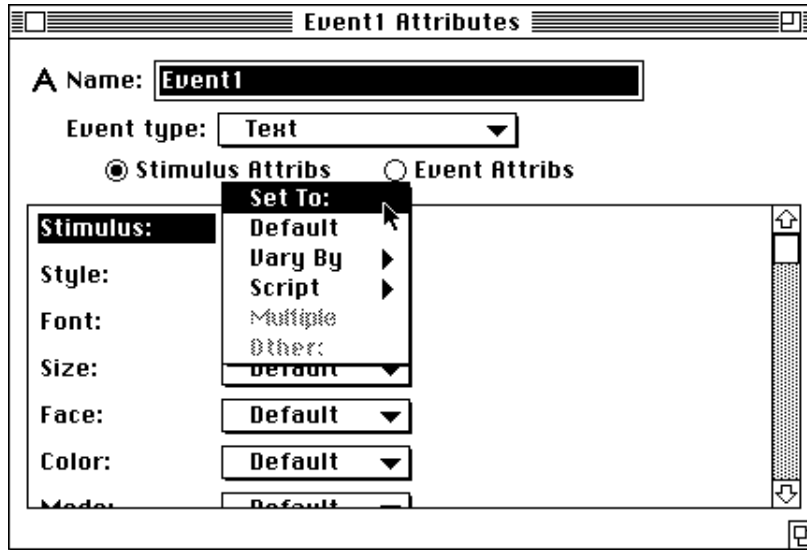


Figure 17 – Setting an attribute

2. A standard text dialog will appear. Enter the text you want displayed during the event, and then click **OK**.
3. The stimulus that you entered now appears next to a bullet to the right of **Set To:** in the **Stimulus** attribute menu:

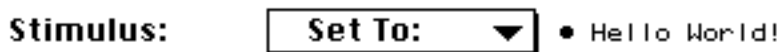


Figure 18 – Attribute showing a constant value for its setting

Single-clicking on the bullet or the stimulus will reopen the standard dialog, allowing you to change its value. The stimulus also appears above the event bar in the Trial Template window, in place of the “???” that was there when you first created the event:

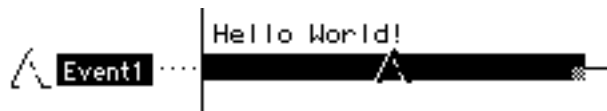


Figure 19 – Event in Template window with the stimulus set

Note: Whenever you see a bullet in a PsyScope dialog, you can click on it — or the item that appears next to it — to change the value of that item.

Set the values of any other attributes that you wish, by choosing **Set To:** in the pop-up menu next to the attribute's name. (The meaning of all stimulus attributes are described in "Part 2: Graphic Environment Reference, 5.8.7.3 Stimulus Attributes", p181.)

*Tip: You can enter the value of any attribute that can be specified as text directly from the Attributes dialog by selecting the attribute and typing Command-right arrow. This automatically chooses **Set To** from the attribute's menu, and creates a text field next to it for entering the value. (See "Figure 20 – Shortcut for setting an attribute value".) Many attributes for which you usually use pop-up menus can also be assigned by text (e.g., colors, font names, font sizes, etc.), although you are then responsible for making sure that you enter a legal value.*



Figure 20 – Shortcut for setting an attribute value

Once you have created at least one event, and assigned its stimulus value, you can run a trial.

Run the trial by doing one of the following:

- Make the Console the active window (by clicking on it, choosing its name in the **Window** menu, or typing Command-1), and then click on the **Run** button.
- Go to the **Run** menu and choose **Run**.
- Type Command-R.

The text that you entered as the stimulus attribute for the event should appear at the center of a blank screen for 1/2 second.

3.2.5 Timing and Sequencing Events

You control the timing of the events in a trial by setting their durations and by linking events to one another in a sequence.

3.2.5.1 Timing an Event

You can make an event last a set amount of time, or allow it to last until something happens, such as input generated by the subject, the end of the trial, or — for certain types of stimuli

(such as sound) — until the end of the stimulus itself. You set the *duration* of an event by first choosing how you want it to be decided (by a fixed time value, or by some occurrence or condition), and then providing any necessary details (i.e., the amount of time, or the specific occurrence that will end the event). You can do this from either the Trial Template window or the Event dialog, whichever is most convenient. If you do not explicitly set the duration of an event, PsyScope will assign it a default duration, depending upon its type. For **Text** stimuli, this is 500 milliseconds.

Note: For screen-based stimuli — such as text and graphics — the actual duration of the stimulus is constrained by the refresh rate of the screen. For most screens this is 66 Hz, which means that the actual duration of the stimulus will be a multiple of 16.6 msec. See “Part 4: Scripting Reference, 15.2 Screen Stimulus Display”, p445 for a more complete discussion of how PsyScope handles the timing of screen-based stimuli.

Set Event1 to last until the mouse is clicked by doing one of the following:

1. Select Event1 in the Trial Template window (single click on it; double-clicking will open the Event dialog).
2. Go to the **Duration** pop-up menu in the Event Status area (at the bottom of the Trial Template window), and choose **Conditions...**. The Duration dialog will appear.

or

1. Open the Event dialog, or click on it to make it active if it is already open.
2. Click on the **Event Attribs** radio button. The list of attributes will change to the event attributes.
3. Go to the **Duration** menu, and choose **Set To:**. The Duration dialog will appear.
4. Choose **Conditions...** in the **Duration:** pop-up menu at the top of the Duration dialog.

Either method will open the Duration dialog, and show **Conditions:** selected in its pop-up menu, with a list of input devices below it:

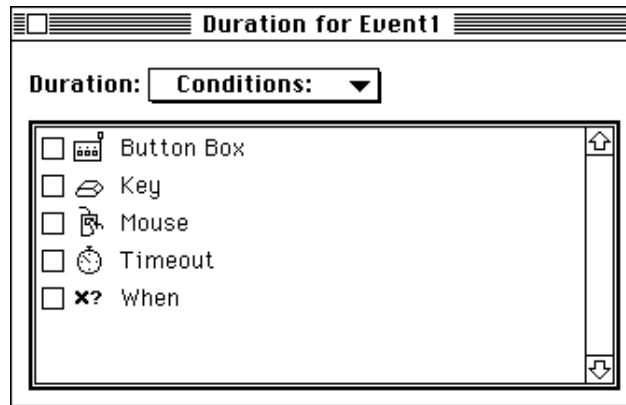


Figure 21 – The Duration dialog in Conditions mode

Conditions can be used to terminate an event in three ways: when the subject responds on an input device (clicks or moves the mouse, presses a button, etc.), when a certain amount of time has elapsed (the **Timeout** condition), or when a trial variable expression evaluates to true (the **When** condition). If you choose more than one of these, the event will end as soon as the first condition occurs. We will discuss conditions more fully below, in “3.2.7 Conditions and Actions”, p30.

To select mouse-click as the condition for terminating the event:

- Click on the mouse icon (see below) or the checkbox next to it. A check will appear in the checkbox, and • Click will appear next to **Mouse** in the list of devices.



Figure 22 – The Mouse icon

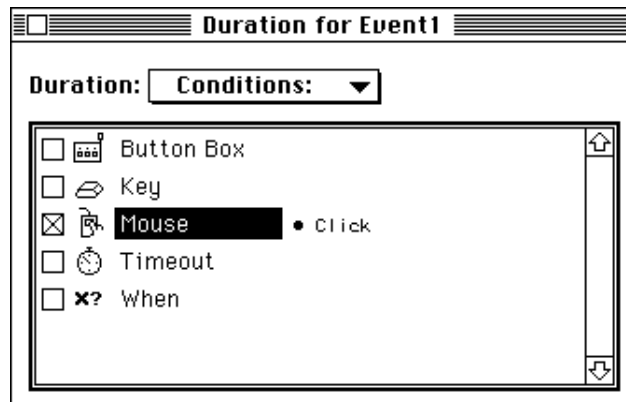


Figure 23 – The Duration dialog with Mouse selected

Whenever you select an input device, PsyScope makes some default assumptions about what response you expect from the user on that device (see “Part 2: Graphic Environment Reference, 5.8.7.1 Event Types”, p177 for the defaults for each input device). For the mouse, this is a click. You can change the response you want PsyScope to look for by double-clicking on the name of the device in the list or, if the device is already checked, single-clicking on the bulleted list of responses that appear next to it.

The **Timeout** condition specifies the maximum amount of time that PsyScope will wait for the input to occur, after which the event will end regardless of whether the input has occurred. If you check **Timeout**, a default value of 500 msec will be assigned. You can double-click on the icon or label to open a dialog in which you can enter a different maximum amount of time to wait.

Close the Duration dialog (by clicking in its close box). Notice that the event bar in the Trial Template window is now “broken”, and that a mouse icon appears at to the right of it:

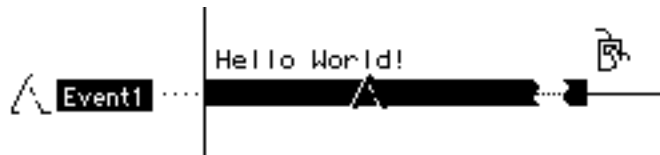


Figure 24 – A **Text** event that ends with a **Mouse** click

Try running the trial, using one of the methods described above. The stimulus should now appear on a blank screen, and remain there until you click the mouse. If you specified a **Timeout** value, then the text should disappear after that amount of time if you don't click the mouse.

The length of the event bar in the Trial Template window tells you about the duration of the event. If the event has a fixed duration, then the length of its event bar tells you how long the event will last. You can change the duration by clicking on the event bar, and resizing it using the handle in its lower right hand corner. As you change the size, the value for the duration changes in the box at the bottom of the Trial Template window. You can also set the value of the duration directly, by entering a number in the box.

If the duration of an event depends on an occurrence (such as a mouse click), then the event bar appears broken, indicating that its exact duration is not known; it will vary from trial to trial, depending upon when the input (or other terminating condition) actually occurs. You can still resize the event bar, however this will not affect the duration of the event. For this type of duration, one or more icons will also appear at the end of the event bar, indicating the conditions that will end the event. Clicking on any of these icons will show the duration menu.

3.2.5.2 Sequencing Events

Most trials that you design will, of course, involve more than a single event. For example, in the little experiment that we are designing, there are two events, one of which follows the other. You sequence events by linking them in the Trial Template window.

Add another **Text** event and assign it a stimulus using the methods described above. The Trial Template window should now look something like this:

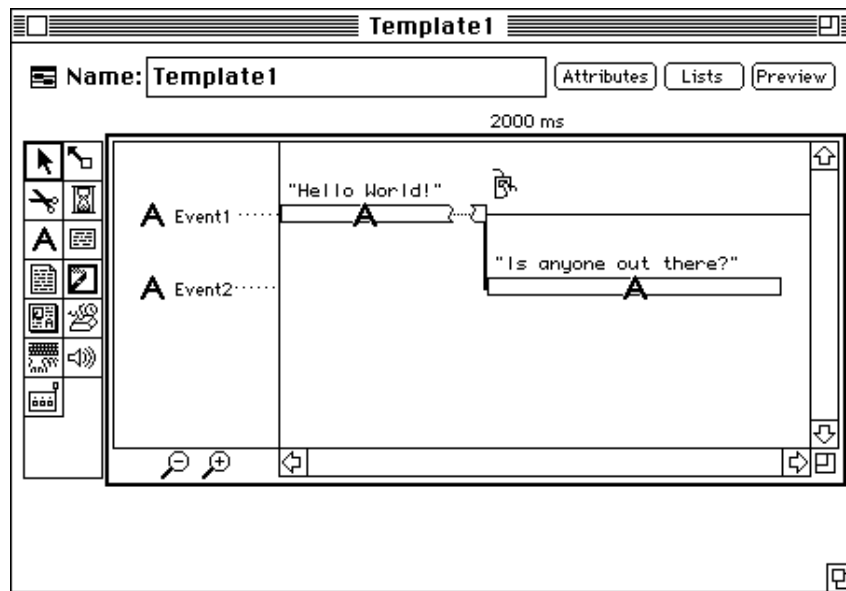


Figure 25 – Template with more than one event

Note: Whenever you add a new event to a trial template, PsychoPy automatically links it to the end of the last event created.

Switch the durations of the two events, so that the first lasts only 500 milliseconds, and the second lasts until you click the mouse. To change the duration of the first event to 500 milliseconds, you can choose **Default** in the **Duration** menu, or choose **msec:** and then enter 500 as the value.

Try running the trial. The first stimulus should flash briefly, followed immediately by the second which remains on the screen until you click the mouse.

Now, add a delay between the two events, by doing the following:

- Add a **Time** event, by clicking on the **Time** icon (see below) in the Events palette of the Trial Template window, and then clicking in the Time Line area. Call the event “Delay”.



Figure 26 – The **Time** icon

The event should appear linked to the second event. You may need to resize the window to see all of the events. Alternatively, you can rescale the Time Line area by clicking on the demagnify icon (see below) just below it.



Figure 27 – The demagnify icon

As noted above, whenever you create a new event, PsyScope links it to the last event in the sequence. However, you can rearrange things as you wish.

To change the sequence of the events, so that the time event comes in between the first and second events, use one of the following methods to re-link the events.

1. Select the Delay event in the Trial Template window.
2. Go to the second of the two pop-up menus that appear to the extreme right of **Start at:** in the Event Status area.
3. Choose Event1 (or the name of first event, if you have changed it):

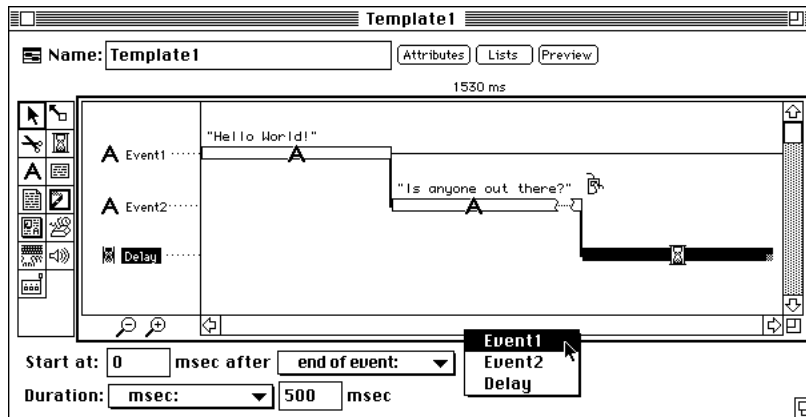


Figure 28 – Setting an event starting time

4. Follow the same set of steps to link the second event to the end of the Delay event.

Here is an alternative method:

1. Hold down the command key and click on the Delay event. The cursor will change to the link tool (see below).



Figure 29 – The link tool

2. Click on the right half of the event bar for first event. The Delay event will now be linked to the end of first event.

3. Follow the same procedure to link the second event to the end of the Delay event.

The Trial Template window should now look something like this:

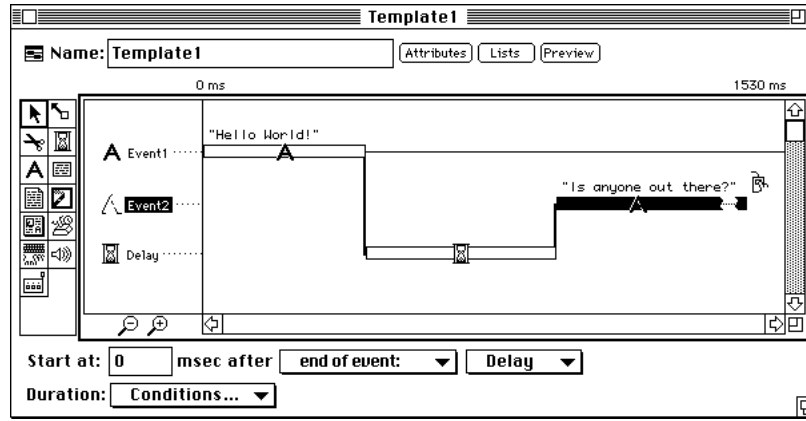


Figure 30 – Template window with a re-linked event

Try running the trial. The stimulus for the first event should appear briefly, followed by a blank screen for a brief period, and then the stimulus for the second event.

Using the Trial Template window, you can arrange the order and timing of events in any way that you want. This can be done using the menus and value boxes of **Start at:** in the Event Status area, or by manipulating the event bars directly. In general, you will find that there is more than one way to accomplish a particular goal. For example, to introduce a delay between the first and second stimulus events in the example above, you could have set the second event to start at 500 milliseconds after the end of the first event, rather than adding a separate time event. Try doing this.

Re-link the second event directly to the first event, by using one of the methods described above. Then, remove the Delay event by doing the following:

1. Select the Delay event.
2. Use either the **Cut** or **Clear** command from the **Edit** menu. The event will be moved to the trash, and can be recovered later if you wish.

Now, make sure the second event is selected, and then use one of the following two methods to delay the start of the second event for 500 milliseconds after the end of the first event:

- Enter “500” in the value box just next to **Start at:** in the Event Status area. You make this area active by clicking on it with the mouse, or pressing the tab key until the number in it is hilited.

or

- Drag the event bar for the event to the right, until the number next to **Start at:** in the Event Status area is “500”.

Trial Template window should now look like this:

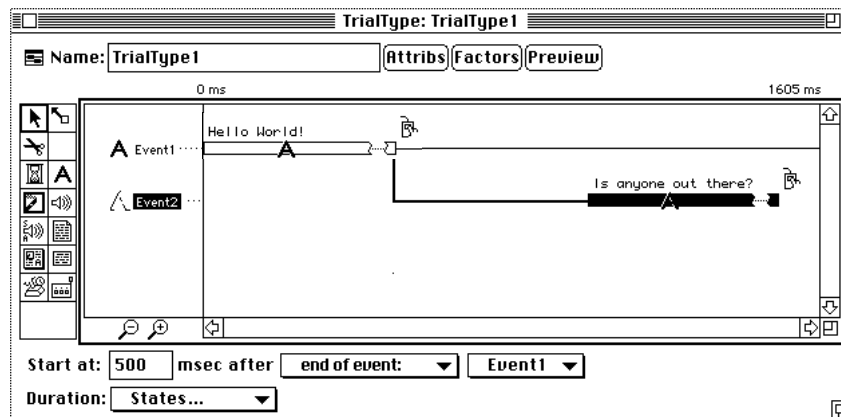


Figure 31 – Template window with delayed event

Running the trial now should produce the same results that it did earlier, when the delay event was used.

In general, if you just want to delay the beginning of an event, do this by changing the value for **Start at** in the Event Status area (i.e., when the event starts relative to the event that it is linked to, or to the beginning of the trial). However, if you want something to happen during the delay, then you should add a new event. For example, if you want to record a response to the first event during the delay between it and the second event, then use a delay event or an input event. In the little experiment we have been working on, we want to record the subject's reaction time to the second event. The next section describes how to do this.

3.2.6 Recording Responses

Recording a response in PsyScope is actually a special case of scheduling an *action* to occur during a trial. Actions are operations that you can schedule to be performed under specified conditions during the running of a trial. This section will show you how to use conditions and actions to record a response. In the next section, we will show you how to schedule other kinds of actions to carry out a variety of operations during a trial.

To record a response (e.g., a mouse click or key press), you must schedule the `RT[]` action to take place when the type of response that you are interested in has occurred. (The brackets following `RT` indicate that it can be assigned parameters. You do not need to worry about this now — they are included here simply for accuracy; parameters are discussed in the next section, “3.2.7 Conditions and Actions”, p30). The `RT[]` action records the time and nature of the response in the data file (the data file is described in “Log File and Data File”, p79; also see “Part 2: Graphic Environment Reference, 6.1.5 The Log File”, p222 and “Part 2: Graphic Environment Reference, 6.1.4 The Data File”, p217). You schedule the `RT[]` action by setting the **Actions** attribute of the event during which you expect the response to occur. The **Actions** attribute is in the **Event Attribs** attribute set. You must also specify the type of response, or *condition*, that will trigger the `RT[]` action — that is,

the response you wish to record. When the RT[] action is triggered, it records the time at which the response occurred relative to the beginning of the event which owned the RT[].

To record a key press during the second event:

1. Open the Event dialog for Event2, and select the **Event Attribs** radio button.
2. Go to the **Actions** pop-up menu, and choose **Set To:**. The Actions dialog will appear.
3. Click on the **New** button in the Actions dialog. It should now look like this:

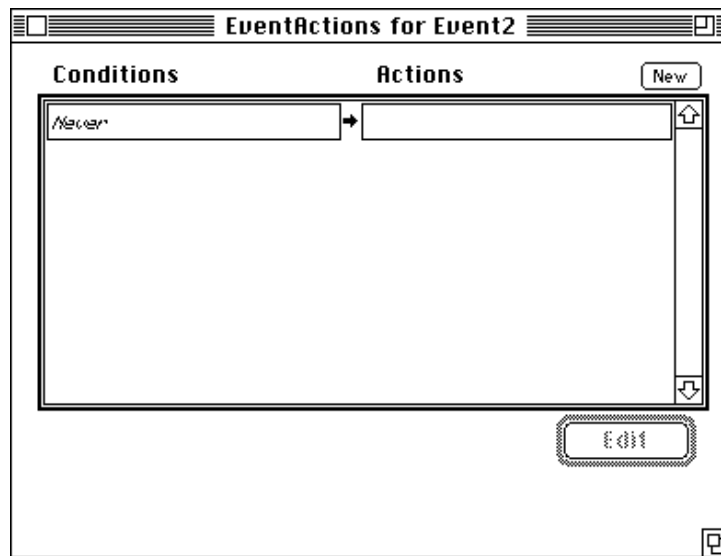


Figure 32 – The Actions dialog with one empty condition-action pair

Notice that there are two boxes separated by an arrow, below the **Conditions** and **Actions** headings, respectively. You specify the action that you want to perform in the actions box, and the condition that you want to trigger it in the conditions box. Follow these remaining steps to record a key press during Event2:

4. Double-click on the box on the left (under the **Conditions** heading, with *Never* written in it). The Conditions dialog will appear:

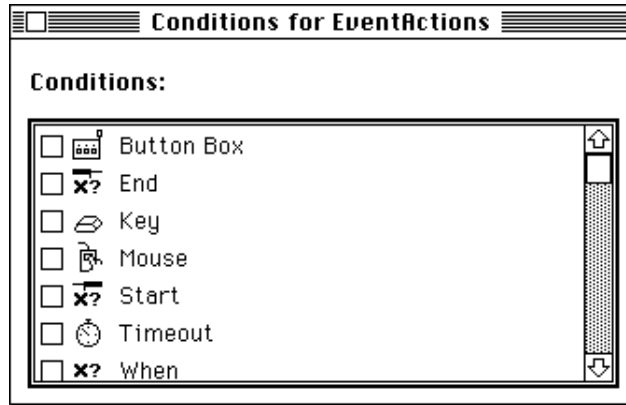


Figure 33 – The Conditions dialog

5. Click on the **Key** icon (see below) or the checkbox next to it, or double-click on the word **Key**. The Key dialog will appear:



Figure 34 – The **Key** icon

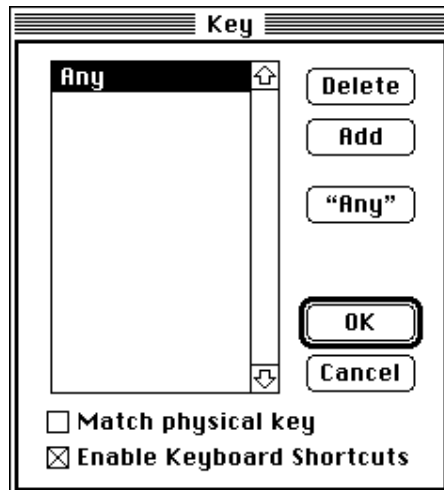


Figure 35 – The Key dialog

6. **Any** is the default. You could replace this by typing in the keys to which you want to record a response, but for now leave **Any** as the value to record any key press. Close the dialog when you are done.

- Go back to the Actions dialog, and select the box to the right of the arrow, under the **Actions** heading. A pop-up menu marked **New** will appear to the right of the **Actions** heading. Choose **RT** from this menu.

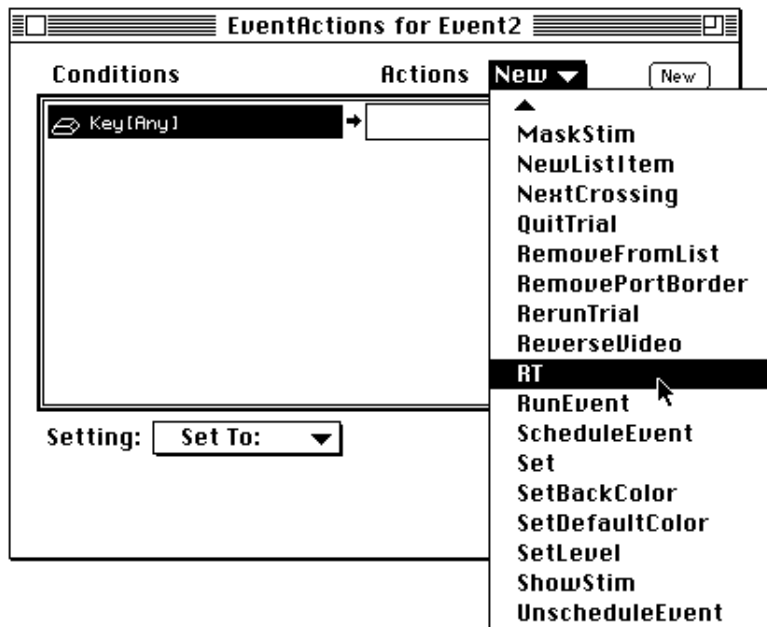


Figure 36 – Selecting a new action in the Actions dialog

The Actions dialog should now look like this:

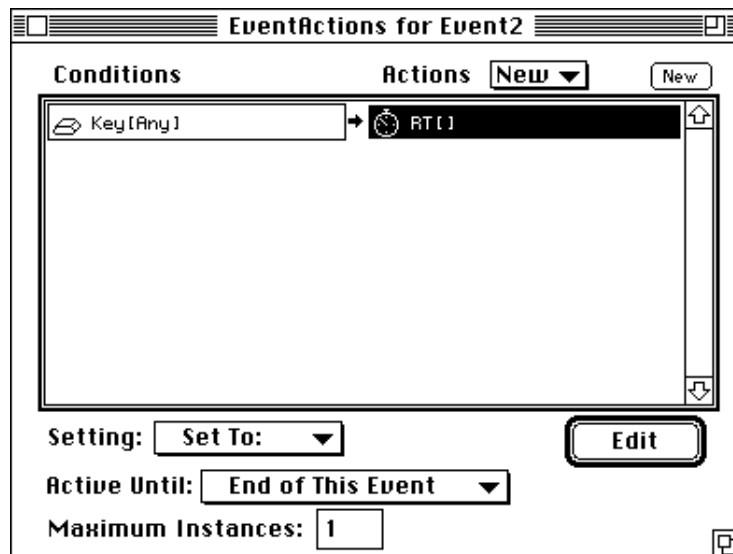


Figure 37 – The Actions dialog with a new action

Try running the trial. Remember, Event2 will continue until you click the mouse (since that is how its duration is set), unless you specified a timeout value in the Duration dialog.

If you press one of the keys that you set in the Key dialog before the event ends, then the key and the time that it was pressed — relative to the beginning of Event2 — will be recorded in the data file. You can see this once the trial has ended by opening the data file using the **View Data File** command in the **Utilities** menu. The format of data files is described in “Part 2: Graphic Environment Reference, 6.1.4 The Data File”, p217.

Recording input from the subject is one example of how you can use conditions (such as `Key[Any]`) and actions (such as `RT[]`) to perform a variety of operations during the running of a trial in PsyScope. There are a number of other actions that you can execute, and a variety of conditions that you can use to trigger these. Some examples of these are discussed in the next section.

3.2.7 Conditions and Actions

Actions are operations that are performed under specified **conditions** during the running of a trial. The action is performed when the specified condition occurs. You can think of the occurrence of the condition as “triggering” the action.

Note: You should not confuse this meaning of the term “condition” with its meaning as a particular crossing of the factors in the factor table (as in a “condition of the experiment”). This other meaning of the term condition is discussed below, under “3.2.8.1 Creating a Factor Table”, p36, and in “Part 2: Graphic Environment Reference, 5.7 Factors and Lists”, p129.

There are three types of conditions that can be used to trigger an action: 1) the start or end of an event; 2) subject input (such as a mouse click or a key press); and 3) the value of a trial variable expression (this is an advanced feature of PsyScope, not discussed further here; see “Part 2: Graphic Environment Reference, 5.10 Trial Manager Variables”, p205 for a description of variables).

As you saw in the previous section, conditions and actions are set using the **Actions** event attribute. There, you set the `RT[]` action to occur when the subject pressed a key. As you probably noticed, there are a variety of other actions that are available. For example, you could have PsyScope beep at the beginning of an event with the `Beep[]` action and the `Start[]` condition; you could initiate or terminate an event when a particular response occurs using the `RunEvent[]` and `EndEvent[]` actions and an input condition; or you could quit the trial when Command-Q is pressed, using the `QuitTrial[]` action and `Key[CMD-q]` as the condition. Explore some of these, by setting the **Actions** attribute of an event. A complete description of all of the actions in PsyScope is in “Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions”, p198.

Note that you can associate a number of actions with a given condition, with the result that all of the actions will be performed when that condition occurs. Conversely, you can associate actions with multiple conditions, so that when any of the specified conditions occurs then all of the associated actions will be performed. As an example, let’s embellish our little experiment, by causing PsyScope to beep when you respond to the second event.

Schedule a beep to occur when the RT[] action is performed in response to a key press:

1. Open the Actions dialog for Event2 if it is not already opened (see steps for the example in the previous section)
2. Click on the box with the RT[] action in it to select it.
3. Choose **Beep**, click on the **New** menu next to the Actions heading.

The Actions dialog should now look like this:

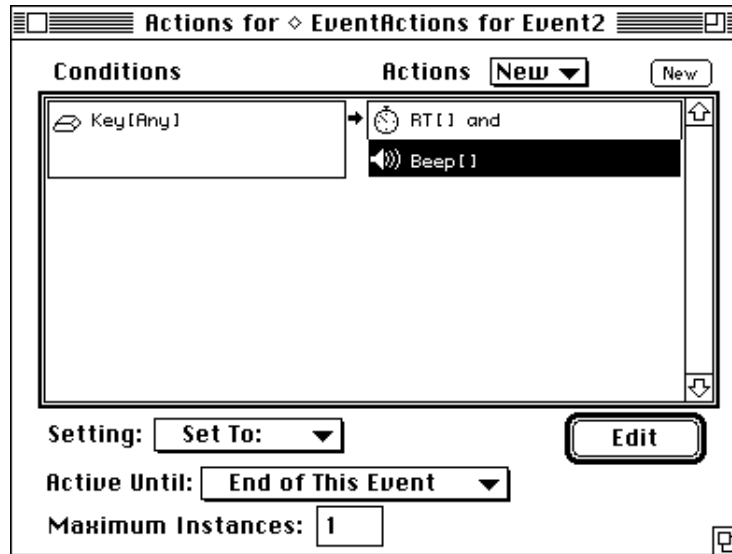


Figure 38 – Actions dialog with a Beep[] action

Try running the trial. You should now hear the system beep when you press any key during Event2.

You can add another condition to trigger the RT[] and Beep[] actions by double-clicking in the conditions box, and checking the box for one of the other conditions. For example, add Mouse[Click] as a condition by doing the following:

1. Click anywhere in the conditions box to select it, and then double-click on it to open the Conditions dialog.
2. Click on the mouse icon or the checkbox next to it, or double-click on the word **Mouse**.
3. In the Mouse dialog, check the box marked **Click**, or click on the icon of the finger pressing the mouse button, and then click on **OK**.
4. Close the Conditions dialog.

The Actions dialog should now look like this:

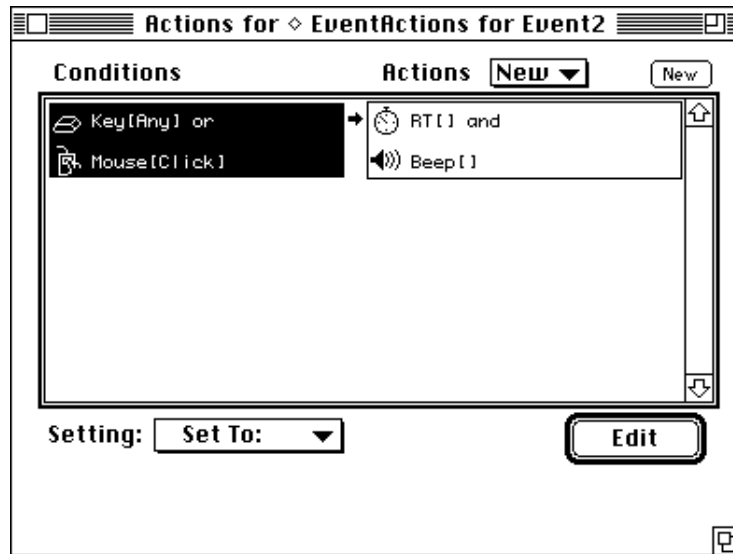


Figure 39 – Actions dialog with more conditions

Now, when either you press a key or you click the mouse, you will hear the beep and the RT will be recorded. Note that if you click the mouse, not only will these actions occur, but the event will end as well; this is because `Mouse [Click]` is also the condition for ending the event.

You can delete actions by selecting them and then choosing **Clear** from the **Edit** menu, or typing Command-X. You must open the Conditions dialog and un-check the device to delete a condition. New sets of condition-action pairings can be added by clicking the **New** button at the top right of the Actions dialog.

Most actions have one or more *parameters*, which can be used to control details about how they will be performed. In most cases you will not need to specify the parameters of an action — PsyScope assumes reasonable default values for them. For example, the `Beep []` action plays “correct beep” (build into PsyScope) if no parameters are specified. However, action parameters can be useful for customizing your experiment. For example, `Beep []` takes a parameter that directs it which sound to use, which can be any sound in the system file, or a ‘snd’ resource that has been opened by PsyScope (see “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216 for a discussion of how to add resources in PsyScope). You set the parameter for an action by double-clicking on the action in the Actions dialog. A dialog will appear with the list of parameters, and a pop-up menu for setting the value of each that is the same as the ones for setting the value of attributes.

Change the Beep that occurs when the subject presses a key by doing the following:

1. Double-click on the `Beep []` action in the Actions dialog. A dialog will appear, with the **Beep** parameter listed (here, the name of the parameter happens to be the same as the action itself).

2. Choose **Set To:** from the pop-up menu next to **Beep**. A dialog with the list of available beeps will appear. Choose the one you want, and then close the dialog.

Pressing the a key or clicking the mouse during Event2 should now produce the new beep sound.

The parameters for all of the actions available in PsyScope are described in “Part 4: Scripting Reference, 14.1 Actions Reference”, p419 (you can also get on-line help for the action you are interested in).

The *active until* and *maximum instances* parameters of an action allow you to control, respectively, the period of time that it is *active* — that is, during which it can be triggered — and how many times it can be performed. Normally, an action is active for the duration of the event that it belongs to. For example, the `RT[]` and `Beep[]` actions in the example above will only be performed if a key is pressed or the mouse is clicked during Event2. If neither of these responses occurs until after Event2 ends, then neither of the actions will be performed. Under some circumstances, however, you may want to extend the period during which an action is active, beyond the duration of the event. For example, you might want to record a response to an event that is very brief. To do this, you could set **Active Until** for the `RT[]` action to be **At Least One Instance**. This would insure that the action remained active until it had been performed at least once. In fact, the trial would continue, even if there were no more events left to run, until the appropriate input occurred to trigger that action. Another way to make an action remain active beyond the event that it belongs to is to set **Active Until** to the end of some other event (for example, a **Time** event following the brief stimulus event).

To demonstrate the above example, make Event2 very brief and then change:

1. Set the duration of Event2 to be 100 msec.
2. Open the Actions dialog for Event2.

3. Select the RT[] action and set **Active Until** to be **At Least One Instance**. Then do the same for the Beep[] action.

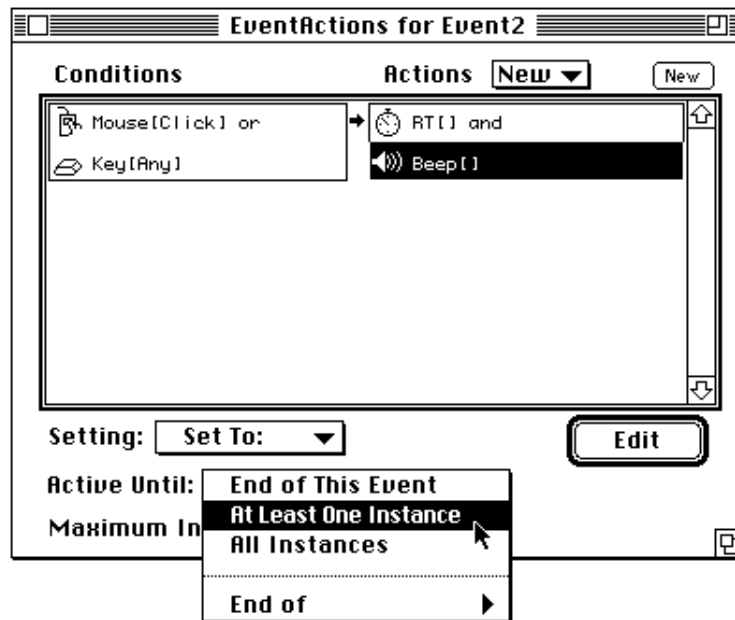


Figure 40 – Setting Active Until

Now when you run the trial, it will last until you press a key or click the mouse.

Try removing Mouse[Click] as a condition for the RT[] and Beep[] actions (open the Conditions dialog and uncheck the **Mouse** line). Now try running the trial and clicking the mouse during Event2. Event2 will end, but PsyScope will wait (that is, the trial will continue) until you press a key. This is because the duration of Event2 is still Mouse[Click], but this no longer triggers the actions; since they are now set to remain active until they are performed at least once, the trial will continue until they are triggered — that is, until you press a key.

Now try keeping an action active until the end of another event:

1. Go to the Trial Template window, add a new **Time** event, and make sure that it is linked to the end of Event2.
2. Set its duration to be 1000 msec.

- Open the Actions dialog for Event2, and set **Active Until** for the RT[] and Beep[] actions to be **End of Event3**.

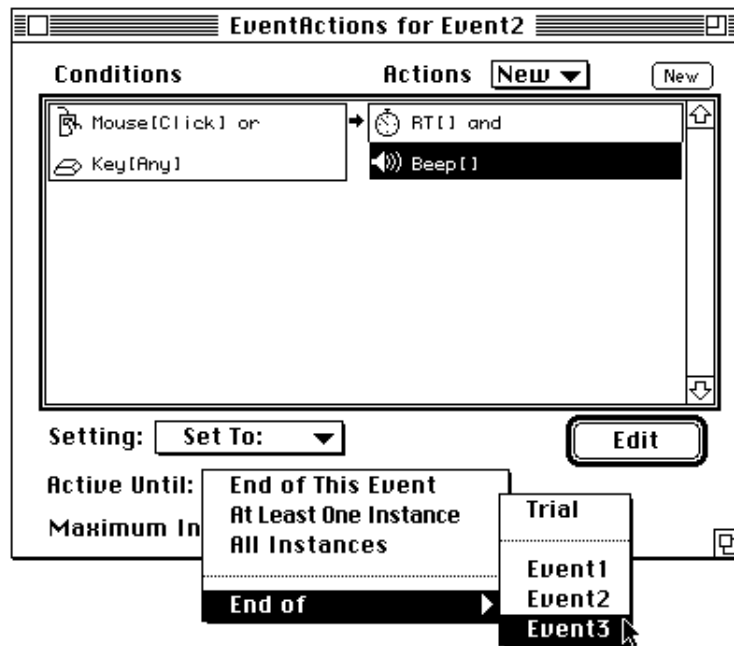


Figure 41 – Setting Active Until relative to another event

Now you will have until the end of Event3 to press a key or click the mouse. Once Event3 ends, its too late. Note that once Event3 ends, the trial also ends.

Maximum Instances determines how many times an action can be performed. Once it has been performed this number of times, it is immediately inactivated, regardless of what **Active Until** is set to. Usually, **Maximum Instances** is set to 1, which means that it can only be performed once. If this value is greater than 1, then the action will be performed every time its associated condition(s) occur(s) until it has been performed its maximum number of times, or until its active period is over.

Modify your script to measure how many times you can press a key in one second:

- Remove Event3.
- Set the duration of Event2 to be 1000 msec.
- Set **Active Until** to be **End of This Event**.
- Place the cursor in the **Maximum Instances** box by either pressing the tab key or clicking the mouse in it, and then enter a large number, such as 100 or 200.

Run the trial, and as soon as Event2 begins, starting pressing a key as furiously as you can. When the trial ends, open the data file, go to the end of it, and count how many lines are in the data area (consider using the **Statistics** utility; see “Part 2: Graphic Environment Reference, 7.3.1 Editor Menu Items”, p258). This is how many times you pressed the key; one

line for each key press equals one triggering of the action. If the number is greater than 60, something has gone wrong.

Events, attributes, and actions give you a powerful set of tools for constructing trials in PsyScope. Until now, however, the trials you have worked with were fixed. That is, once you made a change to the trial, every time you ran it, it was identical. Needless to say, most experiments require that you vary at least the stimuli in some way from trial to trial. There are a number of ways to accomplish this in PsyScope. The most important is the use of *factors*.

3.2.8 Using Factor Tables and Lists

In a PsyScope experiment, as in general, a *factor* is a variable that is relevant to the scientific question you are attempting to answer with the experiment. For example, you may be interested in how the size of a stimulus affects subjects' response times. It would be impractical to test all possible sizes, so you would choose a few— say three — and test subjects' response times to stimuli presented in these three sizes. In this case, the size of the stimulus would be a factor in the experiment, and it would have three *levels*, corresponding to the three sizes you are testing. You may also want to vary other aspects of the stimulus, such as its distance from the fixation point (at the center of the screen). This would be another factor in the experiment. When you have two or more factors, then these factors are *crossed* to determine how the combinations of their levels are sampled from trial to trial (e.g., large and to the left, small and in the middle, etc.).

There are two ways that you can create factors in PsyScope. The easiest, and most common way is to create a *factor table*. Factor tables are displayed in the Factor Table window. You can think of a factor table as an interactive table that shows you all of the factors and levels that you have created, shows you the way in which they are crossed, and allows you to edit the factors, levels, and crossings. Factor tables also give you a convenient way to set the values associated with the levels of each factor.

The other way to create factors is through the List dialog, which is used with *lists*. This is a somewhat more flexible, but more advanced technique. In this section, we will show you how to use factor tables. Creating lists is discussed in “3.2.8.4 Lists”, p59.

To demonstrate the use of factor tables, let's switch to a more realistic example of an experiment. We will recreate the Acuity Experiment that you worked with in “Chapter 2. Running Your First Experiment”, p3.

Create a new experiment by choosing the **New Experiment** command in the **Design** menu, and calling it “Acuity Experiment”. If the Design window doesn't open automatically, go ahead and open it (see above, “3.2.1 Using the Design Window”, p11, for a description of how to do this).

3.2.8.1 Creating a Factor Table

As you saw earlier, when you create a new experiment, PsyScope starts you out with the experiment object. We then went on to create a trial template, that defined the trials in the experiment. When you use a factor table, however, the trial template object is replaced by the factor table object, and you design the trials via the table.

The basic idea is this: After you create a factor table object, you open the Factor Table window by double-clicking on the object. In the Factor Table window, you create the factors for your experiment, and the levels for each factor. These are represented as a table, where the rows and columns correspond to the different levels of the factors, and each cell in the table corresponds to a particular combination of levels — or *crossing* — of the factors. Let's first see how this works, and then go on to designing the actual trials of the experiment.

First, create a new factor table by doing the following:

1. Click on the factor table icon in the tools palette (see “Figure 2 – Factor table icon”, p4), then click in the Design window, a factor table object should appear, named Table1
2. Drag the link from Table1 to the Experiment

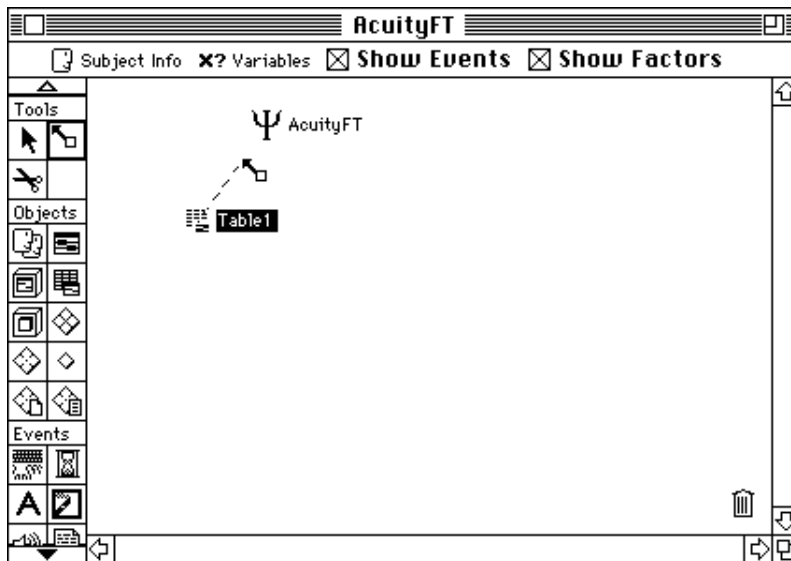


Figure 42 – Linking a factor table to the experiment

3. Double-click on Table1 to open the Factor Table window.
4. Create two factors by clicking on the **New Factor** button, and name them “Size” and “Position”, respectively.

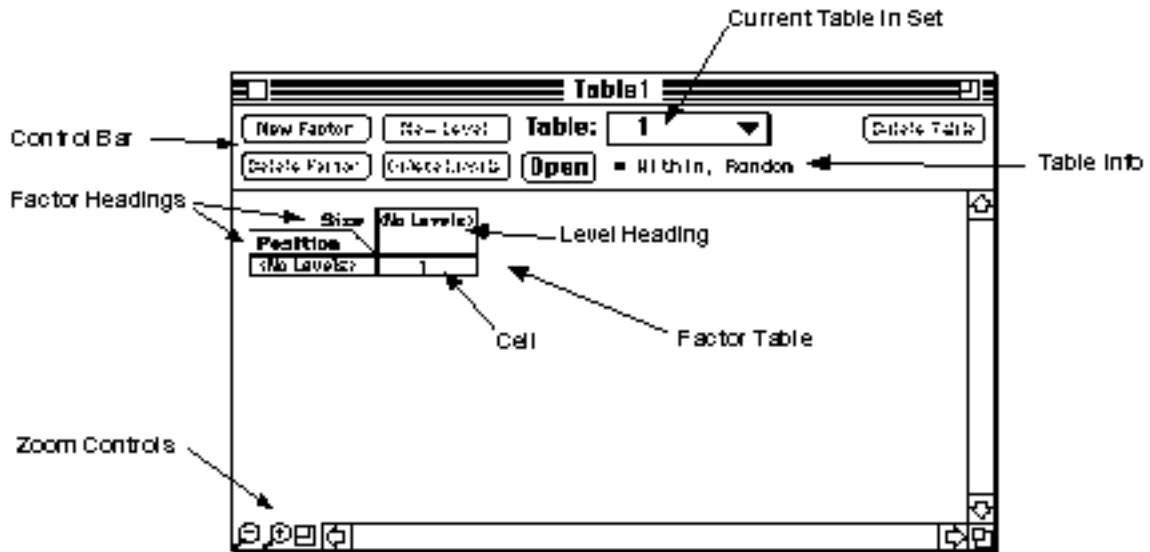


Figure 43 – Anatomy of the Factor Table window

Note: Each factor table object can actually contain a set of closely related factor tables. This is useful when you are working with between subject designs, which is discussed below (see “Groups and Between Subject Designs”, p89). However, the vast majority of the time you will have only one table in the set. To keep things simple, the name of the object is a “Factor Table” rather than a “Factor Table Set”.

Now you need to add levels to the two factors. Start with the Size factor:

- Select the Size factor by clicking on its name.

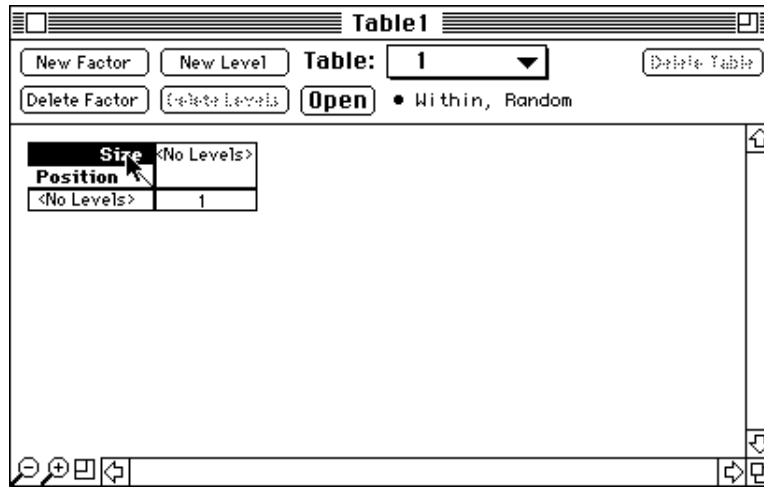


Figure 44 – Selecting a factor in a factor table

- Add a new level to the Size factor either by clicking on the **New Level** button, or typing Command-N. Name the level “Small”.
- Add another new level and call it “Middle”. Before you close the New Level dialog, notice the button called **Another**. You can use this to add additional levels, without having to go back to the Factor Table window.

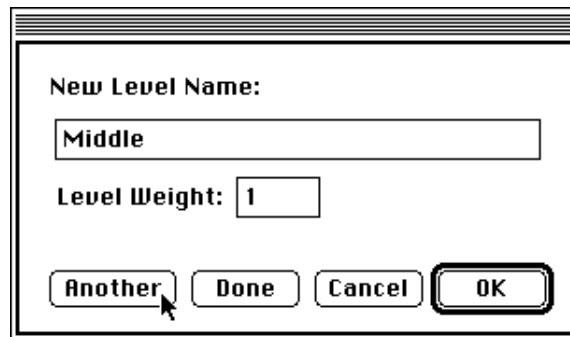


Figure 45 – The New Level dialog

- Click on the **Another** button, or type Command-A, and name the third level “Large”. This is all the levels you will need, so now click on the **OK** button or press the Return key.

The Factor Table window should now look like this:

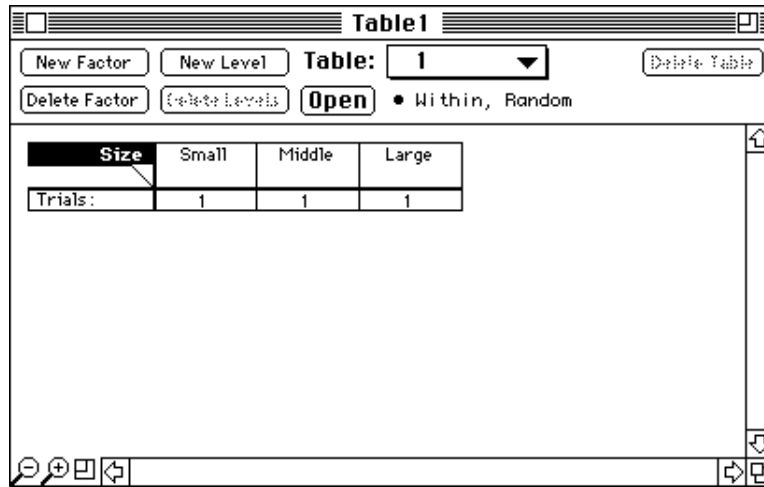


Figure 46 – Factor Table window with new levels

Using the same procedures as above, select the Position factor, and create five levels for it, called “LeftFar”, “LeftNear”, “Center”, “RightNear”, and “RightFar”. The Factor Table window should now look like this:

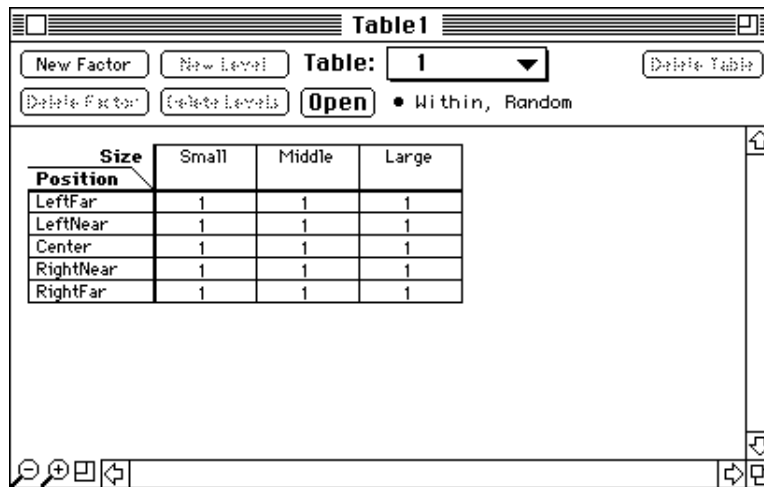


Figure 47 – Factor table window with levels for all factors

You have now created a factor table. Each cell in this table corresponds to a crossing of the two factors in the experiment. For example, the cell in the upper left corresponds to the combination Small and LeftFar, and the one at the extreme lower right to Large and RightFar. At the beginning of each trial, PsyScope chooses one cell, or crossing, and runs the trial according to that crossing. The number in each cell indicates how many times that cell will be chosen before PsyScope cycles through all of the cells again. You can set this by setting the weight for the corresponding cells. This will be discussed below (see “ Cell Weights”, p54). You can reorder the factors and levels in a factor table to customize the display and to influence the order in which cells are chosen when the experiment is run. The

order in which PsyScope chooses cells will also be discussed below (see “Ordering Cells”, p50).

Note: There is a close relationship between the cells of a factor table and the conditions of an experiment. In general, these are related, but not identical items. When there is only one factor table in an experiment, then each cell corresponds to a condition of the experiment. However, when an experiment has more than one factor table (and/or set of free factors), then each condition of the experiment corresponds to a particular combination of cells from the factor tables (and/or crossings of the free factor sets). The relationship between conditions, cells, and crossings is discussed fully in “Part 2: Graphic Environment Reference, 5.7 Factors and Lists”, p129.

As we have seen, factors and levels determine how aspects of a trial vary from one cell to the next. The next step, therefore, is to create the trials for each cell, and assign the values that the factors and levels in the table actually correspond to in those trials.

3.2.8.2 Creating Trials from a Factor Table

You create the trials for the cells in a factor table by “opening” them and constructing the trial in a Trial Template window, similar to the standard one that you have already worked with. When you double-click on a cell or group of cells in a factor table, a Trial Template window opens that allows you to create the trials for that set of cells. However, this Trial Template window is special (as indicated by the diamond (◊) at the left of the window title) in that it is linked to the factor table; anything you do in this window applies only to trials associated with the cells that are currently selected in the factor table. So, for properties of a trial that are common to a set of cells, you would select those cells and then assign the properties to the trial in the Trial Template window. For properties that differ, you would select the only the appropriate subset of cells in the table, and set the properties in the Trial Template window accordingly. At this point, an example will surely be instructive.

First, however, it will be helpful to know some of the navigation and selection tricks for working with the table and cells.

The factor table you have already created has two factors: Size and Position. Now we have to create the trials and events that these correspond to. The best place to begin is by constructing the elements that are common to all of the trials in the experiment. In the Acuity Experiment, every trial has four events: the fixation event, which displays the fixation point, and waits for the subject to click the mouse to continue; the stimulus event, which presents the stimulus briefly on the screen; the RT period during which the subject responds; and the intertrial interval (ITI). Let’s begin by creating these:

1. Select all of the cells in the table, and then either click on the **Open** button, or double-click on any of the cells. This will open a Trial Template window, which has a diamond (◊) at the left of its title.
2. Create the four events in the order listed above. The fixation and stimulus events should be **Text** events, the RT period should be an **Input** event, and the ITI should be a **Time** event. You may want to reduce the scale of the event area to see all of the events; do this by clicking on the zoom icon in the lower left corner of the window.

When you are done, the Trial Template window should look something like this:

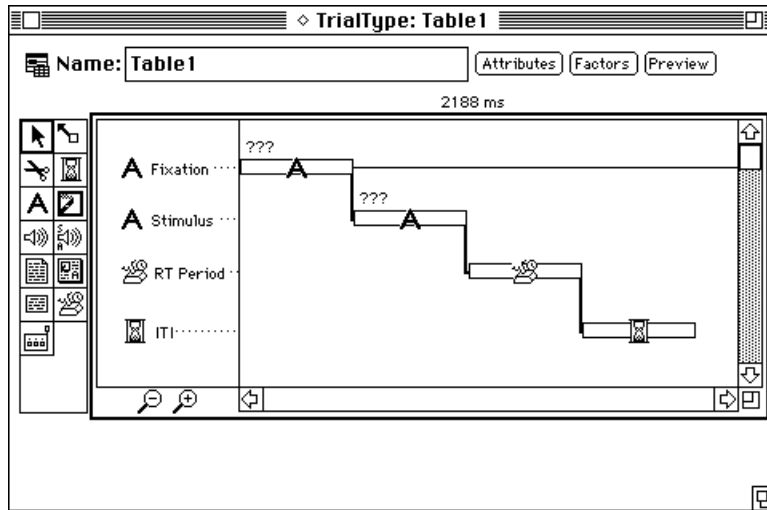


Figure 48 – Template window with more events

Now that you have the events in place, set the values of the attributes that will not vary from trial to trial. (Remember, since all of the cells in the factor table are still selected, the event structure that you have created — and any attribute values that you set — will apply to all trials.) First, set the duration of each event:

- Set the duration of the Fixation event to be `Key[2]`.
- Set the duration of the Stimulus event to be 100 msecs.
- Set the duration of the RT Period event to be 1500 msecs.
- Set the duration of the ITI event to be 500 msecs.

The fixation point will be the same for all trials, so go ahead and set the stimulus for that event:

- Open the Attributes window for the Fixation event, and set the **Stimulus** attribute to be an asterisk (*) or a plus sign (+).

The Trial Template window should now look something like this:

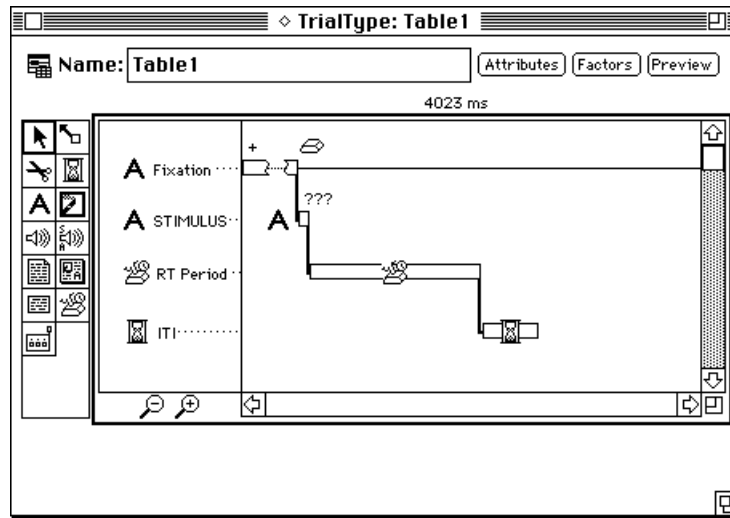


Figure 49 – Template window with more parameters set

At this point, you have the basic structure of the trials in the experiment in place. Now it is time to set the attribute values that vary from trial to trial according to the conditions of the experiment (i.e., according to the factors and levels you defined earlier). Let's begin with the Size factor.

1. Make the Factor Table window the active window by clicking on it or by choosing it in the **Window** menu.
2. Select the cells for the Small level of the Size factor by (single-)clicking on its heading at the top of the column:

Size	Small	Middle	Large
Position			
LeftFar	1	1	1
LeftNear	1	1	1
Center	1	1	1
RightNear	1	1	1
RightFar	1	1	1

Figure 50 – Selecting the cells for a level

3. Return to the Trial Template window, either by choosing the window name from the **Windows** menu, or by clicking on it directly. You could also double-click on the cells selected in the table, or click on the **Open** button.

If you position the Trial Template window so that you can still see the table, you will notice that the selected cells in the factor table are grayed:

	Size	
Position	Small	Middle
	LeftFar	
LeftNear		
Center		
RightNear		
RightFar		

Figure 51 – Cells are grayed when the window is in the background

Any changes that you make in the Trial Template window or an attribute window apply only to trials associated with the cells that are currently selected in the Factor Table window. This allows you to set the values associated with each level of a factor.

Set the size of the stimulus for each level of the Size factor, starting with Small, which you already have selected:

1. Open the Attributes window for the Stimulus event, set its size to 9, and then click on **OK**. Now, for all trials run when the level of the Size factor is Small, the size of the stimulus will be 9 point.
2. Leave the Attributes window for the Stimulus event open. Make the Factor Table window active, and select Middle. Now return to the Attributes window. Notice that the value for size has changed back to “Default”. Set the value to 12
3. Follow the same steps to set the size for the Large level to 18.

When you select a new set of cells in the Factor Table window, PsyScope updates the Trial Template and Attribute windows (if they are open), to reflect the values corresponding to the newly selected cells (sometimes it may take one or two seconds for this to occur).

Try this out:

- Make the Factor Table window active, and select Small. Notice that the value in the Attributes window in the background changes to 9. Select Middle, and it should change to 12.

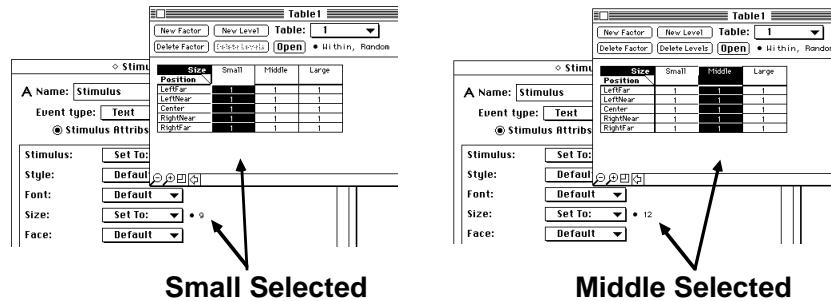


Figure 52 – Attribute values depend on which cells are selected

You can also use the Cell Chooser palette to select cells. The Cell Chooser is a palette window that has a copy of the factor table in it. Because it is a palette, it “floats” in front of other windows — that is, it will always be available whenever a template window or attribute dialog is selected (when any other type of window or dialog is selected, it will be hidden). You can use the Cell Chooser to select cells, as well as to change to another table in the set, but you cannot double-click on cells in the palette to open the template window.

Open the Cell Chooser by selecting it in the **Window** menu, and use it to set the values for the levels of the Position factor:

1. Click on LeftFar, at the left of the first row in the table:

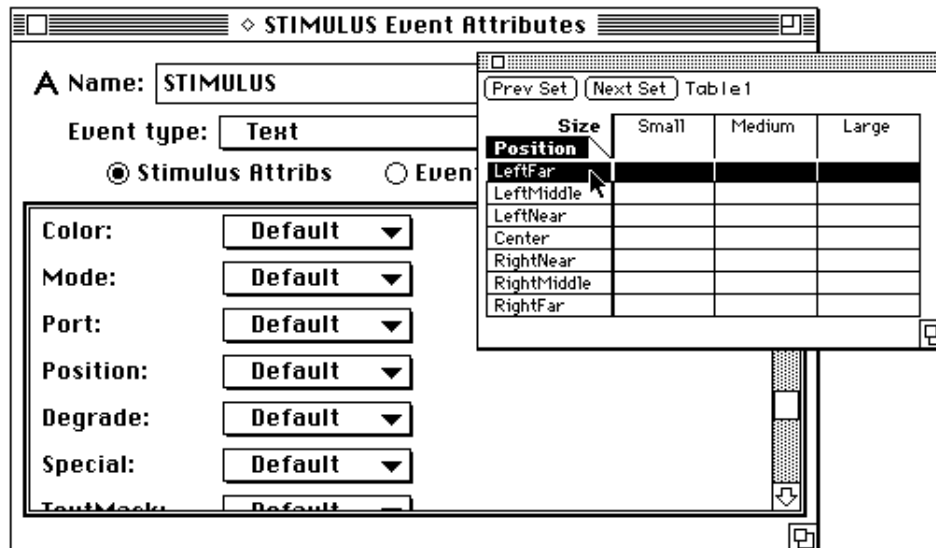


Figure 53 – Attribute window with the Cell Chooser open

2. Select the **Position** attribute in the Event Attributes dialog (notice that the Cell Chooser palette remains active and in front of the attributes dialog).

3. Select **Set To** in the **Position** attribute menu. This will clear the screen and open the Positions dialog:

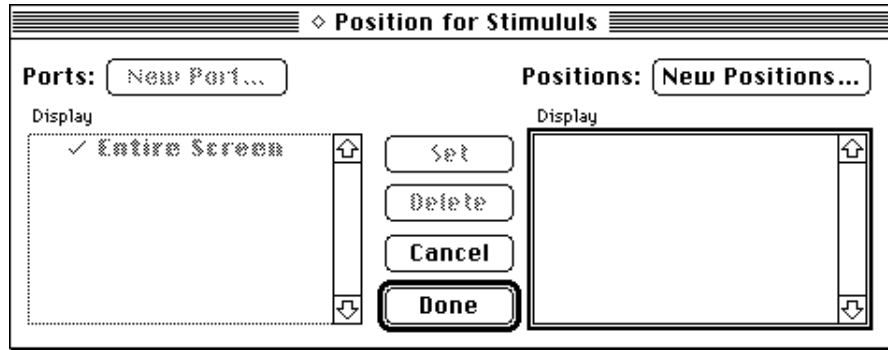


Figure 54 – The positions dialog

The Positions dialog allows you to create ports and to define positions within them for placing stimuli on the screen. The screen clears so that you can work interactively with these elements as you create them. For now, however, we will just use this dialog to create positions for each of the locations at which we want the stimulus to appear (see “Part 2: Graphic Environment Reference, 5.8.8.4 Ports and Positions Dialogs”, p187 for a complete explanation of ports and positions and a full description of the Positions dialog).

4. Click on the **New Positions** button. This will open the New Position Dialog.

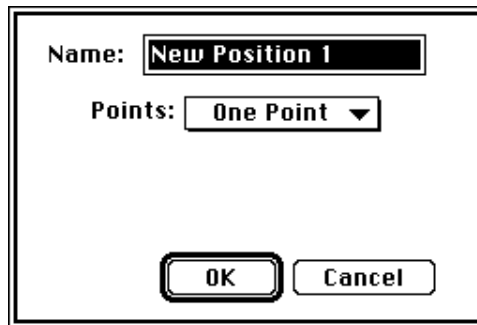


Figure 55 – New Position dialog

- Click on **OK** to create a single position. A position named “New Position 1” will appear at the center of the screen, and in the list of positions at the right of the Positions dialog. You move the New Position 1 by directly dragging it to a new location or by using the Position Info dialog (which is what we will do).

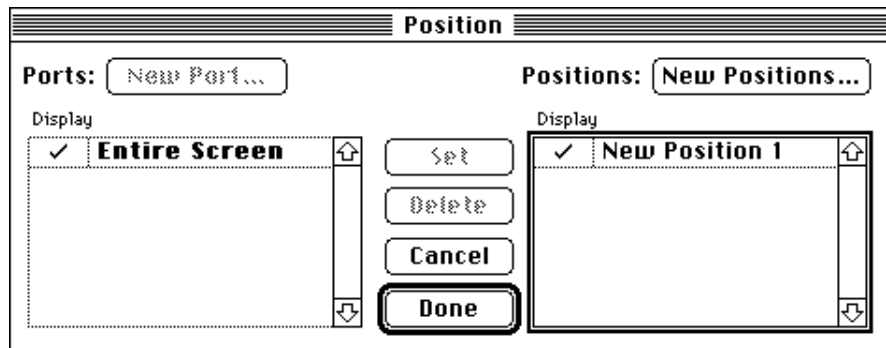


Figure 56 – Positions dialog with a new point

- Double-click on New Position 1 to open the Position Info dialog, and rename the position to “LeftFar.” Then enter “10%” in the **Horizontal** field and “50%” in the **Vertical** field (be sure to include the “%” sign, so that Psy-Scope will know to use this as a relative rather than absolute position value). When you are done, click on **OK** to close this dialog.

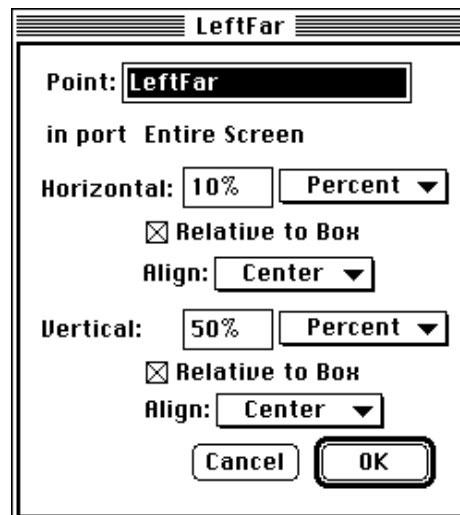


Figure 57 – Position Info dialog

- Be sure that LeftFar is selected in the list of positions in the Positions dialog, and then click on the **Set** button to its left. This sets the value of the Position attribute to this position. LeftFar should now appear in bold to indicate this.

8. Click on the **Done** button, which will return you to the Event attributes dialog. Notice that the value of the Position attribute is set:

Position: Set To: ▼ • 10% Port Center 50% Port Center

Figure 58 – Position attribute showing new port specification

At this point you can repeat the steps above, each time selecting a different level of the Position factor in the Cell Chooser, creating an actual position that corresponds to that level, and assigning it as the value for the **Position** attribute. A slightly quicker method is to return to the Positions dialog, create all four of the remaining positions you need, and then assign them to each level of the Position factor. Let's do it that way:

9. Reopen the Positions dialog by clicking on the value of attribute.
10. Repeat steps 4 and 5 above to create 4 more points, named "LeftNear", "Center", "RightNear", and "RightFar". Assign a **Vertical** value of 50% and **Horizontal** values of 30%, 50%, 70%, respectively. The positions dialog should now look like this:

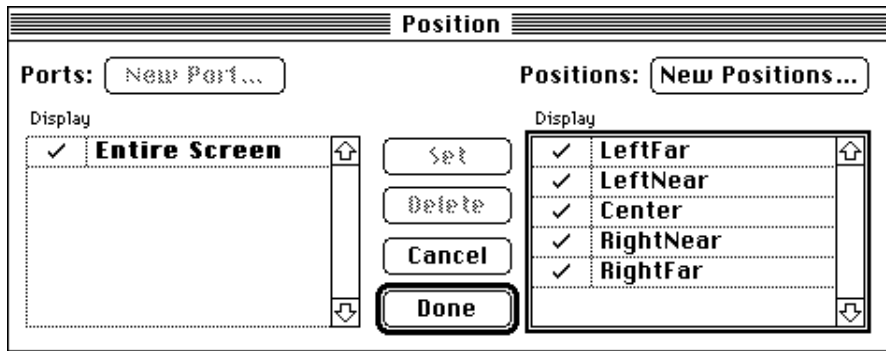


Figure 59 – Positions dialog with five points

11. Click on **Done**, to return to the Event Attributes dialog, and then click on **LeftNear** in the Cell Chooser. Wait a couple of seconds; the value of the Position attribute will change to **Default**.
12. Select **Set To**, and in the Positions dialog select **LeftNear** in the list of points. Then click on **Set** and then **Done**. The value of the **Position** attribute should now be set to the position of the "LeftNear" point — 20% horizontal and 50% vertical.
13. Repeat steps 9 and 10 to set the value for the other levels of the Position factor.

When you are done, try clicking on individual cells in the table, or in the Cell Chooser, and notice that both the size and position values will change in the Event Attributes window for Stimulus, corresponding to the levels of each factor for that cell.

At this point, the only thing that you need to do before you can run some trials is to specify the text to be displayed during the Stimulus event. For now, let's just set this to be the same for all trials, and see how things are working.

- Select all of the cells (a shortcut for this in the factor table window is CMD-A), and then set the stimulus attribute for the Stimulus event to be “GLEEP”, or any other word that you like.

Try running a trial. First you should see the fixation character appear; then PsyScope should wait for you to press the “2” key, following which you should see the stimulus word — in 9 point — flash at the far left side of the screen.

Now try running several trials, by doing the following:

1. Open the Experiment dialog by double clicking on the experiment icon in the Design window.
2. Enter the number of trials you want to run in the **Cycles** field at the bottom of the dialog
3. Choose **Run** from the **Run** menu, or type CMD-R

(Other methods for controlling the number of trials to run are discussed below, under “3.2.12 Running Trials”, p92, and in “Part 2: Graphic Environment Reference, 6.3 The Trial Monitor”, p238.)

Run at least 4 or 5 trials. Notice that from trial to trial, the cells of the factor table are sampled randomly, but that every cell is sampled once before any is repeated.

By default, PsyScope selects cells from a factor table in random order as it constructs each trial to run. The next section will show you how you can control this order.

3.2.8.3 Controlling How Cells are Chosen

Unless you specify otherwise, PsyScope will run one trial from each cell in the factor table in random order. Once all of the cells have been run, PsyScope will start again at the beginning, picking cells in a new random order. However, there are a number of ways you can modify the order and number of times that cells in a table are chosen. You can do this table-wide, or for subsets of cells. In this section we consider some of the simpler methods. You will find a full discussion of how to control cell selection in “Part 2: Graphic Environment Reference, 5.7.1.7 Level Order and Crossing Types”, p132.

You control the manner in which cells are chosen on a table-wide basis from the Table Info dialog.

Open the Table Info dialog by clicking on the Table Info area in the control bar of the Factor Table window:

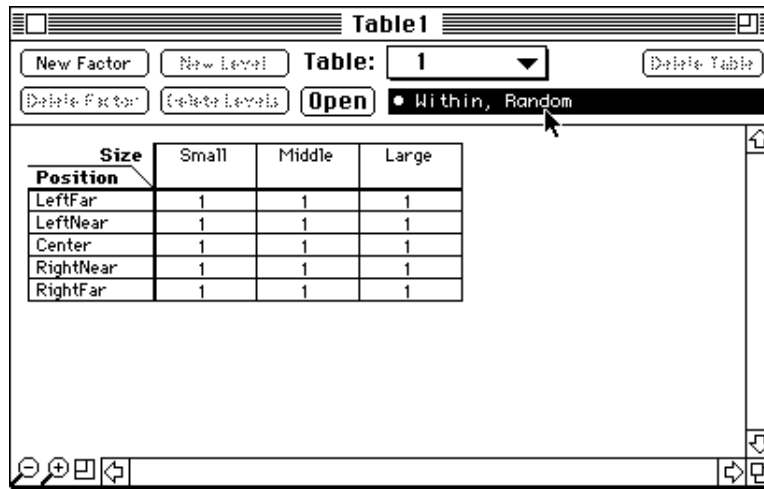


Figure 60 – Opening the Table Info dialog

The Table Info dialog will appear:

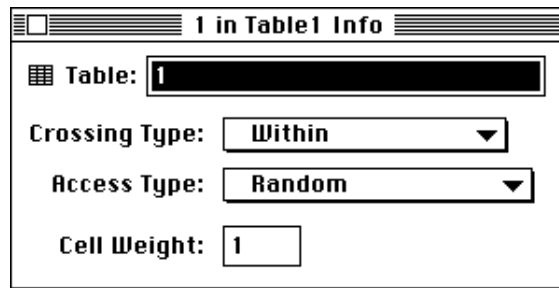


Figure 61 – The Table Info dialog

The Table Info dialog allows you to change the name of the table you are working with, set the *crossing type* and *access type*, or set the base weight for all of the cells in the table. These settings provide control over how cells in the table are chosen.

Ordering Cells

There are a number of ways you can influence the order in which cells are chosen: by selecting the **Access Type** for the table, by rearranging levels and factors in the table, and by setting the **Level Order** value for individual factors. Below, we consider the first two of these. Working with individual factors is a more sophisticated method, which is discussed fully in “Part 2: Graphic Environment Reference, 5.7.1.2 Lists”, p130.

Setting Access Type for the Table

You can use the **Access Type** menu in the Table Info dialog to select the method by which cells are chosen from the table. There are several choices:

Random (the default)
Cycle Random
Random with Replacement
Blocked Random
Least-Used Random
Sequential
Blocked Sequential
By Factor

All but the last choice set the order by which cells are chosen on a table-wide basis. The last choice, **By Factor**, allows you to control the order on a factor-by-factor basis. This method is significantly more sophisticated, and is discussed in “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140.

By default, **Access Type** is set to **Random**. With this method, PsyScope chooses cells from the table randomly. Each cell is chosen a number of times equal to its weight (see below, “Cell Weights”, p54), after which PsyScope begins the selection again.

Setting **Access Type** to **Sequential** causes cells to be picked in columnar order, from top to bottom and then left to right.

Select **Sequential** from the **Access Type** menu, and then run some trials. They should occur in the following order:

	Size		
Position	Small	Middle	Large
LeftFar	1	6	11
LeftNear	2	7	12
Center	3	8	13
RightNear	4	9	14
RightFar	5	10	15

Figure 62 – Sequential ordering for cells in a table; see note below

Note: the numbers in this figure correspond to trial order, not to cell weights (as they do in the actual factor table).

Rearranging Levels

Notice that when **Access Type** is set to **Sequential**, the order of factors and levels in the table determines the order in which cells are chosen. You can modify this order by rearranging the levels and factors.

Try reversing the order in which the levels of the Size factor are arranged:

1. Select the Large level by clicking on its heading in the table.
2. Drag it to the position currently occupied by the Small level.
3. Select the Small level and drag it to the right end of the table.

Note: Whenever you drag a level to a position occupied by another level, it is placed just beyond that level.

The factor table should now look like this:

Size	Large	Middle	Small
Position			
LeftFar	1	1	1
LeftNear	1	1	1
Center	1	1	1
RightNear	1	1	1
RightFar	1	1	1

Figure 63 – Factor table with levels rearranged

Try running the experiment. All of the Large trials should now run first, followed by Middle, and then Small.

Rearranging Factors

Another way of influencing the order in which cells are chosen is by rearranging factors. This can also be useful for customizing their display in the factor table. Let's first look at how factors can be rearranged, and then consider — in some simple cases — how this influences the ordering of cells.

Change the table so that both of the factors are displayed as columns:

1. Select the Position factor heading, and drag it up and to the right, over the diagonal line that separates the two names.

The table should now look like this:

Position	LeftFarr			LeftNear		
	Large	Middle	Small	Large	Middle	Sma
Trials:	1	1	1	1	1	1

Figure 64 – Factor table with factors rearranged

Notice that the cells no longer all fit within the window. You can scroll the window, to see cells that extend beyond it, or you can use the size tool in the lower left-hand corner of the window to reduce the size of the table. Click on the zoom icon (see below) to the right of the magnification icons to return the table to its original size.

Figure 65 –

More importantly, notice that the columns for the Size levels are now subdivided within the columns for the Position levels. That is, there are now Small, Middle and Large sub-columns of LeftFar, and of LeftNear, etc.

Whenever there is more than one factor in a table, then there is an ordering of factors from *inner* to *outer*. Inner factors can be thought of as being subdivided within outer ones. The table displays this relationship explicitly for the factors in each dimension, by showing inner row factors subdivided within outer row factors, and similarly for factors in columns. In addition, all of the factors in rows are considered to be within those in columns. In sequential designs, PsyScope cycles through all of the cells corresponding to the levels of a inner factor (for example, all of the rows within a column, or all of the subdivided columns within an enclosing one), keeping the level of outer factors constant, before moving on to the next level of the next outer factor, etc. Having spelled this out, it is best understood through experimentation.

Position is currently the outer factor, while Size is the inner one. Try reversing this arrangement by dragging Size back above Position, or Position below Size. Also try placing them both as column factors. Run the experiment with the different arrangements to see the effects that each has.

Cell Weights

In the examples we have seen so far, each cell was chosen once per pass through the table. However, you can modify this by changing *cell weights*. The weight of a cell — the number that appears inside of it in the table — determines the number of times PsyScope will choose that cell on each pass through the table. For example, if there are four cells in your table, and each cell has a weight of two, then PsyScope will choose each cell twice before going through the table again. Cells will not necessarily be chosen twice consecutively; as we will see, this depends on how **Access Type** is set.

You can modify the weight of all of the cells in the table at once, by setting the **Cell Weight** value in the Table Info dialog. You can also adjust the weights of cells in specific rows or columns, by modifying the weight of individual levels in the table.

Setting the Cell Weight

The **Cell Weight** value in the Table Info dialog multiplies the weight of all of the cells in the table. You set it from the Table Info dialog.

Go to the Table Info dialog, and enter 2 in the **Cell Weight** field, just below the **Access Type** menu. The weight of all of the cells in the table should now be “2”, and the table should look something like this:

	Size	Small	Middle	Large
Position				
LeftFar		2	2	2
LeftNear		2	2	2
Center		2	2	2
RightNear		2	2	2
RightFar		2	2	2

Figure 66 – Factor Table with Cell Weight of 2

Set the **Access Type** menu back to **Random**, and run at least 30 trials; otherwise, trust us: Each cell will have run twice before any were repeated again. However cells did not necessarily run twice in a row.

The total number of trials that make up a complete pass through the table is equal to the sum of the weights of all of the cells in the table. In other words, this is how many trials you need to run the complete design.

Note: There is no way to have PsyScope automatically run the current number of trials to exhaust the current design.

The process of picking cells works something like this: At the beginning of each trial, PsyScope picks a cell from the table, and gives it a check mark. When the number of check marks for a cell is equal to its weight, it is crossed out. When all of the cells are crossed out, the pass is over, PsyScope clears the boards, and begins the process over again from the

beginning. Cell weights have different effects, depending upon how **Access Type** is set for the table.

When **Access Type** is set to **Random**, PsyScope selects cells and checks them off in random order, so that even if the weight of a cell is greater than one, it will not necessarily be chosen consecutively. If you want to run cells consecutively, you need to change **Access Type** either to **Blocked Random** or **Sequential**.

With **Blocked Random**, PsyScope picks a cell randomly from the table, and then continues to pick that cell until it has been “crossed out”; i.e., until it has been picked a number of times equal to its weight. PsyScope then chooses another cell randomly from the table, and continues with that one until *it* has been crossed out, etc. **Sequential** operates in a similar way, except that when PsyScope is finished with one cell, it moves to the next one in columnar order (see “3.2.8.3 Controlling How Cells are Chosen”, p49).

With **Random**, **Blocked Random**, and **Sequential**, every cell in the table is assured of being chosen the number of times specified by its weight. **Random with Replacement** works differently. In this case, the weight of each cell determines the *probability* with which that cell will be chosen. It is possible that some cells will never be chosen, while others may be chosen more often than their weight.

*Note: The exact probability with which a cell is chosen — with **Random with Replacement** — is the cell’s weight divided by the sum of all of the weights in the table.*

Weights are particularly useful when you want to assign different frequencies or probabilities to different sets of cells. To do this, however, you need to assign different weights to each set of cells. You do this by setting level weights.

Setting Level Weights

Using *level weights*, you can modify the weights of cells in a specific row or column. You do this by changing the weight of the level to which those cells belong, using the Level Rename dialog.

Open the dialog for the Large level of the Size factor, by double-clicking on its name in the factor table.

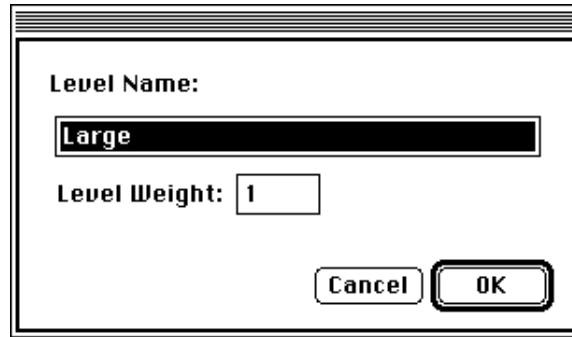


Figure 67 – The Level Rename/Set Weight dialog

Change its weight to 3. Notice that the weight all of the cells in the column under Large is now 6. This is the cell weight (2, set in the Table Info dialog) times weight of the Large level (3). The table should now look like this:

	Size	Small	Middle	Large
Position				
LeftFar		2	2	6
LeftNear		2	2	6
Center		2	2	6
RightNear		2	2	6
RightFar		2	2	6

Figure 68 – Factor table with weight 3 added to Large level

Now try changing the weight for one of the levels of the Position factor. For example, change the weight of the Center level to 2. The table should now look like this:

	Size	Small	Middle	Large
Position				
LeftFar		2	2	6
LeftNear		2	2	6
Center		4	4	12
RightNear		2	2	6
RightFar		2	2	6

Figure 69 – Factor table with weight 2 added to Center level

The weight of a cell is the product of the **Cell Weight** value in the Table Info dialog and the weights of the levels that the cell belongs to.

Setting level weights is useful if you want to alter the frequency with which a particular set of cells occurs in your experiment (for example, you want stimuli to appear more often at the center of the screen than to the sides). Set the **Access Type** of the table to **Random** or **Sequential**, and then the weight of each level to the relative frequency with which you

want those cells to occur. Remember when **Access Type** is set to **Random with Replacement**, weights influence only the *probability* with which cells are chosen: the higher the weight, the higher the probability it will be chosen.

Using Crossing Type and Factor Table Sets

In addition to setting **Access Type** and cell weights, there is another important way by which you can influence how cells are chosen. This is using the **Crossing Type** menu in the Table Info dialog. This determines how the factors in the table are crossed. Usually you will want this to be **Within**, the default setting. In this case, cells are run as described in the sections above. That is, all of the cells in the table will be run in each pass (except, as noted above, when the **Access Type** is **Random with Replacement**).

You can limit the set of cells that are picked each time the experiment is run by choosing one of the other settings in the **Crossing Type** menu. The two other primary choices — **Between** and **Latin Squares** — cause PsyScope to pick only one or a subset of cells for each run of the experiment. These are typically used for between-subject designs, when each subject is to be exposed to only a subset of conditions. Below, we will consider a couple of simple cases of between-subject designs. For a more complete discussion of **Between** and **Latin Squares** designs, see “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140.

Between-subjects Designs

If you set the **Crossing Type** of a factor table to **Between**, then PsyScope will choose one cell from the table, and run all of the trials in the experiment from that one cell. The trials in the next run of the experiment will all be from the next cell of the table, etc. In other words, instead of choosing cells from trial to trial, PsyScope chooses them from run to run. This is called a full *between-subjects* design (because typically each run of the experiment is performed by a different subject). However, it is rare that you will want to do things in this way. More frequently, you will want to vary only certain sets of factors between subjects, while you vary the rest within.

Suppose, for example, that you wanted to vary the font of the stimulus from subject to subject, but you still wanted to vary the size and position from trial to trial, as before. In other words, you want to add Font as a between-subjects factor, while Size and Position remain as within-subjects factors. To do this, you need to add a new factor table, and assign it a **Crossing Type** of **Between**.

Create the new factor table for the Font factor by doing the following:

1. Choose **New** from **Table** menu in the control area of the Factor Table window. A dialog asking you to name the new table will appear.
2. Name the table and click on **OK**.
3. Create the Font factor by clicking on the **New Factor** button in the Factor Table window, and naming the factor in the dialog that appears.
4. Create three levels for this factor, and name them “Times”, “Helvetica”, and “Monaco”.

5. Assign the font of the stimulus for each level by opening its cell, then opening the Event Attributes dialog for the Stimulus event, and setting the font accordingly.
6. Open the Table Info dialog, and choose **Between** from the **Crossing Type** menu. The Choose Crossing dialog will appear, with a menu allowing you to select how cells will be picked.

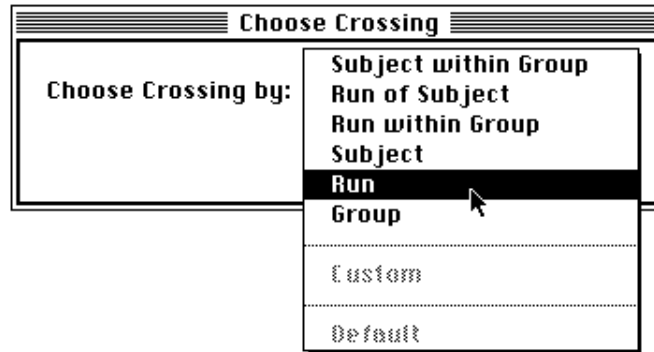


Figure 70 – The Choose Crossing dialog

7. Choose **Run** from the menu. This will cause PsyScope to access a new level of the Font factor each time the experiment is run. That is, this part of the design is “between by run.”

*Note: There are six pre-defined types of “between” designs in PsyScope: group, subject, run and subject within group, run within group, and run within subject. Each of these uses a different variable (CurrentGroup, SubjectCount, RunCount, SubjectNumber, GroupRunCount and RunNumber, respectively) to choose the levels of **Between** factors. In addition, a **Custom** setting allows you to use other variables in the script, or to define and use new variables. See “Part 2: Graphic Environment Reference, 6.2.1 Subject Info Items”, p225 for a description of the pre-defined variables, how to use others, and how to define your own. See “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140 for a more complete description of **Between** designs.*

Try running the experiment several times. Each time you run, a different font should be chosen.

Access type, cell weight, and level weights are ignored in **Between** designs. This is because all trials within a run are being selected from a single cell in that table.

Notice that the template that appeared when you opened cells for the Font factor was the same one you created while working with the Size and Position factors. This is because these factors all belong to tables in the same *factor table set*. All of the tables in a set refer to the same template.

Note: Whenever you pick one or more cells from one table, it is as if you selected all of the cells from all of the other tables in the set. Therefore, any changes you make to the template from a cell (or cells) in one table influences all of the cells of all of the other tables in the set.

By using the various features of factor tables — access type, cell weight, level weights, and the various crossing types — you can create highly sophisticated experimental designs. For a complete description of all of the features associated with factors in PsyScope, see “Part 2: Graphic Environment Reference, 5.7 Factors and Lists”, p129.

3.2.8.4 Lists

Lists, like factors, provide a means for varying parts of a template from trial to trial. The most common use of lists is to vary some aspect of the trial independently of the factors in a table. For example, you may want to randomly choose the stimulus for each trial. Let's add this feature to the Acuity Experiment, which is the last addition we have to make for the experiment to be complete.

First, we must create the list of stimuli to choose from:

1. Go to the Factor Table window, and select all of the cells in the table. Double-click on the table (or click the **Open** button) to edit the master template.
2. Open the Event Attributes dialog for the Stimulus event.
3. Select **Vary By** and **List** from the **Stimulus** attribute menu. The following message dialog will appear:



Figure 71 – Create New List alert

- Click on the **Create** button, to create a new list. The List dialog will appear:

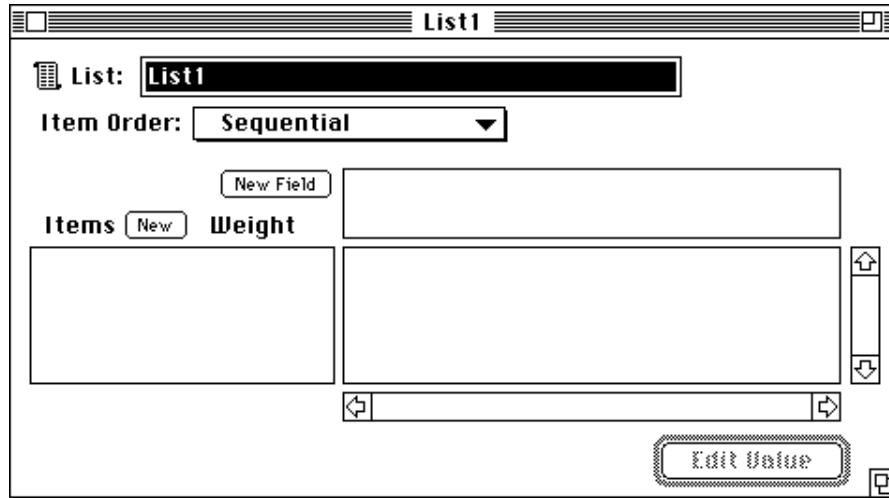


Figure 72 – The List dialog

- Name the list “Stimulus List”.

Once you have created a new list, it will appear as a List object in you experiment. Confirm this by selecting the design window; Stimulus List should appear as a list object linked to the factor table. If it is not there, try checking the **Show Factors** box in the control area.

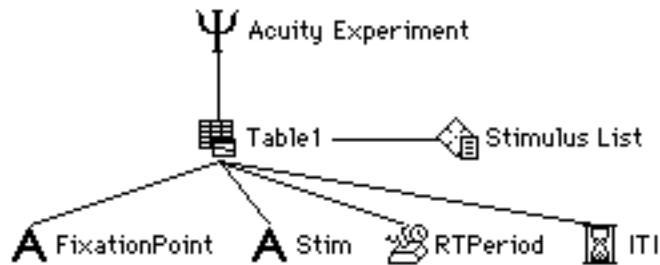


Figure 73 – Experiment hierarchy showing a list connected to the table

Each list is made up of a set of *items*, in much the same way that a table is made up of cells. One difference between lists and tables, however, is that a new item is chosen from the list only if the list is *accessed* in that trial, that is, if some attribute in the trial is linked to the list. If no attribute in the trial is linked to the list, then no new item is chosen. This is in contrast to tables, where a cell is chosen at the beginning of every trial. You can change this feature of a list by changing its crossing type (see “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140). For now, however, we will consider lists only in their native state.

The **Item Order** menu for a list functions like the **Access Type** menu for a table, controlling the order in which items are chosen from the list. The **Weight** of an item is also similar

to the weight of a cell: it determines the number of times that each item will be chosen each pass through the list.

You assign the value of each item in the list using the area to the right of the list of items and their weights.

Add a new item to the list:

1. Click on the **New** button in between the **Items** and **Weight** headings. An item named Item1 will appear, with “???” in the value field.
2. Double-click on the question marks, or select them and then press return. A standard text dialog will appear,
3. Enter a word that you want to use as a stimulus, and then close the dialog. You can press Return to close the dialog.

Tip: You can quickly create a list of items and assign their values as you go, by first selecting the list fields in the dialog, and then repeating the following sequence of steps:

1. Press Command-N (creates a new list item).
2. Press Return or Enter (opens the value dialog).
3. Enter the value.
4. Press Return or Enter (closes the value dialog).
5. Press Command-N (to create the next item in the list).

Create several more stimuli. Then set the **Item Order** menu to **Random**, so that a stimulus will be chosen randomly from trial to trial.

You can also try setting the weight of one or more items, by clicking on the value in the column under the **Weight** heading (or typing Command-Right arrow). A text area will appear, in which you can enter the weight for that item.

Items <input type="button" value="New"/>	Weight	Stimulus
Item 1	1	DOG
Item 2	<input type="text" value="1"/>	HOUSE
Item 3	1	TABLE

Figure 74 – Setting the weight for an item in the list

When you are done, close the list dialog. Note that the Event Attributes window now shows that the value of the **Stimulus** attribute is being set by the list:

Stimulus: • "Stimulus List"

Figure 75 – Attribute status shows vary by the new list

Try running the experiment. The stimulus word should now vary randomly from trial to trial.

You can have as many lists as you like in an experiment, which can be used to vary the stimulus or any other attribute of an event (e.g. the size, font, or color of a stimulus). Whenever you create a new list, PsyScope figures out what type of attribute it is for, and creates a list for that type. You can also link more than one event to the same list, as long the attributes you are linking are all of the same type (for example, they are all size or all font attributes).

Note: PsyScope will actually allow you to link incompatible attributes to a list, however unexpected results may occur. Usually the need for this arises when you want several different attributes of an event or a trial all to vary together, in parallel. To do this, you should use fields within the list. See “Part 2: Graphic Environment Reference, 5.7.1.2 Lists”, p130.

Just for fun (well, also to show you a couple of additional things about working with lists), let's create another list, which we will use to vary the type face of the stimulus.

1. Go back to the Event Attributes dialog for the Stimulus event, and choose **Vary By** and **List** in the **Face** attribute menu. Because you have already created a list, you will get the Vary By List dialog, instead of the message dialog you got earlier.
2. Select the **List** pop-up menu. Notice that the Stimulus List is an item on the menu.

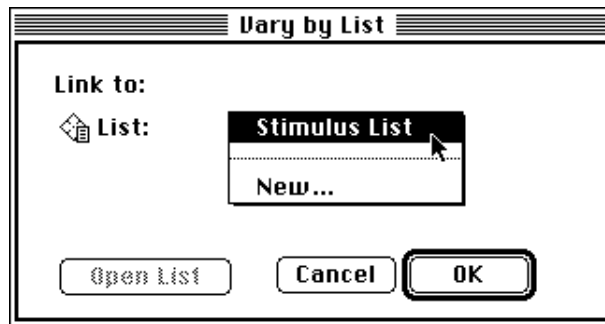


Figure 76 – The Vary by List dialog

3. Choose **New...** from the menu. You will get the List dialog you saw earlier, only now the heading for the value field will be **Face**, indicating that the items in this list are faces rather than stimuli.

Note: The heading of the value field in the List dialog indicates the type of event and type of attribute that items in list are compatible with.

4. Create a few items and assign them each a different face (bold, italics, etc.).

Run the experiment. Now both the word and the face of the stimulus should vary from trial to trial.

Lists, like factors, provide a means for varying features of the experiment from trial to trial. Lists provide a convenient method for doing so independently of the factors in a table. The list dialog also provides a convenient format for entering the values you want the feature to assume, rather than having to go through the table cell-by-cell.

3.2.8.5 Counterbalancing

Stimulus *counterbalancing* is a common experiment design requirement. In a counterbalanced design, each subject sees every stimulus in only one condition; but, across subjects, every stimulus is seen in every condition.

Usually, experiment conditions are implemented in a factor table and stimuli are implemented in a list. To counterbalance the stimuli against the conditions, you must:

- import the list into the factor table
- use the **Latin Squares** crossing type

When we created a list earlier, it was automatically put into its own factor set, independent of the factor table. This meant that a stimulus could be selected randomly, regardless of the current cell in the factor table. Now, however, we want the stimulus selection to depend on the factor table cell selection, because we want each stimulus to appear in only one condition. To create this dependency, we have to import the list into the factor table.

A full discussion of **Latin Squares** is reserved for “Part 2: Graphic Environment Reference, Latin Squares”, p133. In brief, **Latin Squares** uses only a “diagonal” of the cells from a table, and the diagonal that is used depends on the **Choose Crossing** index.

The Size and Position factors will need to be placed into the same *Latin square partition*, and Stimulus List will need to be in a separate partition. Factors which are in the same partition are crossed fully, but only a subset of the crossing is used for factors in separate partitions.

To counterbalance Stimulus List against the Position and Size factors, we need as many stimuli as we have cells in the table. Then, we have to relink the list to the factor table, choosing to put it into the table. Then, we will change the **Crossing Type** and set the Latin square partitions.

1. Double-click on the Stimulus List icon in the Design window to open the List dialog.
2. Add a few more items to the list, as described in “3.2.8.4 Lists”, p59, so that there are fifteen items.
3. Using the scissors tool in the Design window, cut the link between Stimulus List and Table1. A message dialog may appear, warning you that some attributes use the list that you have disconnected; if it does, hit **Ignore**, because we are going to reconnect the list.

- Use the link tool in the Design window to reconnect Stimulus List and Table1. The Connect List dialog will appear (see figure below).

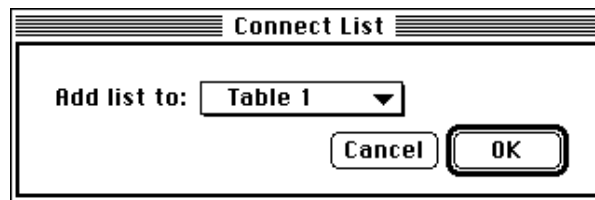


Figure 77 – The Connect List dialog

- This dialog is used to specify how the list should be connected to the factor table object. In this case, we want to add Stimulus list into the same table as Size and Position, so choose **Table 1** from the pop-up menu and hit **OK**. (“1” is the table within the Table1 object which contains Size and Position).

Now, you can open the Factor Table window for Table1, and you will see that Stimulus List has been added to the table (see figure below). Since Stimulus List is imported, however, its items are not shown like the levels of a regular factor; instead, they are all collapsed into “<All Levels>”.

		Size		
Position	Stimulu..	Small	Medium	Large
LeftFar	<All Levels>	1	1	1
LeftNear	<All Levels>	1	1	1
Center	<All Levels>	1	1	1
RightNear	<All Levels>	1	1	1
RightFar	<All Levels>	1	1	1

Figure 78 – Factor table with an imported list

- Go to the Table Info dialog and choose **Latin Squares** in the **Crossing Type** menu. This will open the Latin Squares dialog. Drag the list items around so that Size and Position are together and Stimulus List is separate, as shown in the figure below.

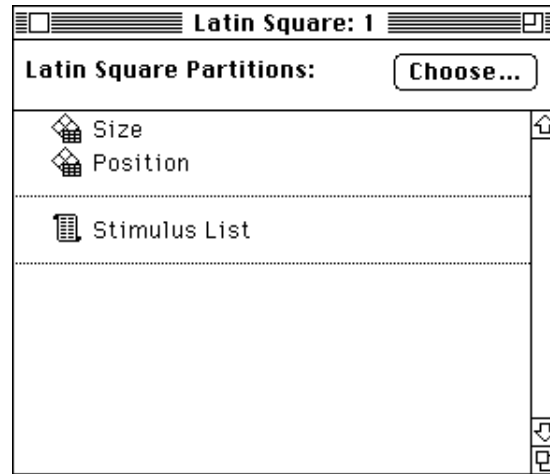


Figure 79 – The Latin Squares dialog

- Hit the **Choose...** button in the Latin Squares dialog. This will open the Choose Crossing dialog which was introduced in “Between-subjects Designs”, p57. Just as then, choose **Run** from the pop-up menu. This will cause PsyScope to use a different counterbalancing each time the experiment is run.

With counterbalancing set up on Stimulus List, you should see each item in the list in only one position and size for any single run of the experiment. However, over fifteen runs of the experiment, you should see every stimulus in every combination of size and position (assuming that you run enough trials to see every stimulus every time).

3.2.9 Using Blocks

Often, you will want to group trials into discrete sets for presentation when the experiment is run. For example, you may want to present some instructions, followed by one set of trials, and then different instructions followed by another set of trials. This process is called *blocking*, and there are several ways to accomplish it in PsyScope. The simplest and clearest way to do this is to use blocks.

A **block** is set of trials that will be presented together when the experiment is run. Each block can contain one or more types of trials: PsyScope will present all of them together, before presenting the trials of the next block. There are two types of blocks, that have slightly different icons. A regular block (see “Figure 80 – Block icon”) is made up of trials. A **superblock** (see “Figure 81 – Superblock icon”) is made up of other blocks. In this section,

we will focus on regular blocks. Superblocks are described in “Part 2: Graphic Environment Reference, 5.5.2 Block Dialog”, p121.



Figure 80 – Block icon



Figure 81 – Superblock icon

3.2.9.1 Blocking Trials

As an example of how you might use blocks, let's imagine that you want to demonstrate that the effects in the Acuity Experiment are due to attention. One way to do this is to compare subjects' performance when they maintain central fixation with performance in a control condition in which they look directly at the stimulus. To do this, you need to divide the trials into two separate blocks, and include a set of instructions before each. The easiest way to do this is to create a block for each set of experimental trials, as well as one for each set of instructions (you can think of these as special trials).

Add four blocks to the Acuity Experiment, by doing the following:

1. Select the regular block icon in the Objects palette of the Design window, and place it in the work area. The cursor will change to the link tool, unless the automatic tool-switching option has been turned off (See “Part 2: Graphic Environment Reference, 7.6.4 Design Options”, p268). You can always select the link tool from the palette at the left of the Design window.
2. Link the block to the Experiment icon. Notice that Table1 automatically gets re-linked to the block.

Note: Only objects of the same type can be linked to a given object. You will get an error message if you try to link objects of incompatible types to the same object. See “Part 2: Graphic Environment Reference, 5.2.1.1 Linking Objects”, p108 for more information about linking objects in the experiment hierarchy.

3. Create three more blocks, linking each one to the Experiment.
4. Rename Block3 and Block4 “Instructions 1” and “Instructions 2”, respectively

In the experiment we have planned, the trials in the central-fixation vs. direct-look blocks will be identical, therefore we can use the same trial template for both. You can do this by linking Table1 (which contains the template for those trials) to both Block1 and Block2:

- Select the link tool in the Tools palette, and link Table1 to Block2, so that it is linked to both Block1 and Block2.

The Design window should now look something like this:

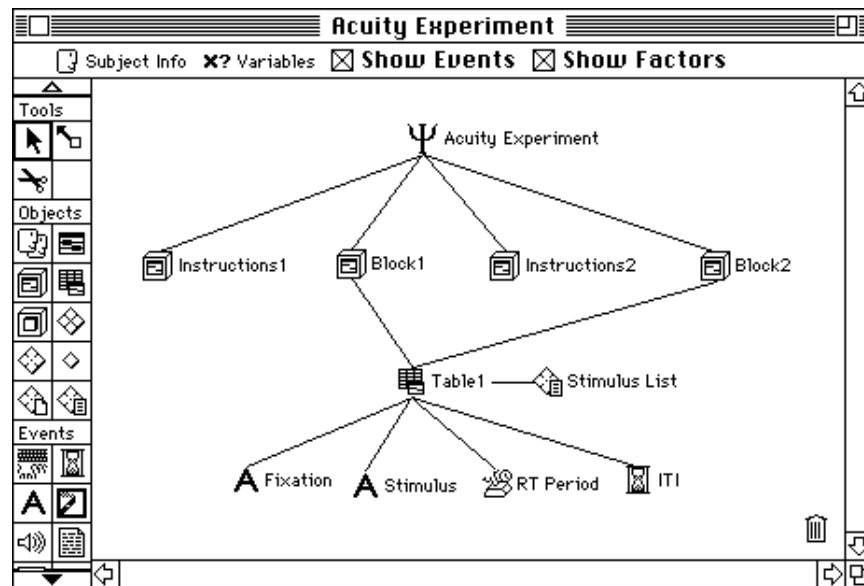


Figure 82 – Acuity experiment using blocked trials

Now let's create the template for the instructions trials:

1. Create a new template in the work area and name it “Instructions Trial.”
2. Open the Template window, and create two events. Make one a **Time** event, and name it “ITI”. Make the other a **Paragraph** event (see below), and name it “Instructions.”



Figure 83 – The **Paragraph** icon

3. Set the duration of the Instructions event to `Mouse[Click]`. (See “3.2.5.1 Timing an Event”, p19.)

Paragraph events allow you to present text that includes line breaks, tabs, etc.; this is more convenient than **Text** for presenting something that includes several lines of text, such as instructions.

Now, we want to present a different set of instructions before each of the blocks of experiment trials. The most straight forward way to do this is to duplicate the Instructions Trial template, and assign a different set of instructions to the paragraph event in each:

1. Go to the Design window, select the Instructions Trial template, and then choose **Duplicate** from the **Edit** menu, or type Command-D. A new template will appear, called “Instructions Trial copy.” This will have copies of all of the events in the Instructions Trial template. At this point, it may be helpful to reduce the clutter in the Design window, by unchecking the **Show Events** and **Show Factors** checkboxes in the Control area.

3. Rename the two instructions templates as “Instructions Trial 1” and “Instructions Trial 2”
2. Open the Template window for each of the instructions trials in turn, choose **Set To** for the paragraph (stimulus) attribute, and type in the instructions for that set of trials.
3. Return to the Design window, and link the template for each instructions trial to the corresponding block.

The Design window should now look something like this:

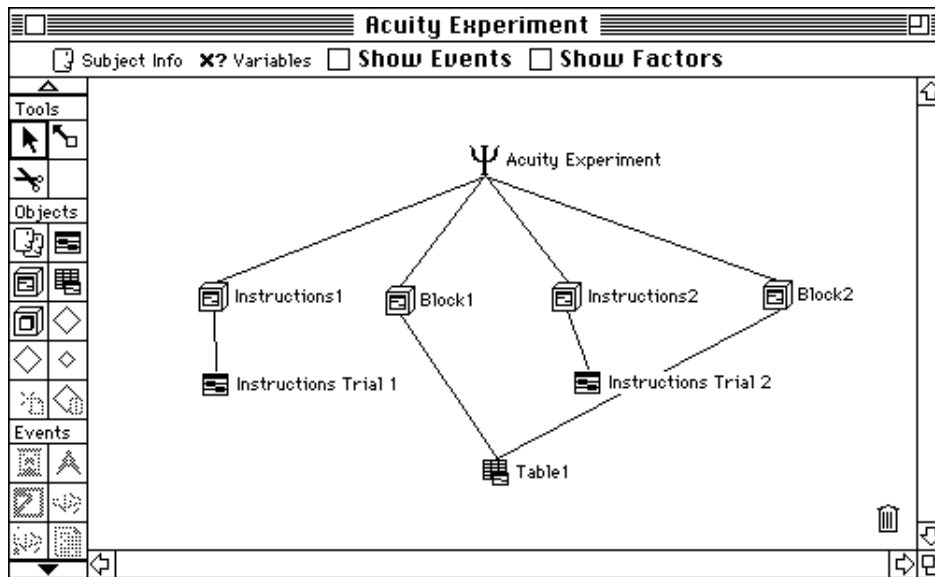


Figure 84 – Experiment hierarchy with instructions trials

*Note: There is another way that you can present different instructions in each of the instructions blocks, using a single instructions template. This involves the use of custom block attributes and the **Vary By Block** setting for the stimulus attributes. We will describe this more advanced technique shortly (See “3.2.9.2 Varying by Block”, p72).*

Now that you have created a set of blocks, you must order them, and specify how many trials will be presented in each,

You set the number of trials presented in a block, or the amount of time that you want the block to run, using the Block dialog.

Open the Block dialog for Instructions1 by double-clicking on it in the Design window.

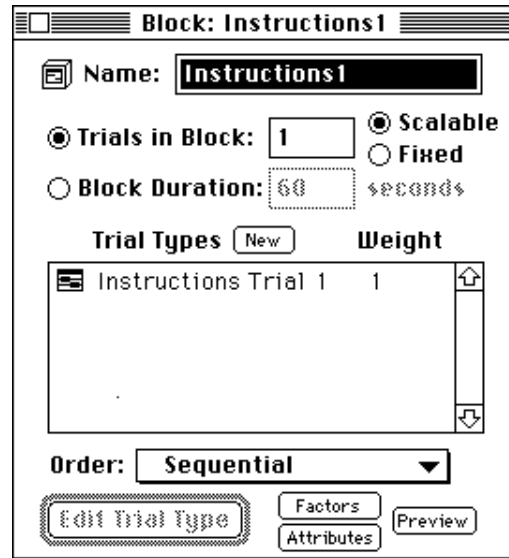


Figure 85 – The Block dialog

At the top of the Block dialog is the usual name field. Below it are radio buttons and value fields for specifying how many trials to run in the block, or to time its duration.

Set the Instructions1 block to run a single trial by doing the following:

- Click on the **Trials in Block** button (to specify a number of trials rather than a timed duration), and enter 1 as the value. Also, click on the **Fixed** radio button. This insures that only one trial will run in this block, even if the block-scaling feature is used by an object above it in the hierarchy (see below).

Do the same for the Instructions2 block. Then set the **Trials in Block** or **Block Duration** for Block1 and Block2. For example, just to try things out, set Block1 to run 10 trials, and Block2 to run 15 seconds.

Below the controls for the block's length is a list of templates that are linked to the block. You can re-order them, assigning them different weights, and create new templates that are linked to the block. However, because each block in the Acuity Experiment has only one type of trial, you will not need to bother with these features.

When you have several blocks in an experiment, you must set the order in which you want the blocks to be run, and the number of times you want to run each one. You do this in the dialog for the object just above the blocks in the hierarchy. In the Acuity Experiment, this is the Experiment dialog.

Open the Experiment dialog by double-clicking on the experiment icon.

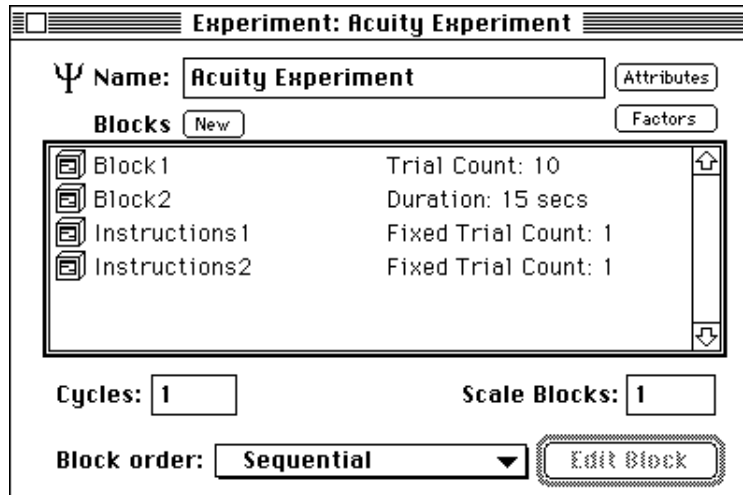


Figure 86 – The Experiment dialog

Note: Which dialog you use for ordering blocks depends upon the structure of your experiment. In the Acuity Experiment, the blocks are linked directly to the experiment, so the Experiment dialog controls how the blocks are run. However, if they were linked to a superblock, then you would use the superblock's dialog; and if they were linked to a group, then you would use the group's dialog. In general, the dialog that contains the list of blocks is called the Block List dialog.

The dialog shows a list of the blocks linked to the experiment. When the dialog is first opened, the blocks are listed in the order in which they were created. If the **Block order** menu (at the bottom of the dialog) is set to **Sequential** (as it is by default), this is also the order in which they will be run. If this is not the order in which you want them to run, you will need to reorder them in the list, so that they run in the correct order.

Reorder the blocks by doing the following:

1. Select Instructions1 by clicking on it once.
2. Drag it to the top of the list.

3. Select and drag Instructions2 to a position just in between Block1 and Block2. This may be a bit tricky. If you don't get it just right, it will end up above Block1 or below Block2. You can try again, or you can just play Chinese-checkers with the blocks until you have them in the correct order.

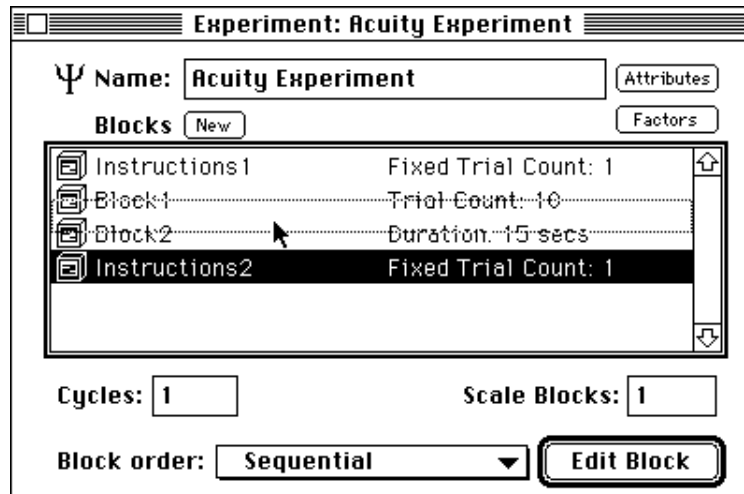


Figure 87 – Reordering a block

Notice that a **Trial Count** or **Duration** value is listed next to each block in the list. These correspond to the number of trials or duration (in seconds) that you previously set for each block (in the Block dialogs). Double-clicking on this value (or anywhere on that line) will open the dialog for that block, allowing you to modify the value.

The fields below the list of blocks provide additional control over how the blocks in the list are run. The **Cycles** value controls how many times PsyScope will run through the list of blocks. **Scale Blocks** influences the number of trials that are run in each block that has **Scalable** selected; this value is multiplied by the **Trials in Block** value to arrive at the total number of trials to run in that block. The **Scale Blocks** value has no effect on blocks that have **Fixed** selected, or that have a time duration specified (see the “3.2.12.4 The Trial Monitor”, p95 for an example of scaling blocks). Finally, the **Block order** menu controls whether the blocks are run in sequential or random order.

Note: A full discussion of trial counting is in “Part 2: Graphic Environment Reference, 5.12.2 Trial Counting”, p211.

In this section we showed you how to use blocks to block trials in PsyScope. There are also other methods for accomplishing this. For example, you can block trials by using a separate template for each block (this is a simpler, but more limited method). You can also use factors to block trials using certain access types (see “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140).

3.2.9.2 Varying by Block

In addition to using blocks for blocking trials, you can also use them to vary attributes of the trials in each block. For example, instead of using two separate templates for the instructions before each experimental block, you can use the same template for both instructions blocks, and vary the actual text of the instructions by block.

Return to the Design window, and follow these steps:

1. Select one of the Instruction Trial templates — let's say the Instructions Trial 2 — and drag it to the trash or choose **Clear** from the **Edit** menu. You might also want to empty the trash (**Design** menu), as there is not reason to keep the extra template around.
2. Link Instructions Trial 1 to the Instructions2 block.

The Design window should now look something like this:

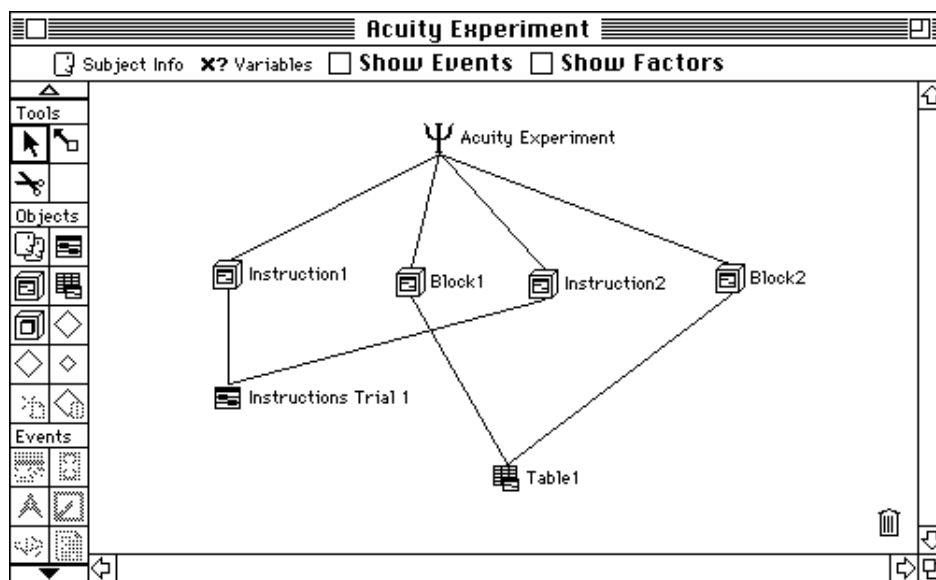


Figure 88 – Using the same template for both kinds of instructions

Now, open the Block Attributes dialog for either of the two instruction blocks — let's say Instructions1 — using one of the following two methods:

- Open the Block dialog by double-clicking on the block, and then open the Block Attributes dialog by clicking on the **Attributes** button.

or

- Hold the Control key down while you select the block icon with the mouse. Choose **Edit Block Attrs** from the menu that appears.

Tip: When you are in the Design window, you can open a menu that will provide direct access to the Attributes dialog for an object by holding the control key down while selecting the object with the mouse.

The Block Attributes dialog should now be open (Instructions1 Attributes). Create a new custom block attribute, and assign its value for each block as follows:

1. Choose **Custom Block Attribs** from the **Attribute Set** menu, if it is not already selected, and then click on the **New** button. This will open the **New Attribute** dialog.
2. Name the attribute “Instructions”.
3. Select **Stimulus Attribs** from the **Attribute Set** menu, and select **Paragraph** from the **Stimulus Type** menu.

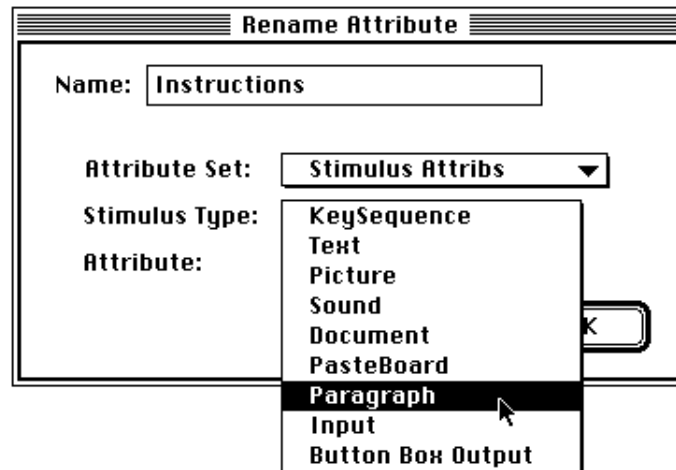


Figure 89 – The Create/Rename/Retype Attribute dialog (used here to create a custom attribute)

When you are done, click on the **OK** button. This will return you to the Block Attributes dialog, and **Instructions** should appear as a custom block attribute.

4. Set the instructions for this block by choosing **Set To** from the menu for the Instructions attribute, and entering the instructions in the text dialog that appears.

When you are done, the Block Attributes dialog should look like this:

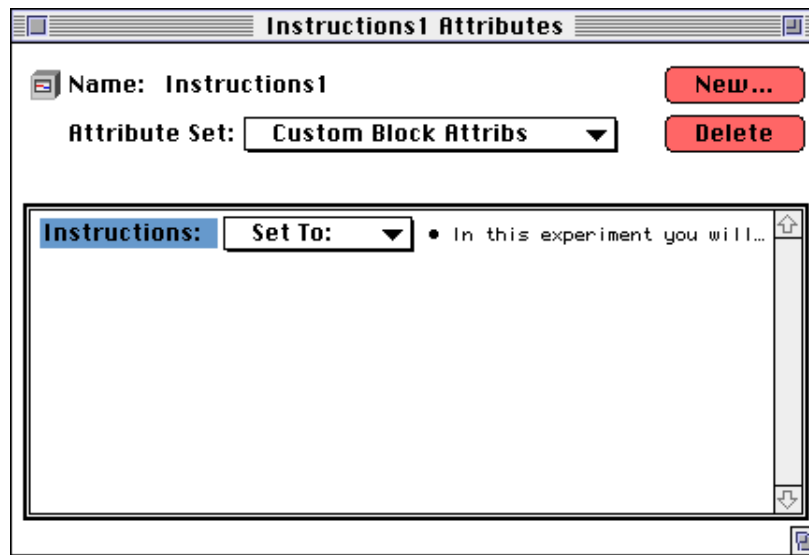


Figure 90 – Block attributes dialog showing a custom attribute

Once you have defined Instructions as a custom block attribute, it will be there for all blocks. Therefore, all you have to do for the Instructions2 block is open its Block Attributes dialog, and enter the instructions for that block as the value of the Instructions attribute.

*Note: Whenever you create a custom attribute for an object of a particular type, PsyScope automatically adds it to all objects of that type in the experiment, and sets their value to **Default**.*

You can use **Vary by Block** to vary the value of any trial, event or stimulus attributes in your experiment. For example, you might want to vary the font of the stimulus in each of the two experimental blocks. This provides a convenient method of blocking features of your design, without having to create a new template for each block. You can also vary the value of attributes according to the group that the subject is in. Of course, this requires that you have defined subject groups, which is the topic of the next section.

3.2.10 Using Subject Info and Groups

When conducting a real experiment, you will want to record information about the subjects you run in the experiment. For some designs, you may also want to vary how the experiment is run based on which group the subject belongs to, or to keep track of the number of subjects run in each group. In this section, we will show you how to use PsyScope to record information about each subject, and how to set up and use groups.

3.2.10.1 The Subject Info Dialog

Let's begin by setting things up to record the Subject's name, age and handedness. You do this using the Subject Info dialog.

Open the Subject Info dialog by clicking on the Subject Info button (see below) in the control area of the Design window. Notice that three items are already defined.

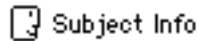


Figure 91 – Subject Info icon button

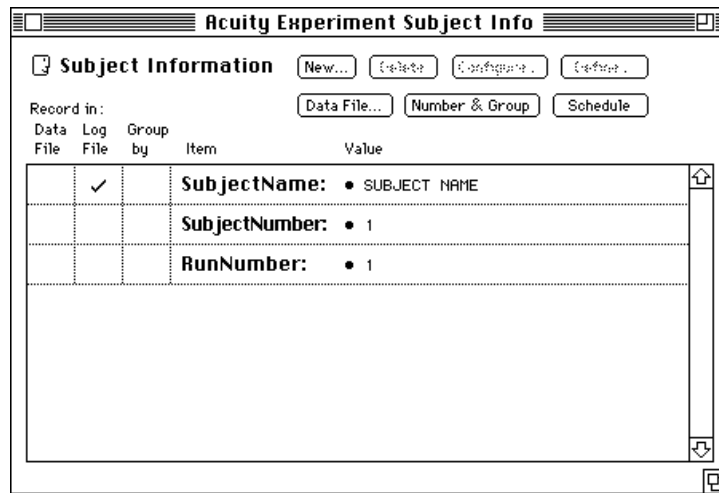


Figure 92 – The Subject Info dialog with the default items

The Subject Info dialog contains a list of the information items that will be recorded for each subject. PsyScope automatically creates three items for you, which must always be present: “SubjectName”, “SubjectNumber”, and “RunNumber”. SubjectNumber and RunNumber are values that PsyScope calculates automatically for each run of each subject in an experiment.

SubjectNumber and RunNumber

PsyScope assigns a SubjectNumber to each subject run in each experiment. Numbers are assigned sequentially, starting with 1. Each subject will have the same SubjectNumber for a given experiment, even if they are run multiple times in that experiment (assuming the name is spelled exactly the same and the group assignment does not change). However, the same subject may have different SubjectNumbers for different experiments. If you are using groups, PsyScope maintains a separate series of SubjectNumbers for the subjects in each group (see “Groups and Between Subject Designs”, p89).

PsyScope also calculates a RunNumber each time an experiment is run, which is the number of times that particular subject has been run in that particular experiment.

See “6.2.3 Subject Number Calculation”, p230 for a complete description of how PsyScope computes the values of SubjectNumber and RunNumber.

Adding New Subject Info Items

You can use the Subject Info dialog to create additional items, specify when and if PsyScope will prompt you to enter information for them, and when PsyScope will record the information.

Add items for the subject’s age and handedness, by doing the following:

1. Click on the **New** button at the top of the Subject Info dialog. The New Subject Info dialog will open.
2. Enter “Age” as the name for the first new item.
3. You will want to insure that a numeric value is entered for the subject’s age, so choose **Value** from the **Item Type** menu.

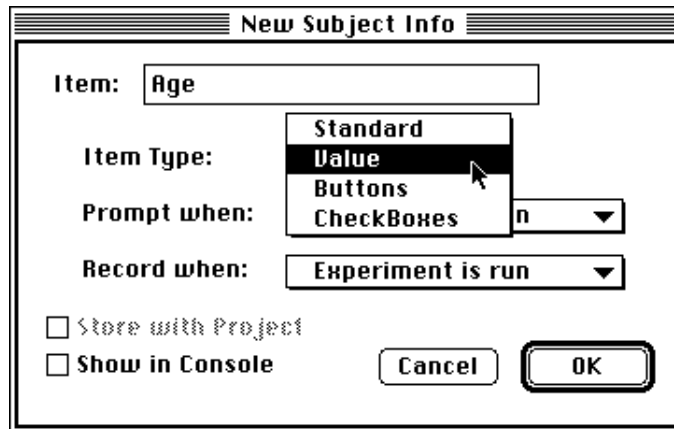


Figure 93 – Creating a new numerical Subject Info item

Now, if anything other than a numeric value is entered for the age, an error message will be displayed, and the user will be prompted to enter a numeric value. You will also be able to specify the particular type of number, and the range of values that are permissible, as you will see shortly.

- Next, use the **Prompt when** menu to choose the time at which you want PsyScope to prompt the user for the subject's age, or choose **Never** for not at all.

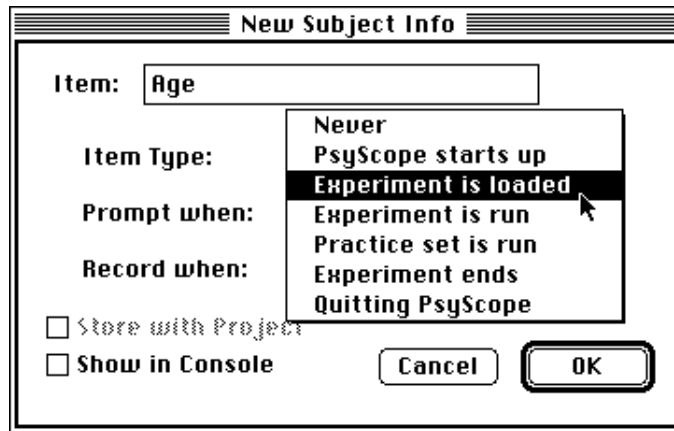


Figure 94 – Setting the prompt time

- Finally, you can set when you want PsyScope to record the information, using the **Record when** menu. For now, leave this set to **Experiment is run**. We will discuss how PsyScope records information shortly.
- Click on **OK**. After the New Subject Info dialog closes, the Configure Value dialog will appear. Check the **Range** box, and enter the range for permissible values of the subject's age.

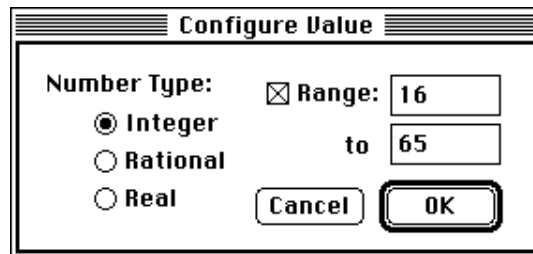


Figure 95 – Defining a numerical item

Now, any non-integer values, or values that are outside of the specified range, will generate an error message.

- Click on **OK** when you are done. PsyScope will present the dialog for entering the subject's age (try entering an inappropriate value), and then return you to the main Subject Info dialog.

Follow the same procedure to add handedness as an item. However, in this case, set the **Item Type** to **Buttons**. When you close the New Subject Info dialog, the Configure Val-

ue dialog will be different, letting you define the list of buttons instead of numerical features. Use this to create two buttons: one for Right and one for Left.

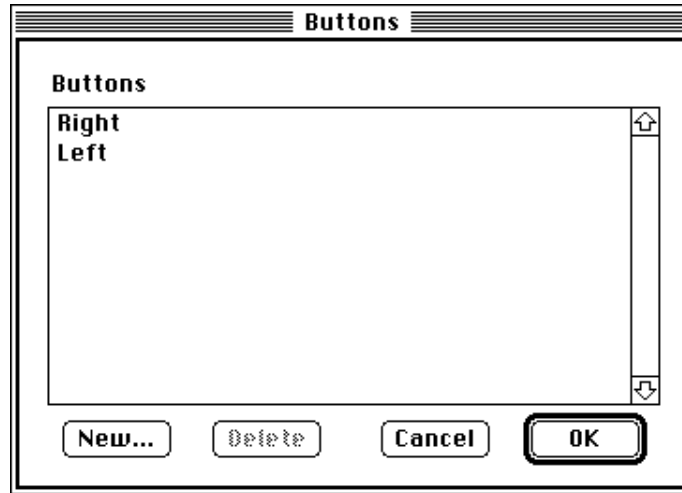


Figure 96 – Creating **Left** and **Right** buttons for Handedness

When you are done, PsyScope will present you with a dialog for entering the Subject's handedness, and then return you to the main Subject Info dialog, which should look something like this:

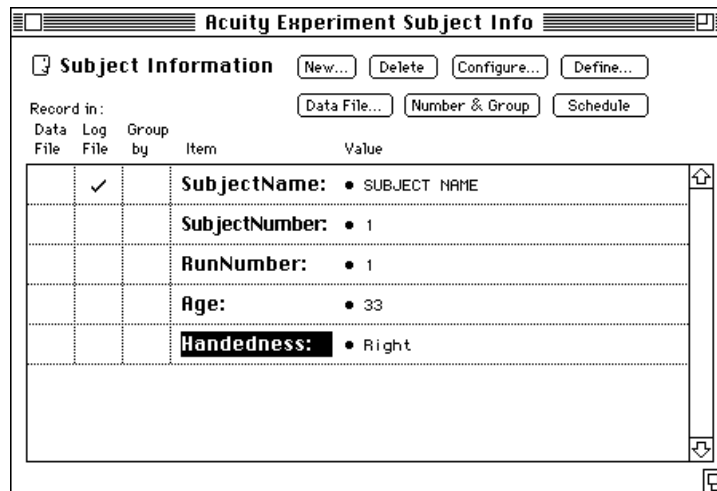


Figure 97 – Subject Info dialog with Age and Handedness added

Once you have created a subject info item, you can modify it by double-clicking on its name in the list, or by using the **Configure** and **Define** buttons at the top of the dialog. The **Define** button opens a dialog similar to the New Subject Info dialog, in which you can set when PsyScope will prompt for and store information. The **Configure** button re-opens the type-specific Configure Value dialog.

You can set the value for each item from the Subject Info dialog by clicking on its value in the list. You can specify where the information will be recorded by checking in the **Data File** and/or **Log File** columns. These are the two files in which PsyScope records information about the session, the experiment, and the subject. PsyScope stores information about each session in the log file, and about the running of each experiment in the data file. Below is some additional information about each of these files.

Log File and Data File

Log file

Whenever the PsyScope application is run, it opens a *log file* in which it records information about the session, such as the version of the PsyScope application that was run, the time each experiment began and ended, and any error messages that occurred.

By default, PsyScope uses a file named “PsyScope.log” in the same folder as the application. However, you can change the log file, as well as add comments to it and view it, using commands in the **Utilities** menu (see “Part 2: Graphic Environment Reference, 7.1.4 Utilities Menu”, p254).

To store information about the subject in the log file, check next to the items you want recorded in the **Log File** column of the Subject Info dialog. You can control when this information is recorded using the Subject Info dialog (see above) or Subject Info Schedule dialog (see “Part 2: Graphic Environment Reference, 6.2.6 Subject Info Schedule Dialog”, p237).

Data file

Whenever an experiment is run, PsyScope records all of the data from the experiment (such as the subject’s responses) in a *data file*.

By default, PsyScope creates a file whose name is the name of the experiment appended with “Data” (for example, “Acuity Experiment Data”), and stores it in the same folder as the PsyScope application. You can change the name and location of the data file, and view its contents, using commands in the **Utilities** menu.

To store information about the subject in the data file, check next to the items you want recorded in the **Data File** column of the Subject Info dialog. You can control when this information is recorded using the Subject Info dialog (see above). You can also use the Subject Info dialog to have PsyScope automatically create a name for the data file, based on information about the subject, as described below.

Note: Both the log file and data file are text files. That is, they do not contain any special codes or characters, so they can be viewed and edited by any standard text editor. However, PsyScope relies on the format of these files for certain types of information. Changing the format may interfere with PsyScope's ability to keep track of information about groups, subjects, and runs. See "Part 2: Graphic Environment Reference, 6.1.5 The Log File", p222 and "Part 2: Graphic Environment Reference, 6.1.4 The Data File", p217 for details about the format of the log and data files.

To have PsyScope automatically generate a custom name for the data file, click on the **Data File** button in the Subject Info dialog. This will open the Data File dialog.

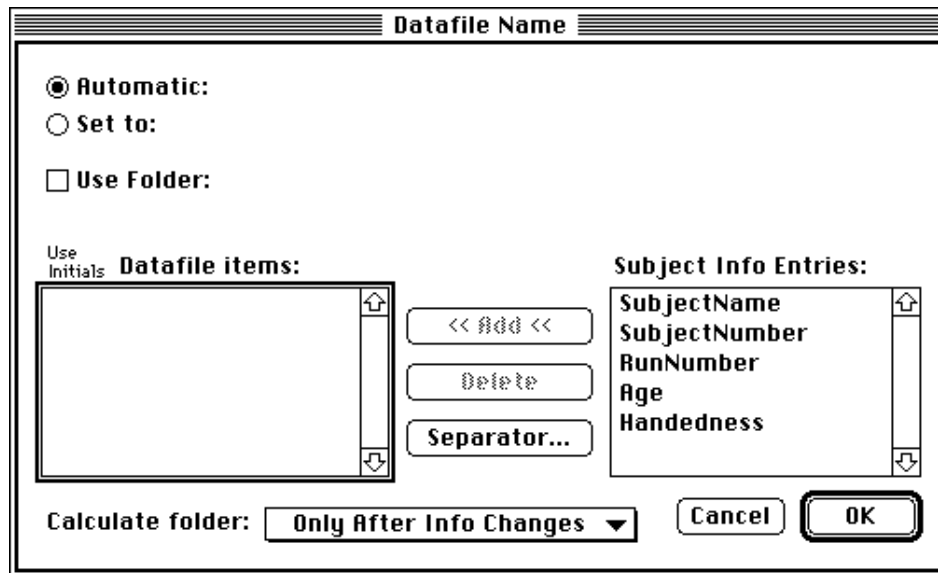


Figure 98 – The Data File dialog

The Subject Info items are shown in the list on the right of the Data File dialog. You can use the value of these items, their initials, and any separators you want to construct the name of the data file.

Let's have PsyScope generate a name for the data file based on the subject's initials, their handedness, and the subject number. For example, the data file for Archie Undergraduate who is subject 12 and is right handed should be AU12-R. For the purposes of this example, set SubjectName to Archie Undergraduate, the SubjectNumber to 12, and Handedness to Right, by clicking on their values in the Subject Info dialog and entering the values you want in the corresponding dialogs. Then return to the Data File dialog and follow these steps:

1. Select SubjectName in the list of Subject Info Entries and then click on the **Add** button (or just double-click on SubjectName). SubjectName should now appear under the **Data File Items** list.
2. Click in the **Use Initials** column next to SubjectName. in the **Data File Items** list.

3. Add SubjectNumber and Handedness to the list of data file items, and check Handedness.

Notice the example of the data file name next to the **Automatic** radio button at the top of the dialog. It should currently be “AU12R”. Now let’s add the hyphen in between the SubjectNumber and Handedness.

1. Click on the **Separator** button, enter a hyphen (“-”) in text dialog, and click on **OK**. The hyphen should now appear in **Data File Items** list.
2. Drag the hyphen above Handedness.

The Data File Name dialog should now look like this:

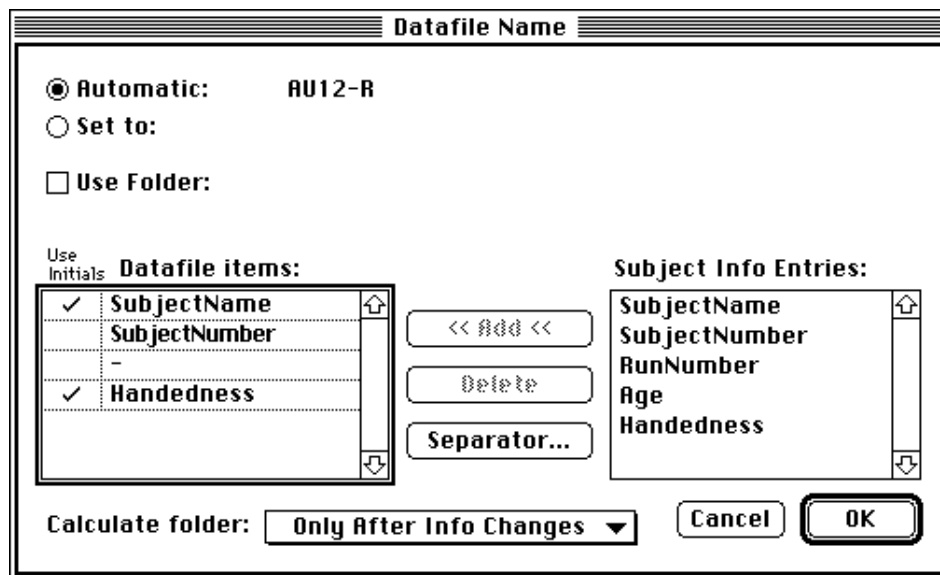


Figure 99 – Automatic Data File Name dialog with a calculated name

Try changing the value of SubjectName, SubjectNumber, or Handedness items, and notice how the data file name changes.

By default, the data file is saved in the same folder as the PsyScope application. You can save it in another folder by clicking in the **Use Folder** checkbox, and setting the folder using the standard Macintosh file dialog.

You can override the automatic data file calculation by clicking the **Set to** radio button, and entering the file name in the text field to its left.

3.2.10.2 Groups

There are two main reasons for using groups: to vary how the experiment is run for different groups of subjects, or to keep track of how many subjects have been run from different groups. The latter is particularly important for between-subject designs, in which you want to be sure that the same number of subjects from each group are run in each condition of an

experiment. In this section we will first show you how to define groups using the Subject Info dialog, and then show you how to use them for each of the purposes referred to above.

You define groups in the Subject Info dialog, by selecting the items that should be used to assign subjects to groups.

Let's group subjects by handedness.

- Check next to this item in the **Group** column of the Subject Info dialog.

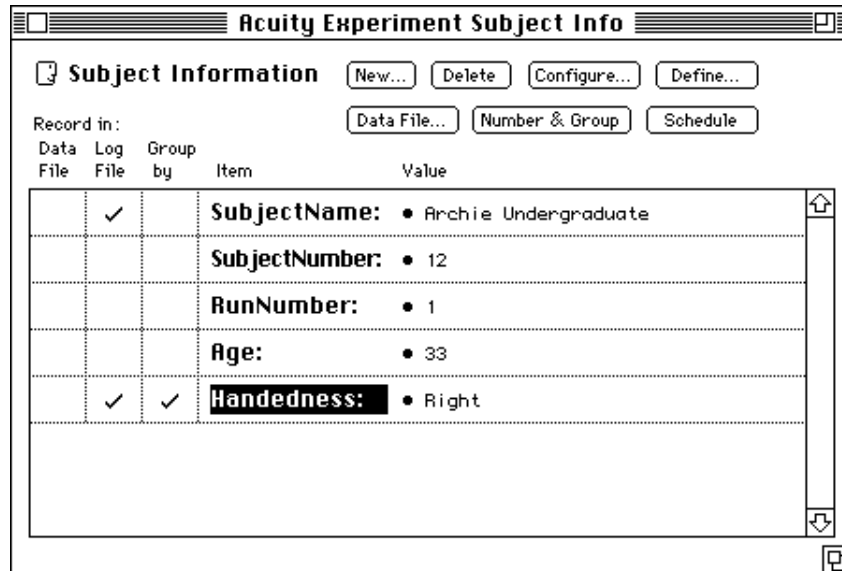


Figure 100 – The Subject Info dialog with grouping by Handedness

Notice that a checkmark automatically appears in the **Log File** column. This is because any information used to group subjects must be stored in the log file.

PsyScope decides which group to assign a subject to by checking the current values of information items that are used to group subjects (in the example above, it is just handedness). These information items are called the *group criteria*. All subjects that share the same values for all of the group criteria belong to the same group. In the example, all right-handed subjects will form one group, and all left-handed subjects will form another. If you had added Age as a group criterion, then all right-handed subjects of a given age would form one group, all right-handed subjects of a different age would form another group, etc.

When you define groups, PsyScope calculates the SubjectNumber for each subject relative to the group. For example, if the first three subjects were right-handed and the fourth was left-handed, the SubjectNumber for the 4th subject would be 1, not 4. This is important for between-subject designs, as we will see below (in “Groups and Between Subject Designs”, p89).

The Group Object and Varying By Group

As we noted above, one use of groups is to vary aspects of the Experiment based on the group that the subject is in. For example, suppose that you wanted to vary the input key used for responding based on the subject's handedness. You can do this using groups, and **Vary By Group** for the input condition.

You defined the two handedness groups above. To be able to vary the experiment according to these, you now need to create two group objects, and “splice” them into the experiment.

1. Go to the Design window, select the group object in the Tools palette (see figure 101 below), and place two groups in the work area. Name them “Left Handers” and “Right Handers” respectively.\



Figure 101 – The Group icon

2. Using the scissors tool (see figure 102 below), clip the links from the experiment to the block objects. Then link the two groups to the experiment, and to the blocks.



Figure 102 – The scissors tool

Tip: You can simultaneously link multiple objects to another one by selecting the link tool, then selecting all of objects to be linked and dragging to the object you want to link them to.

The Design window should now look something like this:

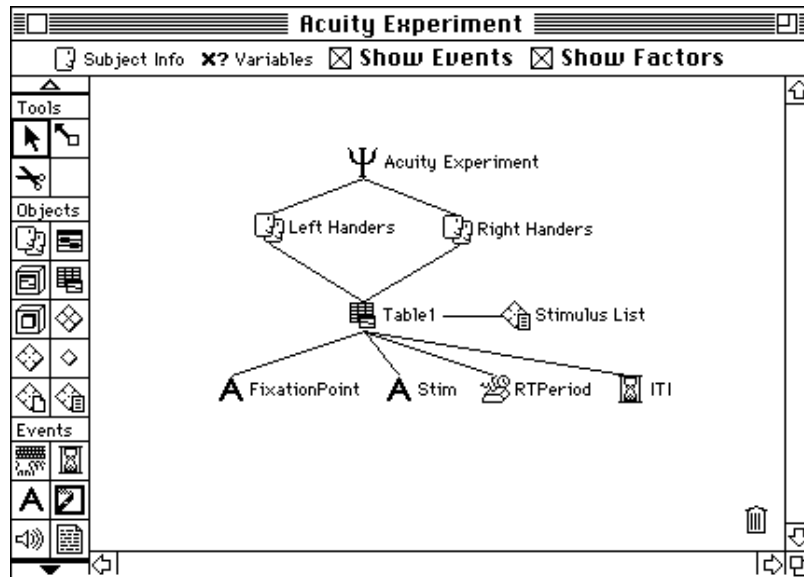


Figure 103 – Experiment hierarchy with groups

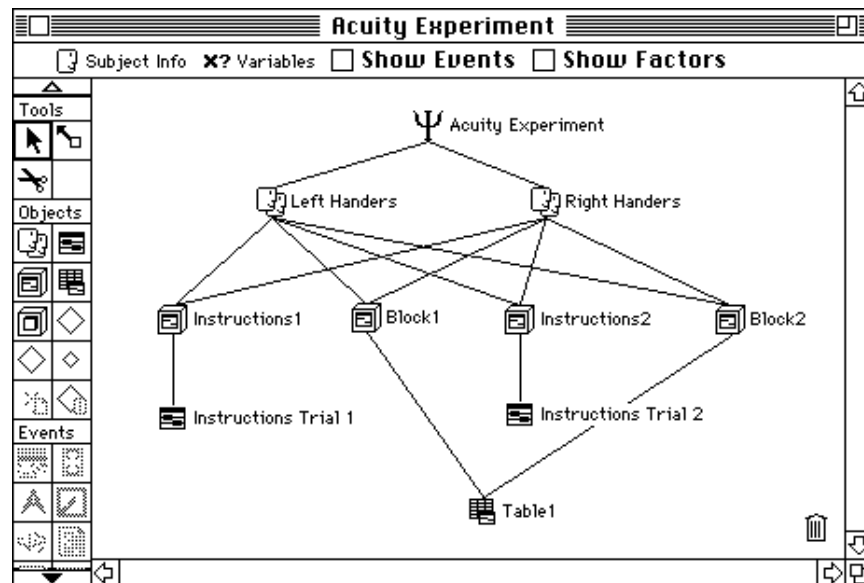


Figure 104 – Experiment hierarchy with groups and blocks

The next step is to link the group criteria you defined earlier to the two group objects.

1. Open the Group dialog for the Left Handers group, and click on the **Criteria** button in the lower right hand corner. This will open the Criteria dialog:

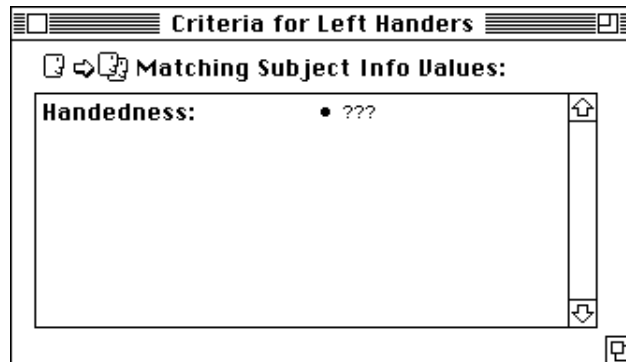


Figure 105 – The Group Criteria dialog

Notice that Handedness is the only item listed in the dialog. This is because this is the only item that was checked as a group criterion in the Subject Info dialog.

2. Double-click on **Handedness** and then click on **Left** in the Handedness dialog.
3. Do the same for the Right Handers group, setting the Handedness criterion to **Right**.

The criteria for each group are now set. The final step is to create a custom group attribute for each group that specifies which input key is to be used for each group, then vary the input condition for the RT Period event by group. The procedure for doing this is analogous to the one for creating custom block attributes, which you saw earlier. In case you skipped that section here is how you do it. Create the custom attributes first:

1. Go back to the Left Handers group dialog, and click on the **Attributes** button.
2. Choose **Custom Group Attribs** from the **Attribute Set** menu, if it is not already selected, and then click on the **New** button. This will open the **New Attribute** dialog.

3. Select **Conditions and Actions** from the **Attribute Set** menu.

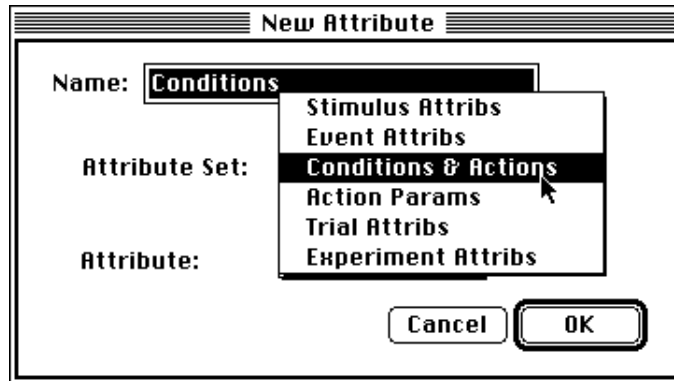


Figure 106 – The Create/Rename/Retype Attribute dialog (used here to create a custom attribute)

Check that **Conditions** is selected in the **Attribute** menu, and then click on **OK**. This will return you to the Left Handers Attributes dialog, and **Conditions** should appear as a custom group attribute.

4. Set the value of the **Conditions** attribute to the “f” and “d” keys (See the example in “3.2.6 Recording Responses”, p26, for a reminder of how to do this).

When you are done, the Left Handers Attributes dialog should look like this:

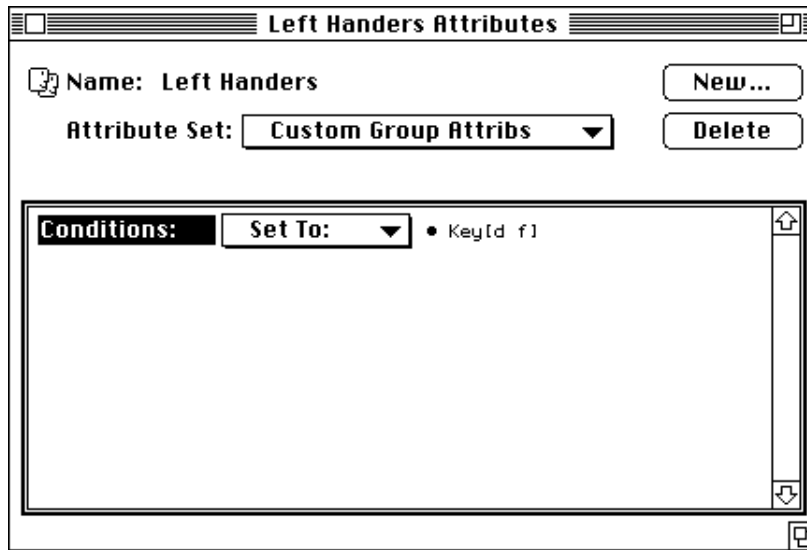


Figure 107 – Group Attributes dialog showing a custom attribute

Once you have defined Conditions as a custom group attribute, it will be there for all groups. Therefore, all you have to do for the Right Handers group is open its attributes dialog, and set the value of Conditions to the “j” and “k” keys.

*Note: Whenever you create a custom attribute for an object of a particular type, PsyScope automatically adds it to all objects of that type in the experiment, and sets their value to **Default**.*

All that remains to be done now is to set the input for the RT Period event so that it will be determined by the subject's group. For left-handed subjects, the response keys on the keyboard will be "d" for word and "f" for non-word, while right-handed subjects will use "j" for word and "k" for non-word:

1. Make sure all of the cells in the factor table are selected, and then go to the Template window and open the Event Attributes dialog for the RT Period event.
2. Select **Set To:** in the **Input/Actions** attribute menu. This will open the Actions dialog.
3. Select the conditions part of the conditions-actions pair (set up in "3.2.6 Recording Responses", p26), but do *not* double-click to open the Conditions dialog.
4. In the pop-up menu which appears at the bottom of the Actions dialog, select **Vary by Group**; this will open the Vary by Group dialog.

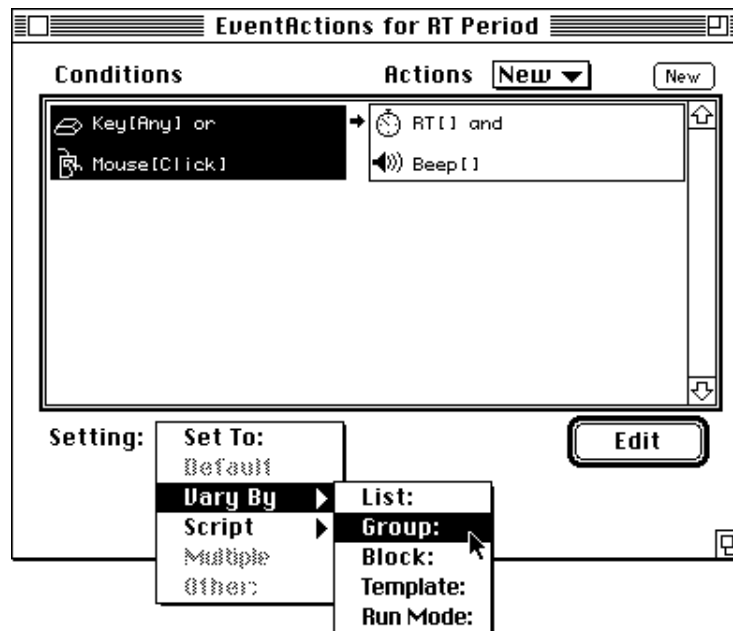


Figure 108 – Selecting **Vary by Group**

3. Choose **Conditions** from the **Attribute** menu in the Vary by Group dialog, and click on **OK**.

The Actions dialog should now look like this:

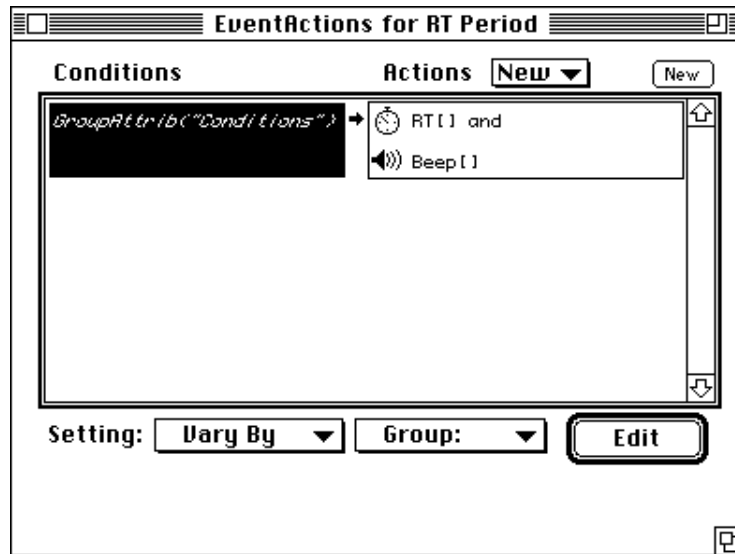


Figure 109 – The Actions dialog showing conditions varied by group

This indicates that the value of the conditions will be determined by the Conditions custom group attribute. Try this out, by setting the value of the handedness in the subject info dialog, and then running the experiment.

The Current Group

Whenever you have group objects in your experiment, there is always one *current group*. This is the group that is used to set the values for all attributes that are set to **Vary by Group**. If you have defined group criteria, then the current group is determined by the values of the subject info items that make up the group criteria (for example, if handedness is set to “Right”, then Right Handers is the current group). If no group criteria have been defined, you can select the current group manually from the Experiment dialog.

Set the current group to Left Handers manually as follows:

1. Go to the Design window, and double-click on the Acuity Experiment icon. This will open the Experiment dialog

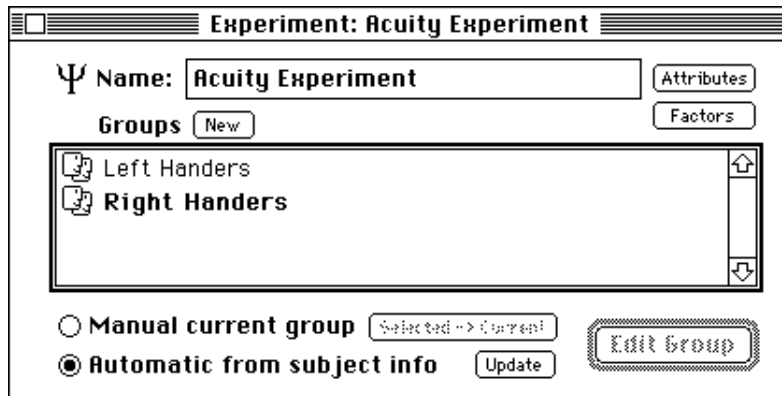


Figure 110 – Group List dialog showing two groups

Notice that Right Handers is bold faced. This is the current group.

2. Click on the **Manual current group** radio button, select Left Handers, and then click on the **Selected -> Current** button.

Left Handers should now be the current group. Confirm this by running a trial. “f” should be the input key.

Setting the current group manually overrides the group criteria, if any have been defined. You can reset automatic group selection by clicking on the **Automatic from subject info** radio button. Clicking on the **Update** button will re-check the value of all of the group criteria, and set the current group accordingly.

Groups and Between Subject Designs

In the previous section, you learned how to use groups to vary features of the experiment across groups. A second important use of groups is to keep track of the number of subjects run from each group. This is particularly important for between-subject designs, when you want to be sure that the conditions of the experiment are distributed across subjects in the same way for each group.

Suppose, for example, that we want to run the Acuity Experiment in normal and visually impaired subjects, to test for differences in peripheral acuity between these two types of subjects. Suppose furthermore that we want to examine the influence that different fonts have in these two different groups, and that we want to do this in a between-subject design. That is, each subject will see only one of the fonts, which we will vary from subject to subject (as in the example in “3.2.8.3 Controlling How Cells are Chosen”, p49). We want to be sure that the assignment of fonts to subjects occurs in the same way for the two groups. That is, we want the between-subjects table to be used in an equivalent way across groups. For example, if we have already run three normal subjects who have been assigned the first three fonts, and the next subject is visually impaired, we want that subject to see the same font seen by the first normal subject, and not simply the next font in the list.

Using groups and setting the crossing type of a factor table to **Between**, and then choosing **Subject within Group** in the Choose Crossing dialog (see “3.2.8.3 Controlling How Cells are Chosen”, p49 for an example of how to do this) insures that the assignment of cells from the table will be done in an equivalent way across groups. This is because PsyScope picks cells based on the value of SubjectNumber when a table’s crossing type is **Between** and the **Between** type is **Subject within Group**. Since, with this setting, PsyScope computes the SubjectNumber separately for each group, corresponding members of each group are numbered the same, and so they will be assigned a comparable cell. For example, the SubjectNumber of the first subject in both the normal and the visually impaired groups will be “1”, it will be “2” for the second subject in both groups, etc. Since this number is also used to choose the cell in Font factor’s table — its Crossing Type is **Between** by subject — corresponding members of each group will be assigned the same level of that factor.

This introduction to Groups should give you an idea of how they can be used, in combination with between-subject factors, to construct a variety of complex experimental designs. In the next section, we turn to a set of features that will help polish your experiment, and get it ready to run with actual subjects.

3.2.11 Experiment Attributes

Experiment attributes are features that can be used to customize your experiment. There are a variety of these, some of which are fairly technical. Here, we will describe three sets of these that are of general use. The remaining experiment attributes are described in detail in “Part 2: Graphic Environment Reference, 5.8.3.4 Standard Experiment Attributes”, p161.

All of the experiment attributes are set in the **Experiment Attributes** dialog. You can open this dialog from the Design Window — like the attributes dialog for any other type of object — in one of two ways: Either by double-clicking on the experiment object to open the Experiment dialog, and then clicking on the **Attributes** button; or directly by holding

down the control key while you click on the experiment object with the mouse, and then selecting **Edit Experiment Attribs** from the pop-up menu that appears.

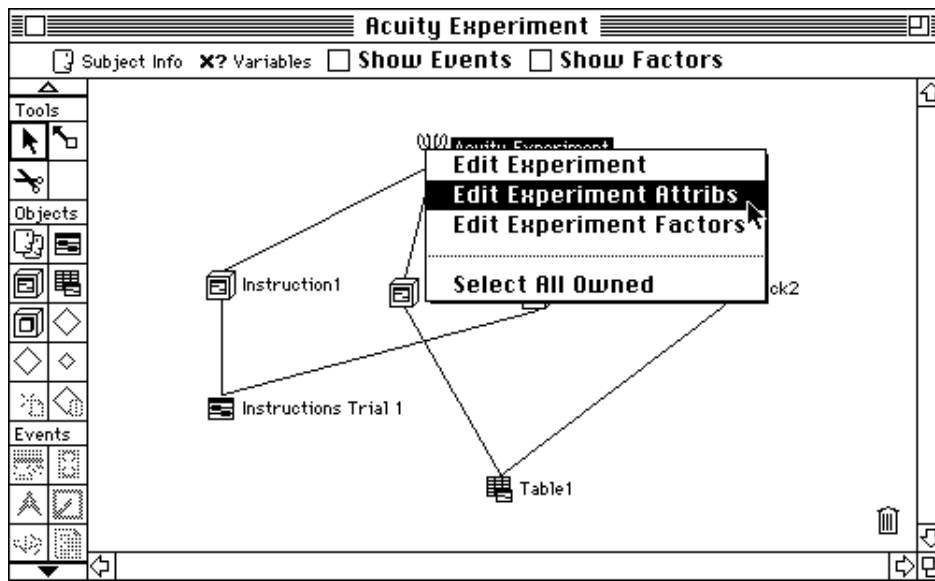


Figure 111 – Control-clicking to get to attributes quickly

3.2.11.1 Instructions and Debriefing Files

In the example that we used in the section on blocks, we showed you how to present instructions before each block. Usually, however, you will only want to present a single set of instructions at the beginning of the experiment, and perhaps another message at the end of the experiment. For routine cases like this, PsyScope provides a simpler method, so that you do not have to create special blocks or templates for this purpose. All you have to do is create a text file for each message, and then set the **Instructions File** and/or **Debriefing File** attributes in the **Experiment Attributes** dialog.

Create an instructions message as follows:

1. Go to the **File** menu and select **New Text File**.
2. Enter the instructions in the text editing window, and then save the file.
3. Go to the Design window, and open the Experiment Attributes dialog.
4. Select **Set To** from the menu of the **Instructions File** attribute. This will open a standard Macintosh file dialog. Select your instructions file and then click on **Open**.

Try running the experiment. At the very beginning, a bordered window will appear showing your instructions. You can experiment with the positioning of the text within the window by using returns, tabs, spaces, etc.

Instructions are presented at the beginning of the experiment, before any trials are run; the debriefing text is displayed at the end of the experiment, after the last trial has been run.

3.2.11.2 Rest Periods

As with instructions, you can use blocks and templates to add rest periods at specified points in your experiment. Also like instructions, PsyScope provides a simpler method for standard cases. You can add “pre-packaged” rest periods to your experiment, using the **Trials Per Rest**, **Num Rests**, and **Rest Duration** experiment attributes. With **Trials Per Rest**, you specify how often you want rest periods to occur, in terms of the number of trials between each one. With **Num Rests**, you specify how many rest periods you want in the experiment. PsyScope then distributes these evenly throughout the experiment, by dividing the total number of trials in the experiment by the **Num Rests** value you specify.

*Note: The **Num Rests** attribute does not work correctly — in such cases, rests may not be distributed evenly— with blocks that have time durations; this is because PsyScope cannot, in advance, predict how many trials will be run.*

The **Rest Duration** attribute determines how long — in milliseconds — the rest periods last. During each one, PsyScope presents a blank screen with the message: “You can take a break now”. When the rest period ends, PsyScope adds the message: “Please hit any key to continue”. Pressing any key on the keyboard will continue the experiment.

3.2.11.3 Reverse Video

Sometimes you may want all of the text stimuli in the experiment to appear in white against a black background; that is, in reverse video. You can do this by choosing **Reverse Video** from the **Experiment** menu in the menu bar, and then clicking the **On** radio button in the Reverse Video dialog. This actually sets the **Backcolor** and **Forecolor** experiment attributes; **Backcolor** is set to black, while **Forecolor** to white.

You can set these attributes directly to achieve other color effects. The value of **Forecolor** is used for any stimulus in the experiment whose color is unspecified, and the **Backcolor** attribute determines the background color.

3.2.12 Running Trials

There are a number of ways of running trials in PsyScope. Of course, you can run the whole experiment as you have been doing. You can also run subsets of trials — to preview or debug them — as well as practice trials.

3.2.12.1 Running the Experiment and Breaking

Running the experiment is simple: Select **Run** from the **Run** menu, type Command-R, or click on the **Run** button in the Console. The number of trials run will be determined by settings in the Experiment and Block dialogs. These are discussed in “Part 2: Graphic Envi-

ronment Reference, 5.12.2 Trial Counting”, p211. You can interrupt the experiment at any time by typing Command-.; this will invoke the Break dialog.

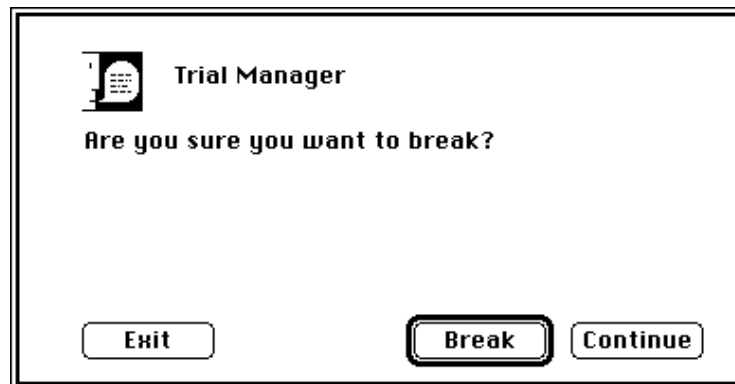


Figure 112 – The Break dialog

Clicking the **Continue** button will continue the experiment where you left off. Clicking on **Break** button will quit the experiment and return you to PsyScope.

3.2.12.2 Practice vs. Run Mode

You can run a set of trials in Practice mode by choosing **Practice** from the **Run** menu or by clicking on the **Practice** button in the Console. The experiment is run exactly as it would be in a regular run, except that in practice mode PsyScope does not store any data in the data file, and RunNumber is not incremented. The practice run *is*, however, logged in the log file.

You can also use custom practice and custom run attributes to vary features of trials according to the mode in which they are run. You do this in a way similar to creating and assigning other custom attributes (see the “3.2.9.2 Varying by Block”, p72 and “The Group Object and Varying By Group”, p83 sections above). First, you choose **Custom Practice Attribs** or **Custom Run Attribs** from the **Attribs Set** menu in the Experiment Attributes dialog. Next, you create the new custom attributes that you want (creating a new attribute in one set automatically creates it in the other), and assign their values for the Run and Practice modes. You can then use these to vary the value of any attribute by setting its value to **Vary By Run Mode**.

*Tip: While **Vary by Run Mode** provides a certain amount of control over practice trials, you may still find yourself limited by the fact that the same trials will be run in both modes. If you wish to differentiate more fully between practice and experimental trials, you can do this by creating a separate block of practice trials, that you place first in the list of blocks for the experiment. If you don't want to record data during these trials, simply omit any RT[] actions from that set of trials.*

3.2.12.3 Previewing Trials and Using the Trial Chooser floating window

When you are in the process of designing an experiment, it can be useful to run some trials to see how things are shaping up. You can do this using the **Preview** button in the Template and Block dialogs. When a trial is run in preview mode, PsyScope does not store any data collected during the trial.

The **Preview** button in the Template window runs a single trial from that template. You can use the Trial Chooser floating window to specify which trial to run; that is, to set the value of any determinants of the trial. For example, if any attributes in the trial are linked to lists, then you can use the Trial Chooser to select the specific items from those lists that you want to preview in the trial. Similarly, if any of the attributes in the trial are varied by block, the Trial Chooser lets you select which block to use when the trial is previewed. The same is true for groups.

Preview a trial from the Template window, as follows:

1. Open the Template window.
2. Open the Trial Chooser palette, by selecting it in the **Windows** menu.
3. Select an item from the Stimulus List in the Trial Chooser:

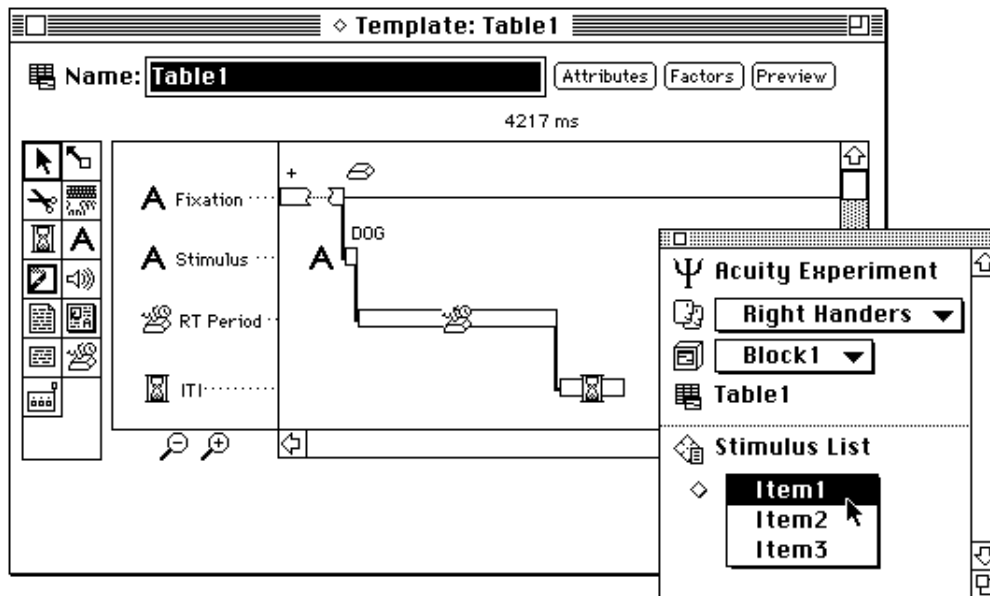


Figure 113 – The Template window with the Trial Chooser open

Recall that the stimulus attribute of the Stimulus event is linked to the Stimulus List. Notice that when you select an item from the Stimulus List, that item is displayed as the stimulus above the Stimulus event bar in the Template window.

4. Click on the **Preview** button at the top of the Template window. The trial should run, and the stimulus should be the item that you selected from the Stimulus List.

Try changing the group in the Trial Chooser, and previewing the trial. Recall that the response key in the RT Period event is varied by group, so it should change according to the group selected in the Trial Chooser.

Note: Selections in the Trial Chooser palette only effect the previewed trial. They have no effect on trials when the experiment is actually run

The **Preview** button in the Block dialog will preview all of the trials in that block. The Trial Chooser can be used to select which group to use. It can also be used to set which item to select first from any lists to which attributes are linked. After the first trial is previewed, PsyScope will continue picking items as usual.

Previewing trials lets you see how they will actually run in the experiment. In some circumstances, however, you may want to conduct more sophisticated tests. The Trial Monitor lets you do this.

3.2.12.4 The Trial Monitor

The Trial Monitor provides you with more elaborate control over how trials are constructed and run, as well as some diagnostic tools for testing your experiment. The top half of the monitor controls how many trials will be run; the bottom half controls how they are constructed, and provides diagnostics.

Open the Trial Monitor by choosing **Monitor** from the **Windows** menu, clicking on the **Monitor** button in the Console, or typing Command-M.

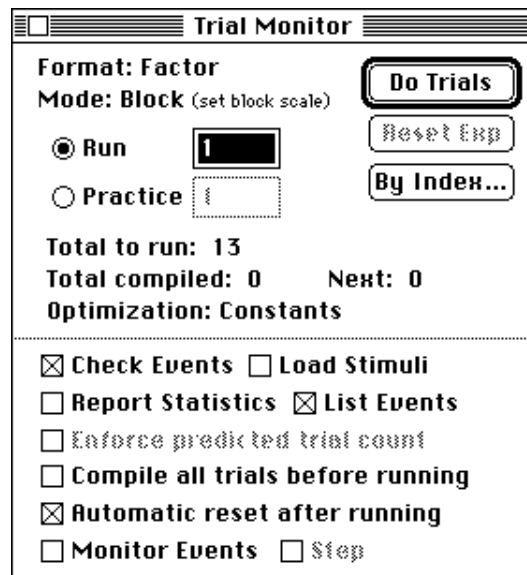


Figure 114 – The Trial Monitor

The **Run** and **Practice** fields at the top of the monitor specify the number of trials to run. If there are blocks in the experiment, **Mode: Block** appears at the top of the monitor, and the Run and Practice values function like the **Scale** value in the Experiment dialog — that is, they multiply the number of trials in each block. **Total to run** reports the actual number of trials that will be run.

Notice that the **Total to run** value is 13. This number reflects the single trial in each of the instructions blocks, the 10 trials in Block1, but only one trial for Block2. Only one trial is counted for Block2 because it has a time duration specified rather than a number of trials. PsyScope cannot accurately calculate beforehand how many trials will run in a specified period of time, therefore it cannot accurately calculate the total number of trials that will run in the experiment.

*Note: If any blocks in the experiment have a time duration specified rather than a number of trials, then the value reported by **Total to run** will be inaccurate — only one trial will be counted for each timed block.*

Open the Block dialog for Block2, and assign it 10 trials in the block. The Trial Monitor should now accurately report **Total to run** as 22.

Try changing the value in the **Run** field to 2. **Total to run** will now be 42. This is because this value functions like the **Scale** value, multiplying the number of trials in each block. However, because the Instructions blocks were set to **Fixed** rather than **Scalable**, they are unaffected. So, setting **Run** to 2 doubles the number of trials in Block1 and Block2, for a total of 20 in each, but keeps only one trial in each of the two instructions blocks, for a total of 42 trials in the experiment.

*Note: The value in the Run field may be linked to the **Scale** field of the Experiment dialog by default. Changing the value in one changes the value in the other. You can uncouple these by checking **Trial Monitor count separate from script** in the Run Options dialog (see “Part 2: Graphic Environment Reference, 7.6.2 Run Options”, p266).*

If there are no blocks in the experiment, then the Run and Practice values directly specify the number of trials to run; this is indicated by **Mode: Direct**.

To run the trials, be sure that the mode you want is selected (by clicking on either the **Run** or **Practice** radio buttons), and then click on the **Do Trials** button. You can run a subset of trials, using the Run By Index dialog. Open this by clicking on the **By Index** button, enter the number of the first and last trials you want to run, and then click on the **Do Trials** button in the dialog.

Precompiling Trials and the Intertrial Interval

By default, PsyScope constructs each trial just before it is run. Depending upon the complexity of your experiment, this process can take anywhere from one millisecond to one second or longer, and can vary from trial to trial. There are two ways you can handle this, if the timing between trials - the *intertrial interval* (or *ITI*) - is critical to your experiment.

If you just want to be sure that the ITI is consistent from trial to trial, you can set the **Minimum ITI** attribute “Part 2: Graphic Environment Reference, 6.5.1 Precompiling”, p246. If you want to eliminate it altogether, you can precompile trials and use the **Preload All Stimuli** special attribute flag. Each of these is discussed below.

Minimum ITI

This is a trial attribute, that specifies the minimum amount of time to wait before beginning each trial. PsyScope constructs the trial during that period, including loading any stimuli that it will need from disk. The default value is 0, so that PsyScope will begin the trial as soon as it has been constructed. This means that some trials may begin sooner than others. However, by setting this to a value equal to or longer than the time it takes to construct the longest trial, you can ensure that all of the ITIs will be the same. Typically, a value of 1 second will do the trick. You can be more precise by running a check on the experiment (see below), to find out the longest amount of time it will take to compile a trial.

Precompiling Trials

Setting **Minimum ITI** ensures that ITIs will be consistent. However, if you want to eliminate the time occupied constructing each trial before it is run (for example, you need very brief ITIs), then you can have PsyScope construct all of the trials at once, before the experiment begins. You can do this by checking **Compile all trials before running** in the bottom half of the Trial Monitor, or setting the **Precompile** experiment attribute to **All**. When you run the experiment, PsyScope will begin by compiling the trials. While it is working, it will present a message dialog indicating how many trials are being prepare, along with a time bar tracking its progress and an estimate of how much longer it will take.

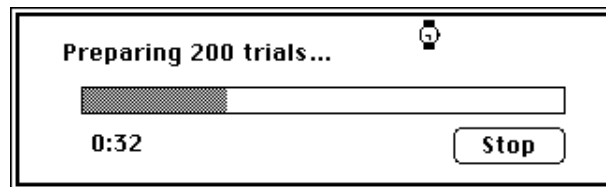


Figure 115 – The experiment compiling timebar

If you are using the **Instructions** experiment attribute, this message will appear below the text, in the same window. You can suppress the number of trials in the message (if you do not want the subject to see this information) by unchecking **Show number to be compiled in run time bar** in the Run Options dialog (see “Part 2: Graphic Environment Reference, 7.6.2 Run Options”, p266).

Even if you precompile trials, there may still be some time occupied between trials as PsyScope loads the stimuli for the next trial from disk, and performs any manipulations on them that you have specified (for example, flipping text vertically or horizontally). You can eliminate this time as well, by setting the **Preload All Stimuli** special experiment flag. Doing this will ensure that the ITI is 0. However, you must have enough RAM available for PsyScope to be able to store all of the stimuli in the experiment in memory at once.

The advantage of precompiling trials and preloading stimuli is that you can then make the time between trials as short as you like. The disadvantage is that you have to wait for all of the trials to be constructed and stimuli loaded before the experiment begins. If you have a large number of trials or stimuli, this can take a while.

See “Part 2: Graphic Environment Reference, 6.5.1 Precompiling”, p246 for more information on precompiling.

Note: Precompiling trials will not work if your experiment uses any Trial Manager Variables (see “Part 2: Graphic Environment Reference, 5.10 Trial Manager Variables”, p205), or if of any blocks have a time duration specified rather than a specific number of trials (this is because, for timed blocked, PsyScope cannot calculate beforehand how long each trial will take, and therefore how many to allocate to the block).

Checking Trials

You can have PsyScope check your experiment by clicking in the **Check** box in the bottom half of the Trials Monitor, and then clicking on the **Do Trials** button. PsyScope will construct the number of trials in the **Run** field (without actually loading stimuli — see below), and report any errors it encounters in the process. When it is done, instead of running the experiment, PsyScope will open the Check Statistics window. This window reports several statistics about the experiment, which are described in “Part 2: Graphic Environment Reference, 6.3.1 Trial Compilation Statistics”, p241. These statistics are useful for deciding how to set **Minimum ITI**, to ensure that the ITIs are consistent when you are not precompiling the experiment (see above, “Precompiling Trials and the Intertrial Interval”, p96).

Checking the **List Events** box in the monitor causes PsyScope to list the events of each trial, and their associated stimuli, in the Check Statistics window. This is useful if you want to verify that randomization of stimuli is occurring as anticipated.

Checking **Load Stims** causes PsyScope to load every stimulus from disk when checking the trials. This will take longer, but will check that every stimulus file in your experiment is appropriately specified and accessible.

A full description of the Trial Monitor is provided in “Part 2: Graphic Environment Reference, 6.3 The Trial Monitor”, p238.

The Event and Variable Monitors

Checking the **Monitor events** box at the bottom of the Trials Monitor opens two windows: the Event Monitor and the Variable Monitor. These windows dynamically track the trials of your experiment as they run.

The Event Monitor reports which trial is being run, all of the events in that trial, and their current status; it also maintains a list of all active functions.

The Variable Monitor reports the status of trial manager variables. If the **Step** box is checked, then the trial will run “step-by-step,” so that you can examine the progress of the trial more carefully. You advance each step by pressing any key on the keyboard.

The Event Monitor and Variable Monitor are extremely useful debugging tools, especially when you are designing an experiment using PsyScript. These are described fully in “Part 2: Graphic Environment Reference, 6.4 The Event Monitor and Variable Monitor”, p241.

3.3 Where to Go from Here

This is the end of your introduction to the graphic environment, and the structure of an experiment in PsyScope.

Details of each of the components discussed above, as well as all of the event types and actions available in PsyScope, are in “Part 2: Graphic Environment Reference”.

The scripting language is introduced in “Part 3: Scripting User Manual”, with a tutorial for creating the Acuity experiment using PsyScript instead of the graphic environment. “Part 4: Scripting Reference” is the corresponding reference section; it contains a complete definition of PsyScript, the available experiment scripting formats, technical details of the Trial Manager, and information about using PsyScript to configure the interface.

Finally, “Part 5: Appendices” provides a list of error messages, details about using the button box, and creating sounds and pictures for use with PsyScope.

Part 2:

Graphic Environment Reference

Chapter 4. Introduction 103

Chapter 5. Windows and Dialogs 105

Chapter 6. Running and Managing Experiments 213

Chapter 7. User Environment 253



Chapter 4. Introduction

This part describes, in detail, how elements of an experiment are created and modified in PsyScope. “Chapter 5. Windows and Dialogs” contains sections on each of the major components of a PsyScope experiment: the experiment itself and the components that make it up — groups, blocks, templates, events, attributes and factors. “Chapter 6. Running and Managing Experiments” provides information on running and managing experiments. “Chapter 7. User Environment” contains information about the general environment.

4.1 A Word About Scripts

PsyScope provides a powerful and easy-to-use facility for designing experiments in graphic form. However, like most graphic approaches to programming, it has inherent limitations. Inevitably, you will find that some experiments — or parts of some experiments — cannot be created using just graphic elements. For these, you will need to use PsyScript, the PsyScope scripting language.

The graphic tools in PsyScope actually create a PsyScript script for your experiment. You can view this script at any time, and edit it directly. In fact, anything you do in the graphic environment is reflected immediately in the script. Conversely, changes made in the script will appear immediately in the graphic environment (as long as it can be represented graphically).

Eventually, you will want to make use of PsyScript to take full advantage of the power of PsyScope. PsyScript is covered in detail in “Part 3: Scripting User Manual” and “Part 4: Scripting Reference”.



Chapter 5. Windows and Dialogs

In the PsyScope graphic environment, you use windows and dialogs to design an experiment. You can keep most of these open as long as you wish, returning to them as you need to. This chapter goes through all of them, describing each window or dialog's purpose and use. Details concerning the concepts and objects that these dialogs correspond to are provided along the way.

5.1 Windows vs. Dialogs

The main difference between windows and dialogs in PsyScope is that windows contain graphic elements (or icons) which represent the components of an experiment and tools for manipulating them, whereas dialogs contain only text (plus non-manipulable graphics).

The standard Macintosh conventions apply for working with objects in a window or dialog. To select objects, click on them with the mouse. To select multiple objects either hold the shift key down while selecting, or hold the mouse button down and enclose the desired objects with the "rubber band". To move objects, drag them with the mouse button held down.

5.1.1 Windows

The three primary windows in PsyScope are the Design window, the Trial Template window, and the Factor Table window.

The Design window is used to view and edit the experiment as a whole; each experiment has only one Design window. The Design window is described in "5.2 The Design Window", p107.

The Trial Template window is used to view and edit specific types of trials. A single experiment can have many Trial Template windows, one for each type of trial in the experiment. The Trial Template Window is described in "5.6.1 The Trial Template Window", p124.

The Factor Table window is used to define and manipulate the factor structure of the experiment. An experiment can have many Factor Table windows. The Factor Table window is described in "5.7.2.1 The Factor Table Window", p138.

5.1.2 Dialogs

Dialogs are the basic method of setting parameters for objects within the graphic environment. Unlike dialogs in many programs, most dialogs in PsyScope are non-modal — that is, you do not ordinarily have to close them in order to work with other dialogs and windows. (You can recognize non-modal dialogs by the absence of **OK** and **Cancel** buttons.)

To open the dialog for an object, double-click on it. To close the dialog, click in the close box at the left of the title bar, just as you would close any window.

Dialogs are made up of the following areas:

- text fields
- list boxes
- buttons
- selection devices (checkboxes, radio buttons, and pop-up menus)

Text fields and list boxes contain items that you can edit. To activate an item for editing, click on it once with the mouse or tab to it (as described below).

To select an item in a list box, click on it once with the mouse. To select a list of items, hold down the Shift or Command key while clicking the mouse. (Many lists do not allow multiple selections, however.)

Active text and list items accept certain keyboard commands — such as the arrow keys — for moving around or selecting an item. When a text field is active, its contents will be highlighted; when a list box is active, its border will be bolded. You can change the active item in a dialog by pressing the Tab key; this will cycle through items in a top-left to bottom-right order.

In an active text field, the cursor responds to the arrow keys in the usual way. Holding down the Shift key while the cursor is moved will highlight an area of text.

In an active list, the arrow keys can be used to move around the list. The Shift key can also be used here to extend a selection (if multiple item selections are allowed). Command-Down arrow is usually the same as double-clicking on the selected item in a list.

Buttons are used in dialogs to open another window or dialog, or to perform some immediate action. In modal dialogs, you can usually press a button by typing Command and the first letter of the button's name.

Selection devices — checkboxes, radio buttons, and pop-up menus — are used to choose among a fixed set of options. These operate in the standard way.

5.2 The Design Window

To open the Design window, use one of the following methods:

- Click on the **Design** button in the Console.
- Select **Design** from the **Windows** menu.
- Type Command-2.

The Design window is shown in the figure below. It is composed of a Control area, tool palettes, and a Work area. The palettes can be optionally moved to a floating window (through the Design Options dialog, see “7.6.4 Design Options”, p268).

The Work area of the Design window displays all of the objects that have been created for the experiment, and their hierarchical relationship to one another.

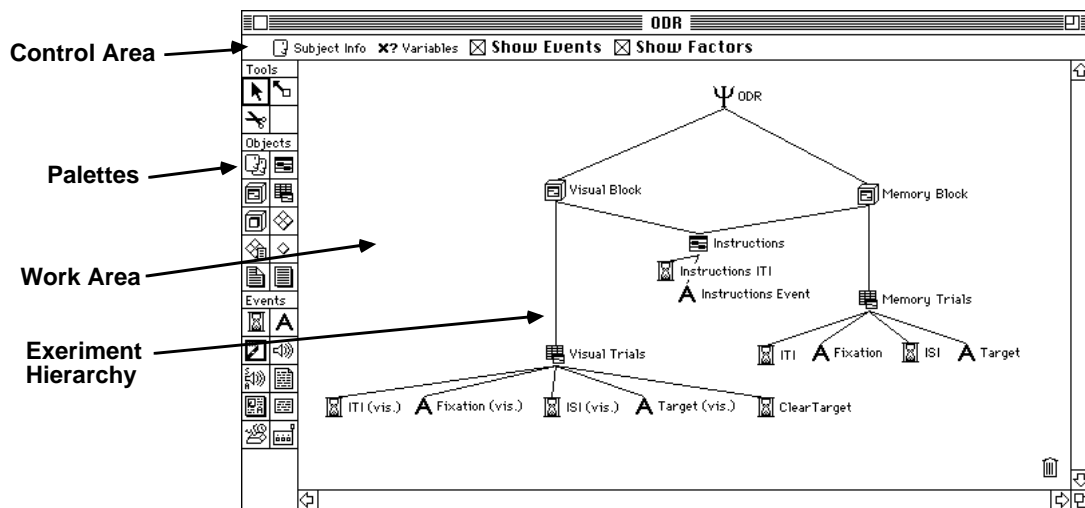


Figure 116 – Anatomy of the Design window

5.2.1 Objects and the Experiment Hierarchy

An experiment in PsyScope is made up of a number of different components which are arranged in a hierarchy. In brief, an experiment is made up groups, a group is made up of blocks, a block is made up of templates and/or tables, and templates and tables are made up of events. An example hierarchy is illustrated in the “Figure 116 – Anatomy of the Design window”, above.

An *object* is a particular instance of a group, block, or template, etc.; each object is represented in the Work area by an icon, as you can see in the figure. An object is *owned* by another (higher-level) object in the hierarchy if it is connected to the object in the graphic display. (The connection lines tend to run from the bottom/right of the owner object to the

top/left of the owned object.) An ownership connection can be direct (as with the connection of a template to an event) or indirect (as with a block to an event, via a template that is connected directly to the block).

An experiment is designed in PsyScope by building a hierarchy and then filling in the details. You build the hierarchy by adding and linking objects to one another, and you fill in the details by double-clicking on the objects and setting values in the dialogs that pop up.

Some levels of the hierarchy can be left out. For instance, blocks can be connected directly to the experiment, leaving groups out. The only objects that must be explicitly specified are the experiment and events; objects in all of the intervening levels can be omitted (although it is unusual to omit templates or tables).

The hierarchy that is displayed in the Design window defines the experiment conceptually. When the experiment is run, however, it is just a list of trials, each of which executes a set of events. To run each trial, PsyScope consults the experiment hierarchy to decide details about the trial to run:

- Blocks and groups are used to direct a search down the tree to get to a template (or factor table with a built-in template).
- The template is used literally as a “template” to construct a single trial. A trial template is made up of a set of events, each of which specifies a simple occurrence (such as the presentation or removal of a stimulus), a delay period, or the recording of a response.
- All of the events owned by the template will be run in that trial.

PsyScope cycles through these operations until the required number of trials have been executed.

5.2.1.1 Linking Objects

When you create an experiment hierarchy, you must first decide what types of objects to include and in what order to link them together. Here are some basic guidelines:

- You can skip any level of the hierarchy except for the experiment and event objects. Every experiment must have an experiment and an event object.
- An object can only own objects of one particular type (not counting lists). For example, a group that owns blocks cannot also own templates or tables. As an exception, templates and factor tables can be used interchangeably. That is, a given experiment, group or block objects can own both templates and factors.
- An object can be connected to many different owners — even owners of different types. For example, a template could be connected to both a group and a block.
- With the exception of blocks, objects of the same type cannot be linked. For example, you cannot connect a template to another template.

- If there are no blocks in an experiment, then an experiment or group that is linked directly to a template will act like a block.
- If there are no templates or factor tables in an experiment, then any object that is linked directly to an event will act like a template.
- Most experiments will have a template or factor table.
- Events are almost always connected to a template or factor table. Linking an event to other types of objects is only useful for very simple types of experiments, in which the same type of trial will be run repetitively.
- Lists are usually connected to a template or factor table. (See also “5.7.1.8 Factors in the Hierarchy”, p135.)

Advanced Note: Connecting lists to the experiment can be useful for maintaining a consistent item-selection across templates; if the same set of lists is included in two different templates, the item-selections will be independent, so that using an item in one template does not keep the same item from being used in the other template.

5.2.2 Design Window Palettes

There are three palettes at the left of the Design window: the Tools palette, the Objects palette, and the Event palette. The Tools palette contains tools that are used to manipulate objects the Work area. The other two palettes contain object-creator tools.

Only one tool is active at any time; a bold box is drawn around the active tool. To activate a tool in the palette, click on it. To use the current tool, move the cursor to the Work area; the cursor will change into that tool.

The set of tool icons may be too large to fit in the Design window; you can scroll it vertically by clicking on arrows at the top and bottom of the list.

If the **Palettes in separate floating window** option is on (see “7.6.4 Design Options”, p268), the palettes will not appear in the Design window. Instead, they will appear in a separate floating window, and a magnet icon will appear to the left of the Control area. When the magnet is on, the floating window will “stick” to the Design window, always appearing to the left of the active window. If the magnet is off, the floating window can be placed anywhere.

To open or close the Palettes window, select **Palettes** in the **Windows** menu. (Clicking on the magnet icon will also open the Palettes window.) Having the palettes in a floating window does not affect the operation of the tools or object-creator palettes.

5.2.2.1 The Tools Palette

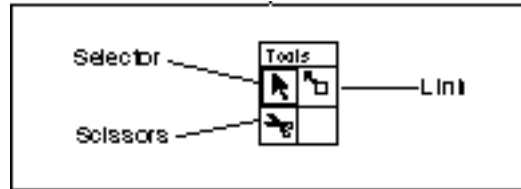


Figure 117 – The Tools palette

Selection Tool



Figure 118 – The selection tool

The *selection tool* functions the same way the as in most other applications:

- To open the dialog for an object - double-click on it.
- To select multiple objects - shift-click the objects or lasso them.
- To change the name of an object (in the same way that you change the name of a file or folder in Finder) - click once on the name or press the Return key once the object is selected.
- Option-double-click on an object to open its dialog and close the Design window.
- Option-click on an object to select it and all the objects it owns.
- Command-click on an object to select it and switch to the link tool (see below).
- Control-click on an object to activate a pop-up menu of operations. For most objects, this menu lets you open the object's dialog, its Attribute dialog, or its Factor Set dialog. For the trash can icon, you can open the trash or empty it.

Link Tool



Figure 119 – The link tool

The *link tool* is used to connect any two compatible objects (see “5.2.1.1 Linking Objects”, p108). To link two objects, click on one object and drag a connecting line to the other. To link multiple objects to a single object, first select the multiple objects, and then drag a connecting line from any of them to the single object.

If you try to link two objects that are incompatible or already connected, you will receive an error message. If the direction in which the link can be applied is ambiguous, the object first clicked on will be the *owned* object and the second will be the *owning* object.

If the Shift key is held down, then clicking on an object with the link tool simply selects the object.

After the link tool is used to connect objects, it changes back to the selection tool automatically.

If some level in the experiment hierarchy had been skipped, but you wish to insert an object of that type (e.g., a block), you can insert the object so that it is in its proper place in the hierarchy, and all objects that were owned by the connected object will be shifted down to the inserted object. For example, to add a block to an experiment that has no blocks — but owns a list of templates directly — create the new block and drag a connection to the experiment; the template connections will be moved to the block automatically.

In some cases, you may want an object to be included in another object multiple times (e.g., including the same block several times in the owning object's list of blocks). You cannot do this in the Design window using the link tool. You must open the owning object's dialog and use **Get [Object]...** from the **Edit** menu.

Scissors Tool



Figure 120 – The scissors tool

The *scissors tool* is used to cut the link between objects, by clicking on the line connecting them. While the mouse button is held down, the link to be cut is hilited. Releasing the mouse button cuts the connection.

Template <-> Table Transform Tool



Figure 121 – The transform tool

The transform tool changes a template into a factor table set, or vice-versa; it is meant to be used when a template-based design needs to be changed to a table-based design. A warning message will be given if some information will possibly be lost in the transformation.

Keyboard Shortcuts

- Space: changes the active tool to the next one in the palette.
- Shift-Space: changes the active tool to the preceding one in the palette.
- Period (.) or Command-.: changes to the selection tool.

The Objects Palette

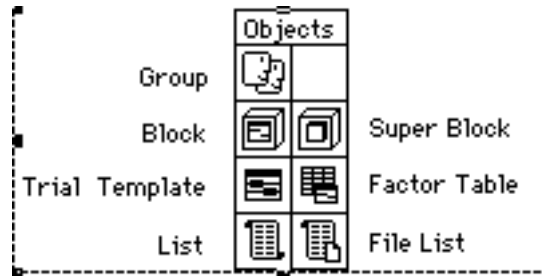


Figure 122 – The Objects palette

The Objects palette is used to create new (non-event) objects. You can create a new object by two methods:

- Drag an object from the palette into the Work area.
- Select an object from the palette and click somewhere in the Work area to drop the object there.

Using the latter method, you can create multiple objects by holding down the Shift key when dropping an object.

If the **Always ask for new object names** Design option is on (see “7.6.4 Design Options”, p268), a dialog will appear asking for the object name whenever you create a new object.

5.2.2.2 The Events Palette



Figure 123 – The Events palette

The Events palette functions just like the Objects palette. The above figure shows the standard Events palette, but you can expand the list of available event types by adding Extensions to PsyScope.

5.2.3 Design Window Work Area

The Work area of the Design window (see “Figure 116 – Anatomy of the Design window”, p107) displays the current structure of the experiment in hierarchical form. The hierarchy may flow top-to-bottom or left-to-right, depending on the **Use horizontal arrangement for structure display** Design option (see “7.6.4 Design Options”, p268).

You can move objects around the Work area using the selection tool (see “Selection Tool”, p110).

5.2.3.1 Cleaning Up

The **Clean Up** item in the **Design** menu repositions all the objects in the Work area, putting them into a standard arrangement. You can undo a **Clean Up** by selecting **Undo** from the **Edit** menu.

5.2.3.2 Trash

To throw away an object that is in the Work area, drag it to the trash can. Objects are not permanently deleted until the trash is emptied. The trash can bulges to show that some items have been thrown away (but not permanently deleted), just like the Finder’s trash can.

To fix the trash can in the lower right hand corner, select the **Trash always in lower right corner** Design option (see “7.6.4 Design Options”, p268). Otherwise, you can move the trash can around the Work area just like any other object.

To see a list of objects currently in the trash can, double-click on the trash can. From this list, you can remove an object from the trash. (See also “5.2.9 View Trash Dialog”, p115.)

When you delete an object from anywhere in the graphic environment, it is thrown into the trash. If you try to name an object and you are told that the name is already in use, it could be because an object in the trash has the same name. Emptying the trash will get rid of the object and allow you to use the name.

5.2.4 Design Window Control Area

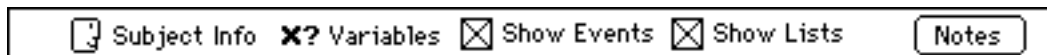


Figure 124 – The Design window Control area

Clicking on the **Subject Info** icon opens the Subject Info dialog, described in “6.2.1.2 Subject Info Dialog”, p226.

Clicking on the **Variables** icon opens the Variables dialog, described in “5.10.5 Trial Manager Variables Dialog”, p208.

The **Show Events** checkbox controls whether events are shown as part of the experiment hierarchy in the Work area. If this checkbox is off, events are hidden from the Work area in the Design window, although they can still be accessed through the Template window. Use this to remove clutter in the Design window when there are many objects.

The **Show Lists** checkbox works like the **Show Events** checkbox, controlling whether lists are displayed in the Work area. If this checkbox is off, lists are hidden from the Work area, but can still be accessed through the **Lists** button in Experiment, Group, and Block dialogs and Template windows by `ctl_clicking` on an object.

Clicking on the **Notes** button opens a standard text-editing window where you can enter notes about your experiment design. These notes are saved in the script. (Separate notes are saved for each experiment within one script.)

5.2.5 New Object Name Dialog

When you create a new object, you can either name the object yourself or have PsyScope assign a default name. If the **Always ask for new object name** option is off (see “7.6.4 Design Options”, p268), the object will get a default name; if the option is on, the New Object Name dialog will appear.

To name a new object, type the name into the text field of the New Object Name dialog. When you hit **OK**, the validity of the new name is checked; the name is rejected if it is already being used, if it is the same as a special keyword (“this”, “start”, “end”, “owner”, “last”, “none”, “force_one”, “force_all”, “factor_format_default”, “factorstructure”, “omit”, or any condition or action name), or if it contains certain symbols (#, :, quotes, or leading spaces).

Hitting **Cancel** cancels the creation of a new object.

In some cases, a pop-up menu appears below the text field from the name so that the object can be further specified. For example, when a new event is created, you can set the event’s type through the pop-up menu.

5.2.6 The Palettes Window

Palettes are usually embedded in the Design and Trial Template windows, but you can use a floating palette window by turning on the **Palettes in separate floating window** Design option (see “7.6.4 Design Options”, p268).

As a separate floating window, the palettes work just as when they are part of the Design window. Tools that do not apply to the current window will be grayed out.

To open and close the floating palette window, select **Palettes** from the **Windows** menu. If the frontmost window is not the Design or Trial Template window, the Palettes window will be automatically hidden.

Windows that use the Palette floating window will have a magnet icon in the top left corner of the window. Clicking on the magnet icon causes the palette to “stick” to the top left of the active window. Clicking on the magnet again leaves the palette in place when the window is moved or the active window changes.

5.2.7 Object List Dialog

The **Design** menu contains an **Objects in Script** submenu. Selecting one of the object types in the submenu opens an Object List dialog, which lists all the known objects of that type in the script including those in the trash. Double-clicking on an untrashed object opens the window for that object.

A trash can next to the object indicates that it is in the trash. Trashed objects cannot be opened. If you wish to open this object, you must first remove it from the trash (see <ref> above).

5.2.8 Get Object Dialog

This dialog is a (modal) version of the Object List dialog — which is used when objects are linked through dialogs instead of through the Design window. When you select **Get [Object]...** (or **Import List...**) from the **Edit** menu, the Get Object dialog is opened.

In the Get Object dialog, select the object that you want to get and hit the **Select** button. You can select an object that is in the trash, and it will be removed from the trash. To cancel the **Get...** command, click on **Cancel**.

5.2.9 View Trash Dialog

To open the View Trash dialog double-click on the trash can icon in the Design window or select **View Trash** from the **Design** menu. The View Trash dialog shows a list of all objects currently in the trash.

To remove an object from the trash, double-click on it; removed objects will appear in the Work area of the Design window. If you recover an object that owns other objects in the trash, the owned objects will be recovered, as well.

To permanently delete trashed objects from the script click on the **Empty** button in the View Trash dialog, or select **Empty Trash** from the **Design** menu, or .

5.3 The Experiment Object

The *experiment* object is the starting place for building an experiment; it represents the entire experiment, and it owns all other objects.

The attributes associated with the experiment object control general features of the experiment (e.g., the instructions file, the background color for the screen, etc.; these are described in “5.8.3.4 Standard Experiment Attributes”, p161).

5.3.1 Connecting Objects to the Experiment

Groups, blocks, templates, factor tables, or events can be connected to the experiment object. If groups are to be used at all in the experiment, they must be connected to the experiment object. See also “5.2.1.1 Linking Objects”, p108.

If groups are connected to the experiment, then only one of the groups — the *current group* — will be used for each run of the experiment. The current group can be set directly through the Experiment dialog (see “5.3.2 Experiment Dialog”, p116), or automatically using the Subject Info facilities (see “6.2 Subject Info”, p224). The trials that are presented during a run will be created only from templates and events owned by the current group.

If no groups are specified, the dialog used for the experiment object (i.e., the dialog that is opened by double-clicking on the experiment in the Design window) will depend on whether blocks, templates, factor tables, or events are connected to the experiment (see “5.3.2 Experiment Dialog”, p116). The Attributes dialog for the experiment will always be the same.

Lists can be connected directly to the experiment (regardless of whatever else is connected to it); these lists will be available to all objects belonging to the experiment. Lists are usually associated with a particular template. (Lists are described in “5.7.1.2 Lists”, p130.)

Advanced Note: Connecting lists to the experiment can be useful for maintaining a consistent item-selection across templates; if the same set of lists is included in two different templates, the item-selections will be independent, so that using an item in one template does not keep the same item from being used in the other template.

5.3.2 Experiment Dialog

To open the dialog for an experiment object, double-click on the experiment icon. The actual dialog you see depends on what objects are connected to the experiment. There are four possible cases:

- Groups are connected to the experiment

- Blocks are connected to the experiment
- Templates and/or Factor Tables are connected to the experiment
- Events are connected to the experiment.

This section will describe the dialog opened in the first case; this is called the *Experiment dialog*. In the second case, the experiment acts as a group (see “5.4.2 Group dialog”, p119). In the third case, the experiment acts as a block (see “5.5.2 Block Dialog”, p121). In the final case experiment acts as a template (see “5.6.1 The Trial Template Window”, p124).

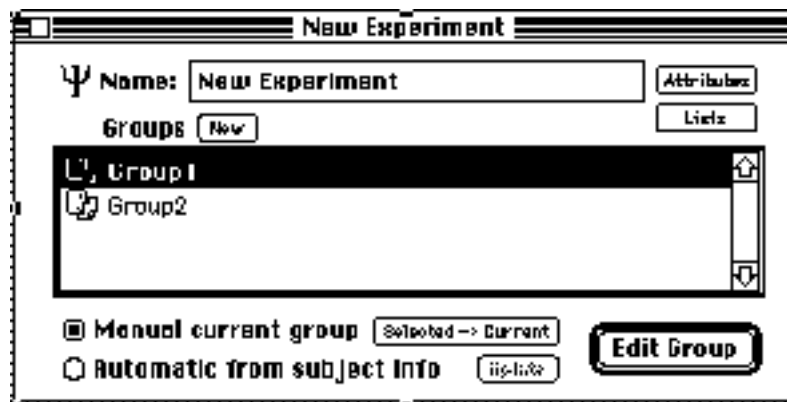


Figure 125 – The Experiment dialog

At the top of the Experiment dialog is a text box with the experiment name. To change the experiment name, type your changes in the text box. The name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a specified delay (see “7.6.4 Design Options”, p268).

The **Attributes** button opens the Experiment Attributes dialog (see “5.8.3 Experiment Attributes”, p158).

The **Lists** button opens the Factor Set dialog for the experiment (see “5.7.3.3 Factor Set Dialog”, p147).

The main body of the Experiment dialog contains a list of the groups that are in the experiment. One of the group names should appear in bold: that group is the current group (see “5.3.1 Connecting Objects to the Experiment”, p116). The current group will be the one used if the experiment is run at this point.

To create new groups and add them to the experiment, click on the **New** button above the list. To include groups that already exist, select **Get Group...** from the **Edit** menu (see “5.2.8 Get Object Dialog”, p115).

To open the dialog for a group in the list, select it and click on the **Edit Group** button (see “5.4.2 Group dialog”, p119). Groups can be re-arranged in the list by dragging them, but the order of groups is irrelevant for executing the experiment.

The radio buttons at the bottom of the dialog determine how the current group is chosen. If **Manual current group** is on, then the current group will never be changed automatically; in this case, the current group is changed by pressing/chicking on the **Selected -> Current** button.

If **Automatic from subject info** is on, then the current group will be selected automatically, based on Subject Info values. You must set up grouping information prior to using this setting (see “6.2.4 Automatic Grouping”, p233). When the subject information changes, the current group will not usually change until the group criteria are evaluated(usually when the experiment is run). Pressing/chicking on the **Update** button forces the automatic group selection to occur immediately.

5.4 Groups

The *group* object is used to specify a part of the experiment that is to be seen only by a particular set of subjects. The way in which a subject is associated to a group is discussed in “6.2.4 Automatic Grouping”, p233. Only one group is used in a given run of the experiment (see “5.3.1 Connecting Objects to the Experiment”, p116), and only the blocks, templates and events owned by that group will be used to create trials.

5.4.1 Connecting Objects to a Group

Blocks, templates, factor tables, or events can be connected to a group.

If blocks are connected to a group, then all of the blocks within the group will be executed. This is called the group’s **Block List**. The number of *trials* executed within these blocks is determined by the trial count value of each block, the cycle count value of the group, and the block-scaling value of the group. See “5.12.2 Trial Counting”, p211 for more details.

Within one execution of the block list, blocks are selected according to a **block order** — **Sequential, Random, or Random with Replacement**. For example, blocks ordered by **Sequential** are executed in the order which they are listed in the Group dialog. See “5.12.1 List Ordering”, p210 for more information on block order.

If no blocks are specified, the dialog used for the group object (i.e., the dialog that is opened by double-clicking on the group in the Design window) will depend on whether templates, factor tables, or events are connected to the group (see “5.4.2 Group dialog”, p119). The Attributes dialog for the group will always be the same.

List objects can be connected to a group (regardless of whatever else is connected to the group), but this is unusual. (Lists are described in “5.7.1.2 Lists”, p130.) Lists that are connected to the group are available to all blocks, templates, and events owned by the group.

5.4.2 Group dialog

To open a dialog for the group, double-click on the group icon in the Design window. The actual dialog you see will depend on which of these cases apply:

- Blocks are connected to the group.
- Templates are connected to the group.
- Events are connected to the group.

This section will describe the dialog presented in the first case; this is called the *Group dialog*. In the second case, the group acts as a block (see “5.5.2 Block Dialog”, p121). In the final case, the group acts as a template (see “5.6.1 The Trial Template Window”, p124).

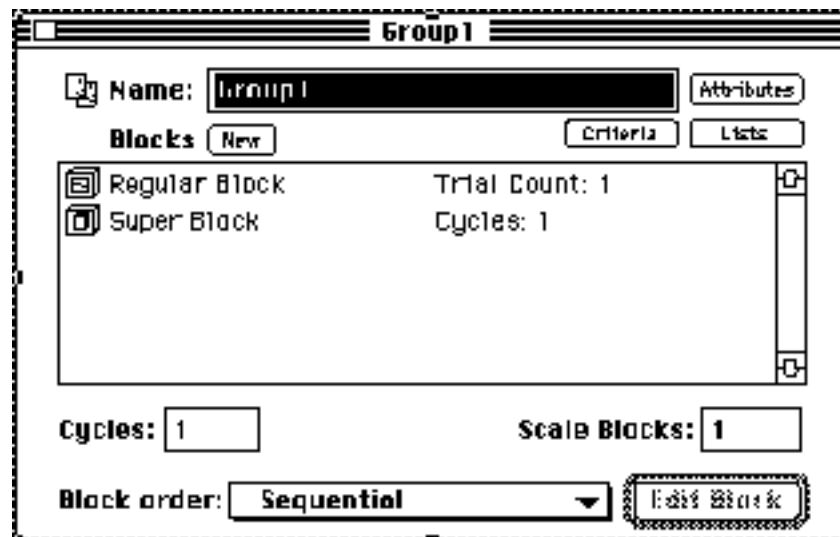


Figure 126 – The Group dialog

At the top of the Group dialog is a text box with the group name. To change the group name, type your changes in the text box. The name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a specified delay (see “7.6.4 Design Options”, p268).

The **Attributes** button opens the Group Attributes dialog (see “5.8.4.3 Group Attributes Dialog”, p170). If the Group dialog was actually opened for an experiment, the Experiment Attributes dialog will be opened instead (see “5.8.3 Experiment Attributes”, p158).

The **Lists** button opens the Factor Set dialog for the group (see “5.7.3.3 Factor Set Dialog”, p147).

The **Criteria** button opens the Group Criteria dialog for setting automatic group assignment parameters (see “6.2.4.2 Group Criteria Dialog”, p235).

The main body of the Group dialog contains a list of the blocks that are in the group. To open the dialog for a block selected in the list, click on the **Edit Block** button (see “5.5.2 Block Dialog”, p121).

To create new blocks and add them to the group, click on the **New** button above the list. To include Blocks that already exist, select **Get Block...** from the **Edit** menu (see “5.2.8 Get Object Dialog”, p115). You can also include blocks that are already within the group.

The order in which the blocks will be run is controlled by the **Block Order** menu; for information about the choices in this menu, see “5.12.1 List Ordering”, p210 and “5.4.1 Connecting Objects to a Group”, p118 for information on block selection.

To set the number of times the entire list of blocks will be executed within the group, type the number in the **Cycles** box. The **Scale Blocks** box sets a scaling factor that is applied to all of the scalable block counts. (See “5.12.2 Trial Counting”, p211 on trial counting.)

5.5 Blocks

A *block* object describes a set of trials that are to be run together within the experiment. The trials can be of all derived from the same template or factor table, or from different templates and factor tables.

5.5.1 Connecting Objects to Blocks

Block objects typically own one or more templates and factor tables. (Trial templates are discussed in more detail in “5.6 Trials and Templates”, p124; factor tables are discussed in “5.7.1.1 Factor Table Sets”, p130). One run of a block consists of executing a series of trials, each drawn from the list of templates and factor tables connected to the block.

The number of trials executed within a block depends on the *trial count* value or *block duration* set for the block object, and possibly the block-scaling value of the block object’s owner. (See “5.12.2 Trial Counting”, p211 for trial count details.)

For each trial, only one template or factor table is used. The way in which templates are selected depends on the *template order* and *template weight* that are set for the block object. A template weight does not guarantee that it will be run the weighted number of times; templates are selected according to the access type until the required number of trials have been run (or the time is exhausted for blocks with a duration); weights in a random design essentially control the probability distribution for selecting a template, unless there are enough trials to exhaust the weights. (See “5.12.1 List Ordering”, p210 on accessing modes and weights.)

5.5.1.1 Connecting Events to a Block

Events can be connected directly to a block, although this is unusual. In this case, the block object takes on the role of a template, as well as a block. Except for attribute-setting purposes, the block object will behave exactly like a template.

5.5.1.2 Connecting Blocks to a Block

Instead of owning a list of trial templates and factor tables, a block can be used to own other blocks; a block used this way is a *superblock*.

The purpose of a superblock is to run a set of blocks together. The owned blocks will specify some set of trials to be run together; the superblock thus provides a higher level of trial grouping.

One run of a superblock will run all of the blocks it owns in the same way that all of the blocks of a group are executed. Like groups, a superblock has a cycles count and block-scaling value. (See “5.12.2 Trial Counting”, p211 for more details.)

The superblock structure is recursive; i.e., superblocks can own other superblocks, ad infinitum. While superblocks cannot contain a mixture of blocks and trials, they can contain a mixture of blocks and other superblocks.

5.5.1.3 Connecting Lists to Blocks

List objects can be connected to a block (regardless of whatever else is connected to the block). Lists that are connected to the block are available to all templates and events belonging to the block. (Lists are described in “5.7.1.2 Lists”, p130.)

Connecting lists to a block can be useful for maintaining a consistent item-selection across templates owned by the block; even if a set of lists is included in two different templates, the item-selections will be independent, so that using an item in one template does not keep the same item from being used in the other template.

5.5.2 Block Dialog

To open a dialog for a block object, double-click on the block icon in the Design window. The actual dialog you see will depend on which of these cases apply:

- Templates and/or factor tables are connected to the block.
- Events are connected to the block.

This section will describe the dialog presented in the first case; this is called the *Block dialog*. In the second case, the block acts as a template (see “5.6.1 The Trial Template Window”, p124).

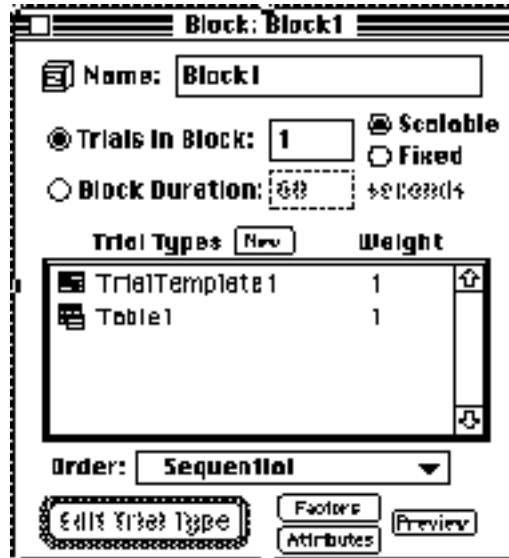


Figure 127 – The Block dialog

At the top of the Block dialog is a text field that contains the name of the block. To change the block name, edit the text in the box; the name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a delay (see “7.6.4 Design Options”, p268).

The main body of the Block dialog contains a list of templates and factor tables that are owned by the block. To open the dialog for the template or factor table selected in the list, click the **Edit Template** button (see “5.6.1 The Trial Template Window”, p124 and “5.7.2.1 The Factor Table Window”, p138).

To create new templates and add them to the block, click on the **New** button above the list. To include templates and factor tables that already exist, select **Get Template...** from the **Edit** menu (see “5.2.8 Get Object Dialog”, p115). You can also include a template multiple times using **Get Template....**

The **Order** menu chooses the template order, i.e., the order in which the templates will be run within the block; see “5.12.1 List Ordering”, p210 for information about the choices in this menu, and “5.5.1 Connecting Objects to Blocks”, p120 for information on template selection.

You can change the weight for a template by clicking under the **Weight** column in the template list. To enter a new weight, type it into the text box that will appear, and then hit Return.

Trials in Block and **Block Duration** are mutually exclusive items that determine the number of trials to run in the block. When **Trials in Block** is selected, the number in the text field is the number of trials that will be run in that block. If **Scalable** is set, the number of trials is scaled by a factor from the block's owner (see "5.12.2 Trial Counting", p211 on trial counting). When **Block Duration** is selected, the number in the text field is the length of time that the block will run (see "5.5.1 Connecting Objects to Blocks", p120); this value is never scalable.

The **Attributes** button opens the Blocks Attributes dialog (see "5.8.5.3 Block Attributes Dialog", p172). If the dialog was actually opened for the experiment, the Experiment Attributes dialog is opened instead (see "5.8.3 Experiment Attributes", p158). Or, if the dialog was opened for a group, the Group Attributes dialog is opened (see "5.8.4.3 Group Attributes Dialog", p170).

The **Lists** button opens the Factor Set dialog for the block (see "5.7.3.3 Factor Set Dialog", p147).

To run a single execution of the block (in practice mode) to check your design, click the **Preview** button. For **Preview** to work, the block must be connected to the experiment. (See also "5.11 Trial Chooser Floating Window", p209).

5.5.3 Superblock Dialog

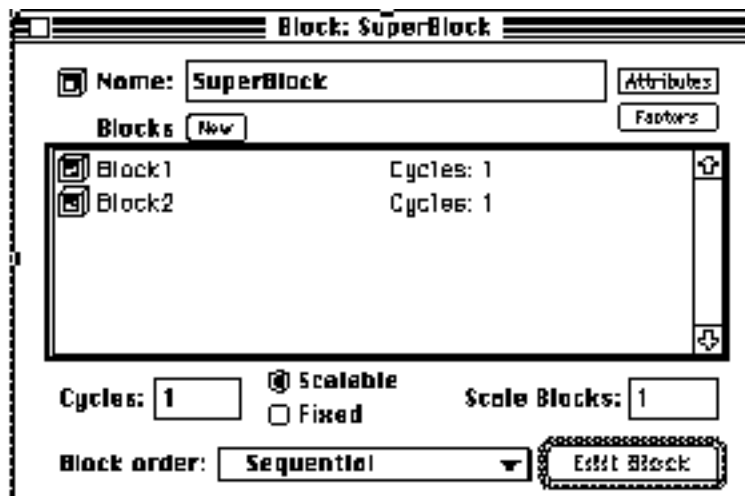


Figure 128 – The Superblock dialog

The Superblock dialog is almost identical to the Group dialog, described in "5.4.2 Group dialog", p119. The only addition is the **Scalable/Fixed** button pair, which controls whether the cycle count can be scaled by the superblock's owner. (See "5.12.2 Trial Counting", p211.)

5.6 Trials and Templates

PsyScope is based on the usual notion of a trial in psychology experiments: a trial is the smallest unit of repetition used to collect data in an experiment.

In PsyScope, a *trial* is defined as a sequence of events, all of which occur within a specific time frame. This time frame is controlled precisely using a millisecond clock (if the CMU button box is used, or a 17 millisecond clock if none is available). At the start of each trial, the clock is reset. If you want to control the relative timing of two events precisely, then both events must be in the same trial.

Not all trials in an experiment are identical. Usually, you will want to vary aspects of the trial — such as stimuli, delay periods, or possible responses — from trial to trial. Nevertheless, the trials of an experiment usually share a common structure (e.g., present one stimulus, wait for a period, present a second stimulus, then collect a response), with only the details (for example, the specific stimuli presented or the duration of a delay) varying from trial to trial.

The basic structure of a trial is represented by a *template*. In many experiments, all trials share the same basic structure, so only one template is needed. In general, you will use a separate template for each type of trial that varies greatly in structure from the others.

For example, suppose you wanted to design an experiment with the following structure: in one type of trial, you want to measure simple responses to a stimulus; in a second type of trial you want to present a question that will test the subject's comprehension. To do this, you would probably use two different templates — one for the stimulus trials, and a second one for the comprehension trials.

Factor table set objects (see “5.7.1.1 Factor Table Sets”, p130) automatically include a built-in template. The above description of templates applies to this built-in template.

5.6.1 The Trial Template Window

The Template window allows you to edit the sequence of events that will take place during a trial.

At the top of the window, there is a text box for changing the name of the template, and buttons for setting template attributes, controlling lists, and running a preview of the template. At the left of the window are the standard tool and event palettes. The rest of the win-

dow is divided into three areas: the Event Name area, the Timeline area, and the Event Status area.

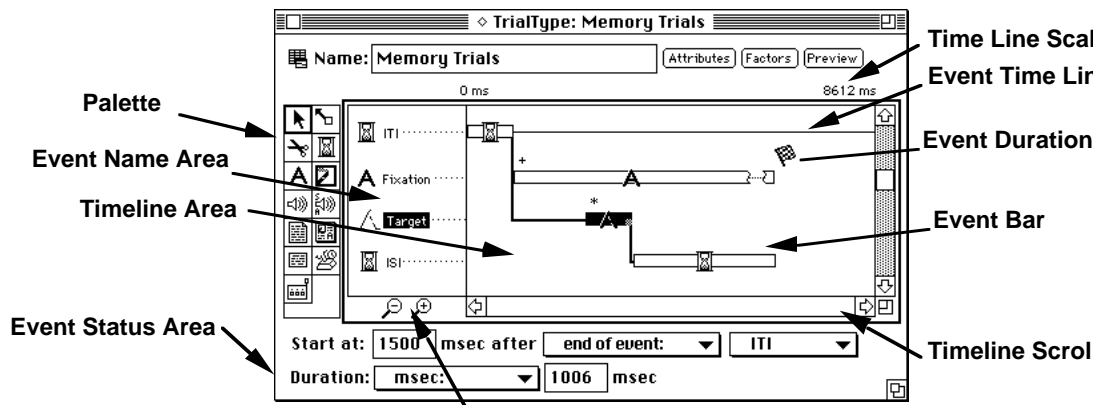


Figure 129 – Anatomy of the Template window

5.6.1.1 Template Name and Buttons

To change the template name, type your changes in the text box at the top of the dialog; the name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a short delay (see “7.6.4 Design Options”, p268).

The **Attributes** button opens the Template Attributes dialog. If the dialog was actually opened for the experiment, the Experiment Attributes dialog is opened instead (see “5.8.3 Experiment Attributes”, p158). Or, if the dialog was opened for a group, the Group Attributes dialog is opened (see “5.8.4.3 Group Attributes Dialog”, p170). Finally, if the dialog was opened for a block, the Block Attributes dialog is opened (see “5.8.5.3 Block Attributes Dialog”, p172).

The **Lists** button opens the Factor Set dialog for the template (see “5.7.3.3 Factor Set Dialog”, p147).

The **Preview** button runs a single trial from the template (in practice mode) so that the template design may be checked. The template must be connected to the experiment for this to work. (See also “5.11 Trial Chooser Floating Window”, p209.)

5.6.1.2 Palettes

The Tools and Events palettes work in much the same way as they do in the Design window. As with the Design window, the palettes may be in a separate window if the **Palettes in separate floating window** Design option is on (see “7.6.4 Design Options”, p268).

To create a new event, select an event type in the palette and drag or drop it into the Event Name area (see below). The link and scissors tools are used in the Timeline area to adjust start references.

5.6.1.3 Event Name Area

In the Event Name area of the dialog, all of the events owned by the template are shown in an area like the work area of the Design window. You can move the event icons in this area, and their corresponding timelines to the right will follow them. To expand the event name area, click on the line that separates the event name area from the timeline area and drag it. (The cursor will change to a special dragging tool when it is over the dividing line.)

Events are created by dragging or dropping an event type from the palette, or by selecting **New Event...** in the **Edit** menu. To add existing events to the template, select **Get Event...** in the **Edit** menu. You cannot include the same event multiple times.

Double-clicking on an icon in the event name area opens the Event Attributes dialog for the event (see “5.8.7 Event Attributes”, p176). Unfortunately, you cannot edit Event names in this area.

5.6.1.4 Timeline Area

Each event displayed in the Event Name area has a *timeline* to the right, in the Timeline area; this timeline is a graphical representation that shows the timing and terminating conditions of each event within the template.

The timeline scale in milliseconds is shown above the timeline area. If the selected event has a known start time, then the start and ending times of the visible area are shown; otherwise, only the duration of the visible area is shown. The scale can be changed using the “+” and “-” magnifying glass icons that are below the event name area.

The scroll bar along the bottom of the timeline controls the “window” of time that is shown in the Timeline area. When the thumb on the scroll bar is all the way to the left, the line dividing the Timeline area from Event Name area marks the beginning of the trial.

Timeline

The *timeline* is the “line” that the event bar sits on; this line is visible if the event has an exact start time, or invisible if the event’s start time is unscheduled or depends on some external input.

Timelines can be moved up and down by Shift-clicking on the event’s bar and then dragging it up or down. (This is effectively the same as moving the event’s icon vertically in the event name area.)

Event Bars

Each event in the timeline area has an *event bar*. The event bar represents the time that the event is “on” within the trial.

The stimulus value for an event is shown above its time bar. “???” means that a stimulus value has not been specified (though possibly only in the current context; see “5.7.2.1 The Factor Table Window”, p138).

Double-clicking on an event bar performs the same operation as double-clicking on the icon in the event name area: the Event Attributes dialog is opened.

Scheduling Dependencies

A **scheduling dependency** means that the start of one event depends on the start or end of another event. The time that elapses between the start/end of one event and the start of the dependent event is called an **offset**.

If the start time of an event is dependent on another event, a dependency line is drawn in the Timeline area from the start of the dependent event's time bar to the start or end of the other event's time bar.

To increase or decrease the offset time for an event, drag the event's time bar horizontally. The offset will be shown interactively in the Event Status area (see below).

Unscheduled events are shown close to the start of the trial without a visible time line. You cannot drag unscheduled events horizontally. (Unscheduled events must be started with the `RunEvent []` action; otherwise, they will never be executed.)

You can change scheduling dependencies in the Timeline area. To link one event to the start of another event, select the link tool and click anywhere on the bar for the first event. Then, to link the event's start time relative to the start of another event, click in the left half of the relative event's time bar; to link relative to the end of the event, click in the right half of the time bar. Command-clicking an event bar highlights the bar and invokes the link tool.

To remove a scheduling dependency, use the scissors tool to cut the line connecting the event bars. The dependent event will be changed to start at the beginning of the trial.

Scheduling dependencies can also be controlled in the Event Status area (see below).

Event Duration

An event's duration is represented roughly by the length of the event's time bar. A solid bar is used when the event has a fixed duration; a "broken" bar is used when the event's duration depends on external input, so that the exact end time is unknown. A broken bar duration is usually accompanied by a list of icons that represent the conditions for terminating the event. (See "5.9.2 Conditions Dialog", p194.)

To change the length of the bar, drag the lower right corner of the bar — just like resizing a window. For events with a timed duration, changing the length of the bar changes the actual duration of the event. For events that are terminated by an event, changing the length simply changes the display size of the bar.

Clicking on a terminating condition icon or option-clicking in the duration-resize area displays a pop-up menu; this menu is the same as the menu in the Event Status area (see below), and can be used to quickly change the duration type of the event.

Event durations can also be controlled in the Event Status area (see below).

5.6.1.5 Event Status Area

When an event is selected, information about the start and duration of the event is displayed at the bottom of the Template window. You can use the value boxes and menus in this area to change the starting time and duration of an event.

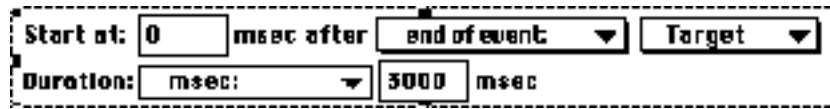


Figure 130 – The Event Status area

Start at

The number in the **Start at** box determines an event’s offset; i.e., how long, in milliseconds, to wait after the starting dependency before starting the selected event. This value is reflected in the Timeline area by an offset from the start of the trial, or an offset from the beginning or end of another event on which it depends.

The menus that follow **msec after** are used to set the starting dependency. The starting time can be relative one of the following:

- start of the trial
- start of another event
- end of another event

The event can also be unscheduled (**no auto-start**). An unscheduled event will not be executed unless it is started by a `RunEvent[]` action; see “5.9.4.1 Available Actions”, p198.

The scheduling dependencies will be represented in the Timeline area with links between event time bars (see above).

*Note: You are not allowed to set an event to start at the end of another event whose duration has been set to **End of Trial** (see below for the meaning of this duration value).*

Duration

The **Duration** pop-up menu is used to control how long the selected event will last. For more information on duration types, see “Duration”, p179. Two of the duration types — **msec:** and **Conditions** — require further parameterization.

If an absolute duration has been assigned to the event (with **msec:**), you can edit the duration directly by entering a new value in the text box. This value is immediately reflected in the event’s time bar. Likewise, if you make changes in the event’s time bar, they will be reflected in this value.

If an event has a list of terminating conditions (with **Conditions**), icons for the terminating conditions are listed. You cannot edit the terminating conditions directly. To open the Conditions dialog, however; select **Conditions** from the **Duration** menu (see “5.9.2 Conditions Dialog”, p194).

To vary the duration of an event in more complex ways, use the **Duration** attribute in the Event Attributes dialog. The **Duration** menu may be missing if the duration is varied in this way, replaced by information on how the duration is varied.

5.7 Factors and Lists

5.7.1 Definitions

A *factor* is an independent variable that controls parameters of an experiment. Each factor is made up of *levels*; the levels represent — abstractly — the possible values for parameters controlled by the factor. For example, A factor that controls the color of a text stimulus might have levels “Red”, “Green”, and “Blue”.

Factors are grouped together in *factor sets*; any number of factors can be in a factor set. All of the factors in a factor set are *crossed* with each other to produce *cells*. Each cell corresponds to a unique permutation of the factors’ levels, where one level is used from each factor in the set.

For example, consider a factor set made up of a Size factor — with levels “Big” and “Small” — and a Color factor — with levels “Red” and “Blue”. There are four cells in this factor set, represented by the combinations “Big Red”, “Small Red”, “Big Blue”, and “Small Blue”.

The usual form of a factor set is the *factor table*. When a factor is in a table, the levels look like rows or columns in the table. The cells of the table are the boxes of the grid created by intersecting all of the level rows and columns. Factor tables are handled by the Factor Table window (see “5.7.2.1 The Factor Table Window”, p138).

		Size	
Color	Font	Big	Small
Red	Chicago	1	1
	Helvetica	1	1
Blue	Chicago	1	1
	Helvetica	1	1

Figure 131 – Example factor table

Factor sets can also be created by crossing the items of *lists* (see “5.7.1.2 Lists”, p130). There is no graphic representation of the set’s cells in this case. Factor sets made from lists are controlled through the Factor Set dialog (see “5.7.3.3 Factor Set Dialog”, p147).

Factor sets control experiment parameters by acting on templates and events. For each trial in the experiment, a **current cell** is selected from each factor set. (See “5.7.2.2 Table Info Dialog”, p140 for information on how the current cell is selected.) The current cell provides special context to templates and events, which can be configured to use different attribute values dependent on this factor context.

Any number of factor sets can act on a template. In the case of tables, both the tables and the template are built into a single object called a factor table set (see “5.7.1.1 Factor Table Sets”, p130). For list-based factor sets, the lists are connected to some object and then grouped into sets within the object.

The **condition** of a trial is the collection of current cells from all of the factor sets acting on the trial. In an experiment with a single factor set, the each cell corresponds to a condition of the experiment.

Note: “Condition” has been defined here with its strongest meaning, also called the “trial condition”. The term “set condition” can be used to mean “cell”, and “factor condition” to mean “level”. Further, the term “condition” is used to refer to a collection of states used to trigger an action during the running of the experiment.

5.7.1.1 Factor Table Sets

Factors and their levels are usually created as elements within a **factor table set** (which is not to be confused with *factor set*). A factor table set owns one or more built-in tables, and one built-in trial template which its tables will act on (see “5.6 Trials and Templates”, p124).

In the experiment hierarchy, a factor table set object can be used interchangeably with a template object.

The tables of a factor table set are shown graphically in the Factor Table window. Each cell in this graphic table corresponds to an actual table cell. The Factor Table window is used by selecting cells and setting the values of attributes — usually event attributes — which depend on the selected cells. See “5.7.2.1 The Factor Table Window”, p138 for more information.

5.7.1.2 Lists

Factors are usually created as elements in a factor table, but they can also be created as independent objects. Factors of this type are called **lists**, and their levels are called **items**. Lists work differently from factors that are in a table; instead of setting attributes in the context of cells selected from a crossing of lists, lists contain physical values for each item, and attributes of other objects are explicitly linked to these values using **Vary by List**.

Lists are defined using **fields**. A field represents a type of value that is to be stored in the list; there is usually a one-to-one mapping between fields and the attributes elsewhere in

the experiment that are to vary by the list. While the items of a list represent different values abstractly, a concrete value is assigned to the field for each item.

As an example, consider implementing a Size factor as a list. The Size list would have one field, “Text Size”, and two items: “Big” and “Small”. A value of 24 (points) could be assigned to the “Text Size” field for the “Big” item, and 9 (points) for the “Small” item. Then, the **Size** attribute of the event that displays the text would be set to **Vary by List** using the “Text Size” field.

A **linear list** is a list that is constrained to have a single field. The Size list in the above example could have been implemented using a linear list.

A single list can belong to any number of objects; usually, lists are owned by templates or factor table sets. Within the owner objects, lists are grouped together in factor sets (see “5.7 Factors and Lists”, p129) through the owning object’s Factor Set dialog (see “5.7.3.3 Factor Set Dialog”, p147).

5.7.1.3 List Files

A **list file** is a list that reads its item values from a separate file. A list file is defined by choosing the file containing the items and specifying the number and type of the **columns** of items in the file. Each column in the file will correspond to one field of the list created from the file. A list file object is used just like a list object.

The content of the file should be item values, in the same form as strings in PsyScript: space-delimited or quoted. (Tabs, carriage returns, and newlines count as spaces.) All of the items must be literal strings, unless the file is specially defined to have non-literals (see below). (Regular list files cannot be used for non-literal or multiple-token values, e.g., duration, actions.) List files can be viewed, but not edited through the usual List dialog (see “5.7.3.2 List File Dialog”, p146).

The file that is read for the list does not have to have items arranged in an actual columnar format, since spaces are ignored. For instance, if a file is defined to have two columns, every other token will be placed in the first column, regardless of the spatial arrangement of the tokens within the file.

List Files with Non-literals

If a list file has a column type that expects non-literal or multiple-token values, the column in the file must be specially formatted to allow the values to be read in. Column types that need this formatting are indicated in the List File dialog (see “5.7.3.2 List File Dialog”, p146) with an asterisk (*) next to the field name.

A list file set up to handle special columns must have the `#NonLiteral` identifier at the very beginning of the file (no preceding characters are allowed). For the special column(s), you must add an extra set of square brackets around the values in the file; this lets the file-reader know how to group the tokens, in case multiple tokens are assigned as a single value.

5.7.1.4 Expanded Lists

An *expanded list* is a type of list in which the items are free objects. An expanded list is typically used as a factor, and the free items are therefore called *free levels*. The extra structure of an expanded list allows other lists or factors to be nested within its levels (see “5.7.1.5 Nested Factors”, p132).

Few experiment designs will use expanded lists. They are provided because the expanded form can sometimes be more convenient for PsyScript users.

5.7.1.5 Nested Factors

A *nested factor* is a factor (in a table or as a list) that is relevant only when a particular level in another factor is current. The level in which the factor is nested is referred to as the *owning level*. (See “5.7.3.1 List Dialog”, p145 and “5.7.2.1 The Factor Table Window”, p138 for information about creating nested factors.)

Nested factors owned by a particular level are grouped into list sets just like regular factors and lists. (See “5.7.1.1 Factor Table Sets”, p130 for more information.)

In most respects, nested factors act the same way as regular factors and lists. The main difference is that a new cell is selected for a nested set only when the owning level is current. Also, nested factors owned by free levels can be linked to only within the owning level (see “5.7.1.4 Expanded Lists”, p132).

5.7.1.6 Lists in a Factor table

Sometimes it is useful to define a factor as free list object to conveniently create a long list of values (or to read the values from a file); however, you may then want to cross this list with other factors that are defined in a factor table set.

The grouping of lists into sets is usually handled in the Factor Set dialog, but this dialog does not allow you to change the tables that are built into a factor table set. To add a list to a factor table, drag a connection from the list to the table set object in the Design window; the Link List dialog will ask in which table you wish to add the list. Alternately, you can open the Factor Table window and choose **Import List...** from the **Edit** menu.

When a list is imported into a factor table, the items of the list will not be individually accessible in the table (like the levels of a factor in the table), and you will still have to explicitly link attributes to the list in the usual way.

5.7.1.7 Level Order and Crossing Types

For each trial in the experiment, one cell is selected in each table (or list set) and is designated the current cell of the table for that trial. The way in which the current cell is selected depends on:

- The *crossing type* of the table

- The *access type* of the table
- The *weights* of cells in the table
- The *scope* of the table

The crossing type of a table specifies the high-level design — which cells of the table are available for a given experiment and when the cell should be changed. The available crossing types are defined in “5.7.2.2 Table Info Dialog”, p140.

The access type of a table specifies the order in which cells are selected among the available cells. This is also described in more detail in “5.7.2.2 Table Info Dialog”, p140.

The scope of a factor table is related to table’s position in the experiment hierarchy; this is described in “Factor Set Scope”, p136.

Cell Weights

For most crossing types, cells are selected from the table so that all the cells are used at least once before any are used a second time. However, if a weight is assigned to a cell, that cell can be used as many times as the weight specifies before becoming ineligible for re-selection.

A weight can be assigned to all cells in a table through the Table Info dialog (see “5.7.2.2 Table Info Dialog”, p140).

If a level is assigned a weight (by double-clicking on the cell in the Factor Table window or by setting a level weight in the List dialog), all of the cells for that level are given the weight. If levels from different factors are weighted, the corresponding cells’ weights are increased multiplicatively.

The Factor Table window shows the weight of each cell in the table (see “5.7.2.1 The Factor Table Window”, p138).

Note: It is the responsibility of the experiment designer to set up the trial counts so that all cells are used the correct number of times.

Latin Squares

In the strictest sense, *Latin squares* refers to a between-subjects two-factor design where each subject sees several conditions. The condition set for a subject is chosen so that he sees every level of each factor, but never sees a level in more than one combination with other levels; however, over a few subjects, every level combination is seen by some subject. This idea is extended to three-factor designs with the *Graeco-Latin square*. The factors must all have the same number of levels.

Usually, a set of conditions is chosen for a particular subject by using a “diagonal” from the full crossing. The figure below shows which cells would be used for subjects 1, 2 and 3.

	Xa	Xb	Xc
Ya	1	3	2
Yb	2	1	3
Yc	3	2	1

Figure 132 – A 3 x 3 Latin square showing the condition selections for three subject groups

PsyScope uses the term “Latin square” for a more generalized cell-selection construct. Latin squares are defined using any number of factors, and by using *Latin square partitions*.

Each partition is made up of a number of factors, which are fully crossed with each other. The cells generated by these crossings are then treated like the levels of factors for Latin square purposes. There can be any number of partitions (although most designs use just two partitions), and a condition is selected from a the (hyper-) “diagonal” of the full crossing of the partition cells.

The factors that are Latin square crossed do not have to be of the same length; the position of a level within a factor is determined modulo the number of levels in the smallest factor.

As an example, a table is shown below where A x B is Latin square crossed with C x D (i.e., there are four factors in the table; A and B are in the first partition and C and D are in the second partition); the numbers in the table correspond to the first, second, third, and fourth subjects.

		Aa			Ab		
		Ba	Bb	Bc	Ba	Bb	Bc
Ca	Da	1	4	3	2	1	4
	Db	2	1	4	3	2	1
Cb	Da	3	2	1	4	3	2
	Db	4	3	2	1	4	3

Figure 133 – Latin square with two partitions

Counterbalanced Stimuli

One common experiment design uses Latin squares in a less obvious way. In this design, there are a number of conditions and a list of stimuli, where each stimulus could be in any condition; every subject should see all conditions and every stimulus, but he should see each stimulus only once. Further, all stimuli should be seen by some subject in every con-

dition, using as few subjects as possible (clearly equal to the number of conditions). This is called *counterbalancing* the stimuli.

This design problem is solved by Latin squaring the list of stimuli with all of the conditions; i.e., all of the factors determining the condition are put together in a partition, and the stimulus list is placed in its own partition.

5.7.1.8 Factors in the Hierarchy

Linking Lists to the Hierarchy

A list can be linked to the experiment object, or to any group, block, template, or factor table in the experiment hierarchy. There are two things that depends on where a list is connected to the hierarchy:

- Which objects can access the list using **Vary by List**.
- How cell-selections within a single factor set interact with the hierarchical path used to generate trials.

The latter of these is addressed below, in “Factor Set Scope”, p136.

Usually, a list is connected to a template or factor table set and it used (via **Vary by List**) by events which are linked only to that template or table set. In such a case, the list is always available to the events through **Vary by List**.

When a trial is built that contains a particular event, it is possible that only part of the hierarchy will be used to build the trial. (For example, if there are two templates which own the event, only one of the templates will be used to generate each trial.) In this case, lists must be properly connected to the hierarchy to allow certain events to use certain lists:

An event attribute can only vary by lists which will *always* be connected to hierarchy that is used in building a trial for that event; i.e., regardless of which group, block, or template is used to build the trial, the list must be connected to an experiment, group, template, or block which is used. In other words, the list must be connected to the hierarchy through *every path* to the event from the experiment object.

For example, consider this hierarchy:

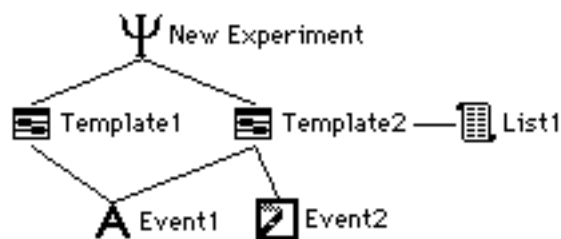


Figure 134 – An example hierarchy where “Event1” cannot vary by “List1”

When “Template1” is used to build a trial, “List1” is not connected to any of the objects used to build the trial. Since this possibility exists, the graphic environment will not allow you to use **Vary by List** with “List1” for an attribute of “Event1”. However, “Event2” can use “List1” because “Event2” is only linked to “Template2”.

Note that lists which are connected to the experiment can always be used by attributes in all objects. However, lists often cannot be linked to the experiment due to scope effects (see “Factor Set Scope”, p136, below).

Factor Set Scope

The pattern of cell selection in a factor table or factor set can depend on the non-factor elements of the experiment hierarchy. When an experiment hierarchy contains multiple templates — or multiple paths to a template — a new cell is *not necessarily* chosen for every factor set at the start of every trial. Also, multiple cell-use histories are kept for a factor set when multiple paths lead to the set’s owner object.

A new cell is chosen only for factor sets which are *relevant* to the trial being generated. A factor set is *relevant* if it is connected to the experiment object, or to a group, block, or template object that is used to generate the trial. (A table is relevant if the built-in template of the factor table set is used to generate the trial.)

For example, consider Figure 135 and Figure 136, below.

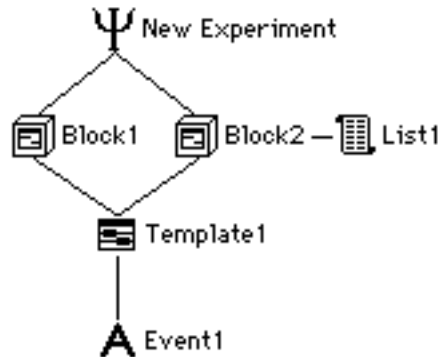


Figure 135 – Example hierarchy with a list connected to one block

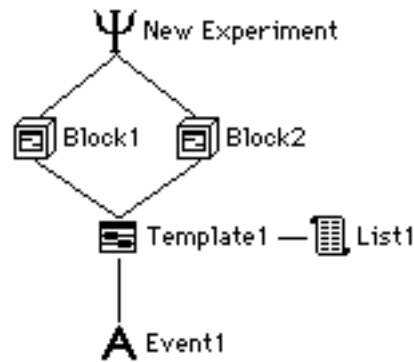


Figure 136 – Example hierarchy with a list connected to the template

In the first figure, “List1” is connected to “Block2”; a new item will be selected from the list only when “Block2” is used to build a trial. In the second case, a new item is always chosen for “List1”, because “Template1” is used for every trial.

Note: In Figure 135, the event “Event1” cannot vary by “List1” directly, due to the rules explained in “ Linking Lists to the Hierarchy”, p135. If this were a real experiment, some attribute in “Block2” must be using the list, since it is the only object which is allowed to use the list.

The hierarchical position of a factor set affects not only *when* a new cell is chosen for a factor set, but also how the chosen cell relates to cells used in previous trials. Specifically, when there are multiple paths from the experiment object to a factor set’s owner, then a separate cell-use history is kept for each path.

For example, in Figure 136, there are two paths that reach “Template1”: through “Block1” or through “Block2”. This means that the item selection for “List1” within “Block1” will be independent from the item selection of “List1” within “Block2”. This can have several implications; for instance, if “List1” uses the **Sequential** access type, then “Block2” will always start with the first item of “List1”, regardless of how many items of “List1” were used in “Block1”.

The scoping of a table can be specially modified through PsyScript; see “Part 4: Scripting Reference, 13.3.9.4 Scripting the Factor Set Scope”, p400.

5.7.2 Factor Table Windows

5.7.2.1 The Factor Table Window

To open the Factor Table window, double-click on a factor table object in the design window. The Factor Table window has a general control area at the top of the dialog, and a factor table area below.

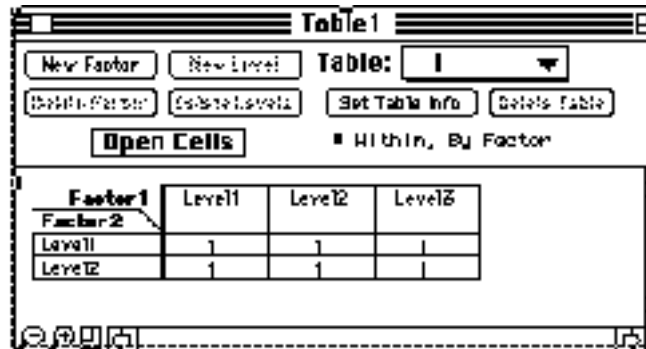


Figure 137 – The Factor Table window

At the top of the Factor Table dialog is the name of the table. To create a new table, click on the **Table** pop-up menu and select **New...** You can create multiple factor tables within a single factor table set object.

To delete the current table, click on the **Delete Table** button. There must always be at least one table in a factor table set.

Below the **Table** pop-up menu, bulleted text shows the crossing type and access type information about the current table. To change these, click on the bulleted text or click on the **Set Table Info** button. Doing either opens the Table Info dialog (see “5.7.2.2 Table Info Dialog”, p140).

The four buttons in the upper right of the dialog allow you to create new factors and new levels for the current table. Unless a level is selected when you click on **New Factor**, new factors will be added to the top level of the current table.

If a level is selected when you click on **New Factor**, the new factor is added as a nested factor to the selected level. (The name of a nested factor will not be visible; only its levels will be selectable.)

Double-clicking on a factor lets you modify the factor name and level order. The level order is used only if the table access type is **By Factor** (see “5.7.2.3 New/Rename Table Factor Dialog”, p142).

New levels are added to the factor that is currently selected by hitting the **New Level** button. (Nested factors are selected by selecting any level in the nested factor.)

Double-clicking on a level lets you modify the level name and weight (see “5.7.2.4 New/Rename Table Level Dialog”, p142).

To rearrange a factor in the table, drag the factor name to a new position. There is a dividing line between factors in this area: factors listed above the dividing line will be displayed with levels as columns, while factors below the line will be displayed as levels with rows. The row/column distinction is for viewing purpose only; it does not affect the accessing of cells in the experiment.

The top-to-bottom ordering of the factors does matter, however, in determining the cell order for **Sequential** and **Blocked** access types. (The exception is tables with Latin square crossing type; in that case, the ordering is controlled in the Latin Squares dialog, see “5.7.2.5 Latin Squares Dialog”, p142.) Level order within factors can be changed by dragging level names in the column or row headers.

The icons at the bottom left of the Factor Table dialog control the scale used to display the factor table. The table can be shrunk, enlarged, or zoomed to the standard scale by clicking on the icons.

The cells selected in a Factor Table window set the context for all Template and Attribute dialogs of objects owned by this factor table set. The Factor Table dialog and Factor Table floating window (i.e. Cell Chooser, see “5.7.2.7 Factor Table Floating Window”, p144) work together to maintain a consistent context.

Cells in the table can be selected directly using the standard Macintosh Shift- and Command-clicking modes. Cells can also be selected by clicking on a level; in this case, all of the cells for the level are selected (including those in a different row or column that nevertheless belong to the selected level). There are two special selection modes for clicking on levels:

- Option-click selects only the cells that are directly in the particular row or column selected.
- Control- or Shift-Command-click is like a selection-extension in reverse: only cells which are already selected *and* would have been selected by the click stay selected. This is useful for selecting cells which are the crossing of particular levels.

5.7.2.2 Table Info Dialog

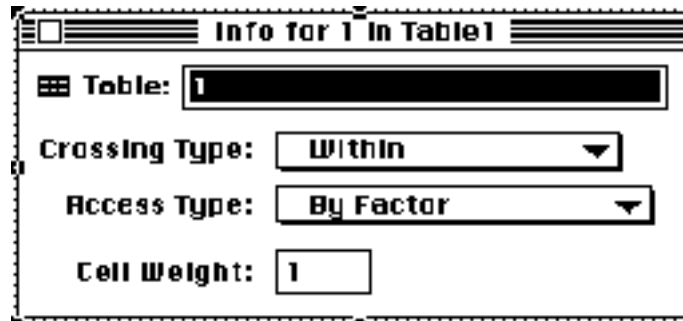


Figure 138 – The Table Info dialog

To open the Table Info dialog, click on the **Set Table Info** button in the Factor Table dialog (see “5.7.2.1 The Factor Table Window”, p138). The Table Info dialog is used to change the table name, its crossing type, or its access type. You can also set a weight which will apply to all of the cells in the table.

The crossing type can be one of the following:

- **Within** crossing type represents a “within subject” design. Cell-selection depends simply on the access type of the table (see below).
- **Between** crossing type represents a “between subject” design; a *single* cell will be used for all trials. The cell that is used depends on the set’s *index*, which usually depends on the subject number. (See “5.7.2.6 Choose Crossing Dialog”, p143 on setting the index.) Access type and cell weight specifications are ignored.
- **Latin Squares** crossing type is used for a Latin squares design (see “Latin Squares”, p133). Factors in the table are partitioned into fully crossed *subtables*; these subtables are then crossed based on a Latin square pattern that depends on the set’s *index*, which is usually linked to the subject number. (See “5.7.2.6 Choose Crossing Dialog”, p143 on setting the index; see “5.7.2.5 Latin Squares Dialog”, p142 about setting the subtable partitions; for a more detailed discussion of Latin squares.) The access type and cell weight specifications are still used to select cells within the Latin square subset.
- **List Access** crossing type represents another type of “within subject” design where a new cell is to be selected for a trial only if an explicit reference (via **Vary By List**) is made to one of the lists in the factor set. When a new cell is selected, the access type and cell weight specifications are used in the standard way. The **List Access** crossing type is not available for tables in a factor table set.
- **Use Access** crossing type is similar to **List Access**, except that a cell is selected after *every* reference (via **Vary By List**) to a list in the factor set; this means that the current cell can change during the compilation a trial. When a new cell is selected, the access type and cell weight specifications are used in the standard way.

- **Use/Reset Access** crossing type is like **Use Access** with the further specification that the table’s access history is erased after every trial. This makes the item-selection independent for every trial.
- **Fixed** crossing type is functionally equivalent to **Between**, but a different name is used to indicate that the set is somehow unrelated to the subject (e.g., the cell for this table will be “manually” incremented during the experiment). This crossing type is for advanced users.

The basic access types are the standard types described in “5.12.1 List Ordering”, p210. The extra access types available for tables are:

- **Cycle Random** access type is like **Random**, but cells are chosen so that all levels of each factor are used before a factor is used again (as if the factor was accessed independently with access type **Random**).
- **Blocked Random** access type is sensitive to the order in which factors are listed within the table. A level is chosen randomly for the first factor, and then that level will be used until all of the cells for that level are exhausted. This process applies recursively within the “blocks”: with the first level fixed, a level is chosen for the second factor and held until all of the cells for the *pair* of levels are exhausted, and so on.
- **Least-Used Random** access type is similar to **Cycle Random**; however, instead of trying to select an independently random level for each factor, a level is selected among those that have been used the least so far. Among the eligible levels within a factor, one is chosen randomly. The order in which factors are listed within the table is important: a level is first selected for the first factor, and then the least-used algorithm is applied for the second factor, counting level-uses from previous trials only when the level selected for the first factor was current. Level-use counting for the third factor takes into account the levels selected for the first two factors, and so on.
- **By Factor** access type moves the burden of access type specification down to the factors in the table, where level order is specified. A new cell is selected by choosing new levels in all the factors — based on the factors’ level orders — such that the cells of the table are properly exhausted. The level order of a factor is independent of other factors, and can be any of the standard access types (see “5.12.1 List Ordering”, p210), plus:
 - **Blocked Sequential** level order specifies that **Sequential** accessing should be used, but once a level is chosen, it should be used as long as possible (i.e. until the need to use an unexhausted cell forces a change in this factor).
 - **Blocked Random** level order specifies that **Random** accessing should be used, but once a level is chosen, it should be used as long as possible.
 - **Cycle Random** level order specifies that **Random** accessing should be used, but the factor’s list of level should be independently exhausted, if possible, before reaccessing a level within the factor. (Independent cycling may not be possible if another factor has **Sequential** level order.)

- **Least-Used Random** level order specifies that **Random** accessing should be used, but with precedence is given to levels which have been least-used so far in the experiment. If this factor is not the first factor in the table, levels from preceding factors will have already been selected; when level-uses from previous trials are counted, only trials which also had these levels as current are counted.

*Advanced Note: The standard table access types are all implemented by assigning appropriate level order types to the factors. For instance, **Random** table access is achieved by assigning all of the level orders to be **Random**, etc.*

5.7.2.3 New/Rename Table Factor Dialog

This dialog is like the New Object Name dialog (see “5.2.5 New Object Name Dialog”, p114), except that the pop-up menu below the text field is used to specify the access type of the new factor.

5.7.2.4 New/Rename Table Level Dialog

This dialog is like the New Object Name dialog (see “5.2.5 New Object Name Dialog”, p114), except that it contains an extra text field for specifying the level weight within the factor. Also when a new level is being created, there is an **Another** button that lets you add multiple levels at one time.

5.7.2.5 Latin Squares Dialog

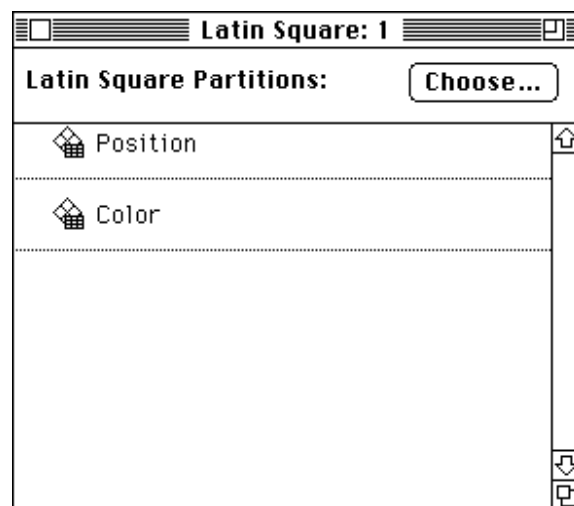


Figure 139 – The Latin Squares dialog

To open the Latin Squares dialog, select **Latin Squares...** as the factor set crossing type in the Table Info or Factor Set dialog (see “5.7.2.2 Table Info Dialog”, p140 and “5.7.3.3

Factor Set Dialog”, p147). (The use of Latin squares crossing type is described in “Latin Squares”, p133.) The purpose of this dialog is to set the Latin square partitions.

The **Choose...** button opens the Choose Crossing dialog (see “5.7.2.6 Choose Crossing Dialog”, p143), which is used to set the index by which a Latin square “diagonal” is selected.

The main part of the Latin Squares dialog is a list of all the factors in the Latin Square set with lines between the partitions. Factors that are in the same partition will be fully crossed; all of the fully crossed partitions will then be crossed Latin square with each other.

To add lines between items in the list, drag the line from the end of the list. You can remove a line by dragging it back to the end of the list. You can also drag the factors to achieve the desired partitioning.

The order that factors are listed in the Latin Squares dialog is the “real” order of the factors within this set; this order can be important for certain access types. Factor order is usually controlled through the Factor Table or Factor Set dialogs, but the Latin Squares dialog overrides those orders.

5.7.2.6 Choose Crossing Dialog

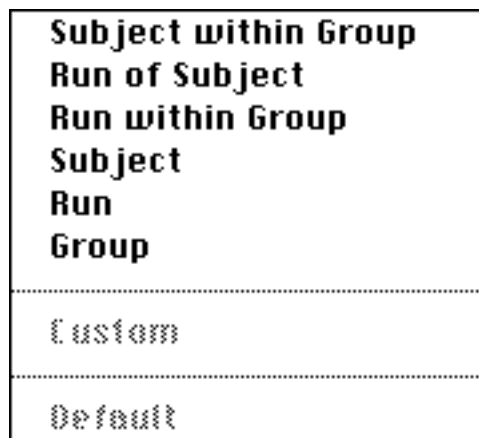


Figure 140 – Choose Crossing menu

The Choose Crossing dialog is used to set the factor set index for **Between** and **Latin Square** crossing types.

The Choose Crossing dialog is opened when you select **Between...** from the crossing types pop-up menu in the Factor Set or Table Info dialogs (see “5.7.2.2 Table Info Dialog”, p140 and “5.7.3.3 Factor Set Dialog”, p147), or when you hit the **Choose...** button in the Latin Squares dialog (see “5.7.2.5 Latin Squares Dialog”, p142). It is used to configure the factor set’s index.

The only item in the dialog is a pop-up menu with the following items:

Subject within Group – links the index to the current subject number.

Run of Subject – links the index to the current run number.

Run within Group – links the index to the current group run number.

Subject – links the index to the current subject count.

Run – links the index to the current total run count.

Group – links the index to the current group number.

Default – If there is a “SubjectNumber” Subject Info item, the value of this item is used. Otherwise, the constant 1 is used.

All of the above numbers are calculated by the **Assign Group and Calculate Numbers** process, which requires that the subject name be logged with each run of the experiment; see “6.2.3 Subject Number Calculation”, p230 for more information.

5.7.2.7 Factor Table Floating Window

The Factor Table floating window lets you quickly change the cells that are selected in a context-setting factor table. This floating window will only be available when a Template or Attribute window is frontmost, and when there is some factor table that sets context for the window. (See also “5.7.1.1 Factor Table Sets”, p130.)

To open the Factor Table floating window, select **Cell Chooser** from the **Windows** menu. If there is more than one table that can set context for the current window, you can cycle through the tables using the **Next** and **Preu** buttons. Only the table that is currently shown will affect the context.

If a table is drawn in both this floating window and a regular Factor Table window in the background, the cell selections are kept consistent. Thus, when you change the cell selection through the floating window, the cell selection will also change in the Factor Table window.

5.7.3 List Dialogs

5.7.3.1 List Dialog

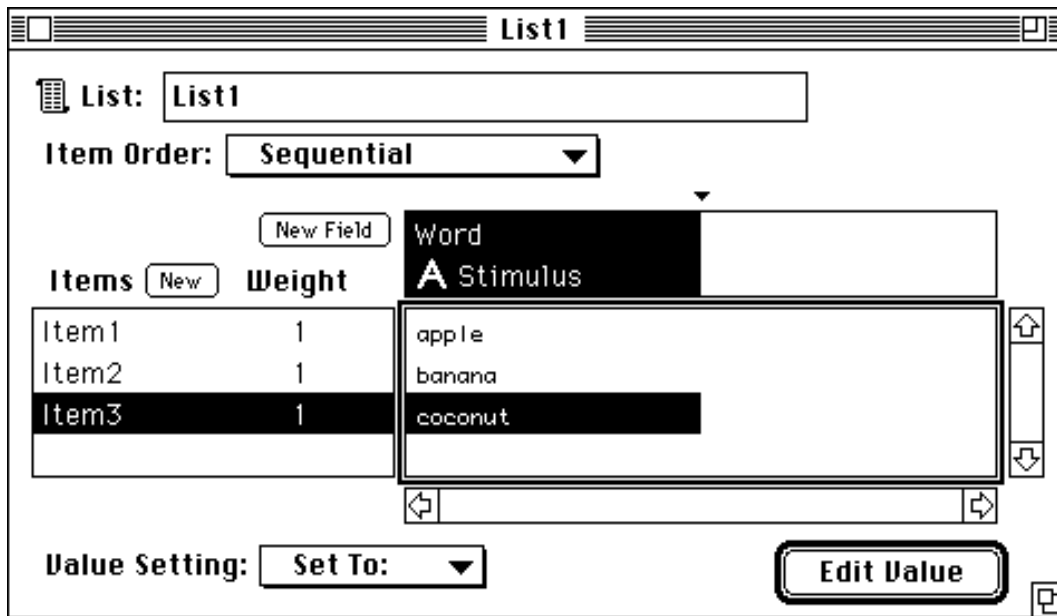


Figure 141 – The List dialog

The List dialog is used to set up the items, fields, and values for a list. (Lists are described in “5.7 Factors and Lists”, p129.) To open the List dialog, double-click on the list’s icon in the Design window.

At the top of the dialog is a text field that contains the list name. To change the list name, edit the value in the text box; the name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a certain delay (see “7.6.4 Design Options”, p268).

The main body of the List dialog is a table (*not* a factor table) in which each row represents an item of the list and each column represents a field. Each cell in the table contains the value of one field for one item. Above the table is the list of fields, and to the left is the list of items and item weights.

When a cell in the table is selected, the **Value Setting** pop-up appears in the lower-left corner. This pop-up works the same as the pop-ups in the list of an Attribute dialog (see “5.8.2.1 Settings”, p152). The **Edit Value** button opens a dialog to change the setting value of the currently selected cell.

To create and add new items to the list, click on the **New** button above the list of items, or select **New Item...** from the **Edit** menu when the item list is active. To change an item name, double-click on the item.

To create and add new fields to the list, click on the **New Field** button to the left of the list of fields, or select **New Field...** from the **Edit** menu when the field list is active. This will open the New/Rename/Retype Attribute dialog (see “5.8.2.4 New/Rename/Retype Attribute Dialog”, p157). To change a field name or type, double-click on the field.

To change the weight for an item, click on the weight that appears next to the item name. A text box will appear to let you change the weight. Hit Return, or click on anywhere else in the dialog, after making the change.

If you are working with an expanded list (see “5.7.1.4 Expanded Lists”, p132), you can include existing free levels in the list using **Get Level...** from the **Edit** menu (see “5.2.8 Get Object Dialog”, p115). You can also open the Level dialog for a level by double-clicking on the level.

The **Item Order** pop-up sets the item order for the list used for factor set cell selection; see “5.12.1 List Ordering”, p210 for information about the choices in this menu; see “5.7.1.7 Level Order and Crossing Types”, p132 for information on the item order.

When a level is selected in an expanded list, the **Nested Factors** button appears; clicking on this button opens the Factor Set dialog for the selected level. For more information on nested factors, see “5.7.1.5 Nested Factors”, p132.

The tick marks above the field list can be dragged to resize the all of column widths; columns cannot be sized independently.

5.7.3.2 List File Dialog

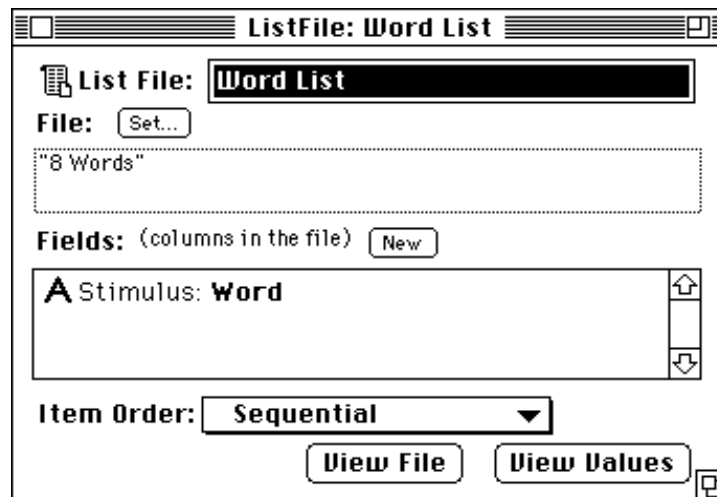


Figure 142 – The List File dialog

To open the List File dialog, double-click on a list file icon in the Design window. This dialog lets you set the file to read for the values and the types of data columns stored in the file. Creating and setting column types works just like fields in the List dialog (see “5.7.3.1 List Dialog”, p145). The item order for the list is controlled with the pop-up menu at the bottom of the dialog.

A column that has a type expecting multiple or non-literal tokens is marked with an asterisk (*) after the name. This indicates that a special format must be used for this column in the file (see “ List Files with Non-literals”, p131).

Hitting **View Values** opens the List dialog for the file list and all of the field values will be shown, but they cannot be changed and no new items can be added. (The List dialog is described in “5.7.3.1 List Dialog”, p145.)

5.7.3.3 Factor Set Dialog

To open the Factor Set dialog, click on the **Lists** button in an Experiment dialog, Group dialog, Block dialog, or Template window. You can also control-click on an icon in the Design window and choose **Lists** from the resulting pop-up menu.

The Factor Set dialog shows how the lists owned by an object are grouped into factor sets; lines in the list separate the sets. (See “5.7.1.1 Factor Table Sets”, p130 for information on factor sets.)

Each set has a pair of pop-up menus associated with it, one for assigning the set’s access type and another for the set’s crossing type.

To move a list from one set to another, drag it. To create a new factor set, drag a list to the bottom of the dialog, and the list will be put into a new set.

Double-clicking on the list opens the dialog for that list (see “5.7.3.1 List Dialog”, p145 and “5.7.3.2 List File Dialog”, p146).

Selecting the **Between...** crossing type opens the Choose Crossing dialog for setting the set’s index (see “5.7.2.6 Choose Crossing Dialog”, p143, and “5.7.2.2 Table Info Dialog”, p140 for information on the factor set index). Selecting the **Latin Squares...** crossing type opens the Latin Squares dialog for setting Latin square partitions (see “5.7.2.5 Latin Squares Dialog”, p142).

If a list has been imported into a factor table, it will be in a partition without pop-up menus, and you cannot drag to or from the partition. This is because the Factor Set dialog is not allowed to modify factor tables.

5.7.3.4 Level Dialog

To open the Level dialog, either double-click on a level icon in the design window, or double-click on a level of a free factor in the List dialog (see “5.7.3.1 List Dialog”, p145). The Level dialog allows the values of the factor fields to be set for one particular level.

Note: This dialog is used only by free levels, which are owned by extended lists. If your experiment does not have any extended lists, this dialog will never be used.

The Level dialog is a standard attribute dialog, described in “5.8.2 The Standard Attributes Dialog”, p151. Its only set of attributes is a list of fields defined for the factor that owns the level. Setting the field values in this dialog is equivalent to setting them in the Factor dialog for the owning factor.

5.7.3.5 Connect List Dialog

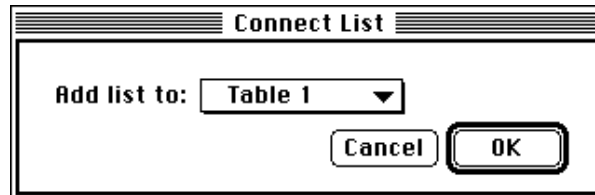


Figure 143 – The Connect List dialog

When a list is linked to a factor table object in the Design window, the *Connect List* dialog appears. This dialog is used to pick the destination table or factor set for the list.

The Connect List dialog contains a single pop-up menu through which a table or factor set is selected. All tables and factor sets which exist in the factor table object are shown in the pop-up menu as **Table X** (for tables) or **Set X** (for factor sets), where **X** is the name of the table or set. There are also **New Table** and **New Set** items, which create a new table or set to contain the list.

Hitting **OK** links the list to the factor table and places it into the selected table or set. Hitting **Cancel** cancels the link.

The Connect List dialog appears *only* when a list is connected to a factor table object. When a list is connected to any other type of object, it is automatically placed into its own set.

5.8 Attributes

5.8.1 Definitions

Every object has *attributes* associated with it. Attributes specify features about the object that are used to build trials when the experiment is run.

The graphic environment groups attributes into *attribute sets*:

Each object type has a set of *standard attributes*, which are attributes used by every object of that type.

Certain types of objects can also have a set of *custom attributes*. These are attributes that are defined by the experiment designer; they are used to by other objects that are owned to the object with the custom attributes.

Except for events, each object has a set of *default attributes*. These are the same as the standard attributes for object types that it can own.

The attributes most commonly used are the standard experiment and event attributes. Experiment attributes set global parameters, such as the instructions file or background color for the screen. Experiment attributes can only be set through the experiment object.

The standard event attributes are actually divided into two sets: *event attributes* and *stimulus attributes*. Event attributes are standard for all event types, such as the event's duration. Stimulus attributes are standard for a particular event type; e.g., the font or color of a text stimulus, or the volume of a sound stimulus.

All attribute values are set using the Attributes dialog. Attributes dialogs are basically the same; each type of object has its own Attributes dialog, which uses attribute sets are appropriate for the type.

Every attribute (except the **Stimulus** event attribute) has a built-in default value which is used if its value is not explicitly set. The built-in defaults for an object can be overridden by its owners through the default attribute sets. For example, in a template that owns events, you can set the font to be used for all of the template's text events; this is done by setting the font attribute in the template's default stimulus attribute set.

In general, specifying an attribute value at one level of the hierarchy overrides all values specified for that attribute at any higher level in the hierarchy. For example, default attributes values assigned in a block override default attribute values assigned in the experiment, and those of the template will override the block's, etc. (See "5.8.1.1 Attribute Inheritance", p150 for more information about default attributes.)

Tip: Assigning values to the default attributes at the level of the experiment object provides a useful means for changing the default values of attributes ordinarily used by Psy-Scope.

5.8.1.1 Attribute Inheritance

Usually, the value of an object's attribute is specified within the object. If no value is specified, the default attribute sets of the object's owners are searched to obtain a value. When the default value of an attribute is found in this way, it is called **attribute inheritance**. If no value for the attribute is found in the owning objects, all attributes — except the **Stimulus** event attribute — have built-in defaults that are used.

The search for an inherited attribute proceeds from the lowest level in the hierarchy to the highest, so that specifications “closest” to the object are the ones that apply.

When a particular attribute in the Attributes Dialog has the **Default** setting, attribute inheritance will be used to find a value (or the attribute will take the built-in default if no value can be inherited). See “5.8.1.3 Attribute Dialogs”, p150.

To set attribute values that you want to be inherited, you use the **Default [Type] Attribs** (e.g. **Default Event Attribs**) attribute sets in the Attributes dialog (see “5.8.3 Experiment Attributes”, p158, “5.8.4.3 Group Attributes Dialog”, p170, “5.8.5.3 Block Attributes Dialog”, p172, and “5.8.6.3 Trial Attributes Dialog”, p174).

5.8.1.2 Factor Table and Attributes

The process of searching for an object's attribute value changes when the object is linked to a factor table. Instead of simply finding an attribute value in the object or finding an inherited value, there may be a different value for the attribute that depends on which cell is current in the factor table.

When an object is linked to a factor table, its attribute values are always set in context; i.e. when you set an attribute value, the value will only apply to the cells which are currently selected in a context-setting factor table. In this way, the linking of attribute values to the various levels of a factor occurs automatically.

If no Factor Table window is open — or no cells are selected in a context-setting table — then attribute value changes will apply to all cells of the table.

See “5.7.2.1 The Factor Table Window”, p138 for more information on the Factor Table window and setting attributes.

5.8.1.3 Attribute Dialogs

The attribute dialog for an event or sub-stimulus object is opened by double-clicking on the object in the Design window, Template window, or Stimuli dialog. For all other objects, the attribute dialog is opened through an **Attributes** button in the object dialog, or by con-

triple-clicking on the object in the view window and selecting **[Object] Attributes** from the resulting pop-up menu.

The attribute dialogs for the standard object types are basically all the same; they just have different sets of attributes available. The object dialogs are all considered examples of the *Standard Attributes dialog*.

5.8.2 The Standard Attributes Dialog

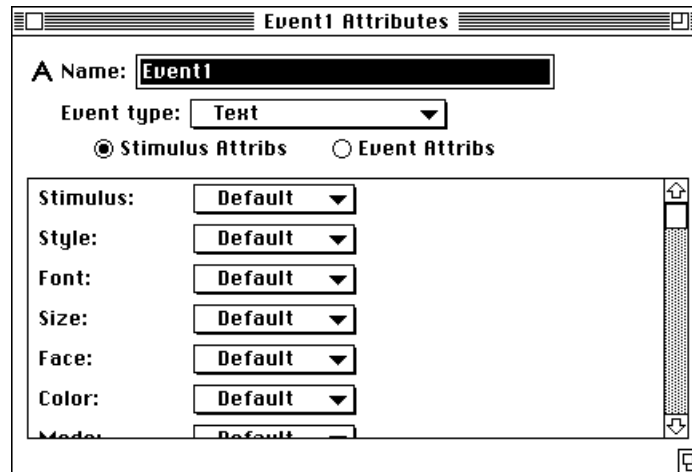


Figure 144 – A Standard Attributes dialog

All attribute dialogs have the same basic form, and are therefore called Standard Attributes dialogs.

At the top of the dialog is a text box that contains the name of the object. You can usually change the name of the object in this text box; the name change will take effect when you close the dialog, move another window to the front, or when no typing occurs after a certain delay (see “7.6.4 Design Options”, p268).

The main body of the dialog consists of a list of attributes. Each attribute has a *setting*, which is displayed in a pop-up menu, and a value with the setting. Except for some of the experiment attributes, all of the settings start out as **Default**. (Some experiment attributes are initialized in the standard script file created by **New Experiment...**)

Not all of the object’s attributes are shown at once. A pop-up menu (or a pair of radio buttons in the case of the Event Attributes dialog) switches the list between different attribute sets.

In the list of attributes, the setting pop-up menu provides a choice of ways to set an attribute's value. The possible settings are **Set To**, **Default**, **Vary By...**, and **Script...**

The settings **Multiple** and **Other** are display-only: they are possible interpretations of the current attribute value, but you cannot directly set an attribute to be one of these.

To the right of the setting pop-up, the current setting value of the attribute is shown with a bullet. (If the setting value is **Default**, no setting value will appear.) To change the setting value, click on the bulleted value.

The setting value that is shown might depend on whether the Trial Chooser is open or closed. When the Trial Chooser is open, the setting value shows the actual attribute value, given the hierarchy and item selections from the Trial Chooser. When the Trial Chooser is closed, the setting value is a parameter for the setting, such as the list and field to link to for **Vary by List**.

5.8.2.1 Settings

These are the possible values for the settings pop-up menu next to each attribute, and the meanings of the settings:

Set To

This is the basic setting: it allows you to assign a fixed value to the attribute. Choosing this setting will open a dialog that is specific to the attribute (e.g., a list of fonts for the font attribute) through which you assign the attribute value (e.g., Helvetica).

Default

This setting causes the attribute value to be determined either at a higher level in the experiment structure (see “5.8.1.1 Attribute Inheritance”, p150) or by a built-in default value.

Vary By...

The **Vary By** submenu allows you to vary the value of the attribute across trials based on another part of the experiment structure. The ways in which an attribute can be varied are described below. When the Trial Chooser is open, the value area shows the actual value assigned to the attribute for the given hierarchy and item selections.

Vary by List

Vary by List links the value of an attribute to the current value of a field in a list (see “5.7 Factors and Lists”, p129). A new current item for the list is chosen for each trial, so that the

value of the attribute will vary from trial to trial. When the Trial Chooser is closed, the value area shows the name of the factors and field to which the attribute is linked.

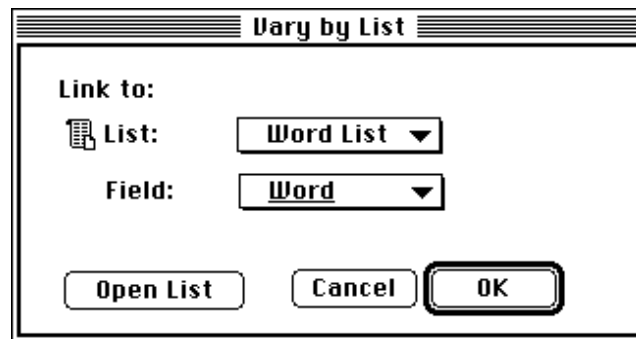


Figure 145 – The Vary by List dialog

To open the Vary by List dialog, select **Vary by List** in the settings pop-up menu. This dialog lets you set the list and field that the attribute will vary with. The top pop-up menu selects a list, and the bottom pop-up menu selects a field.

The list you select must already be connected to something in the attribute’s hierarchy. For event attributes, this means that the list must be connected to the event’s template, block, group, or experiment.

To create a new list, select **New...** from the top pop-up menu. The new list will be automatically linked to the hierarchy, and it will automatically have a field with the correct type.

Fields of the list that have the same type as the attribute being set are underlined in the field pop-up menu. In general, you will only link and attribute to a field that is underlined.

Clicking on the **Open List** button accepts the current pop-up items as the list and field to link to and then opens the List dialog for that list (see “5.7.3.1 List Dialog”, p145).

For more on lists, see “5.7 Factors and Lists”, p129.

Vary by Template

When an event is part of more than one template, the event can take its value from the template that executes it using **Vary by Template**. **Vary by Template** works with custom template attributes (see also “5.8.6.1 Custom Template Attributes”, p173).

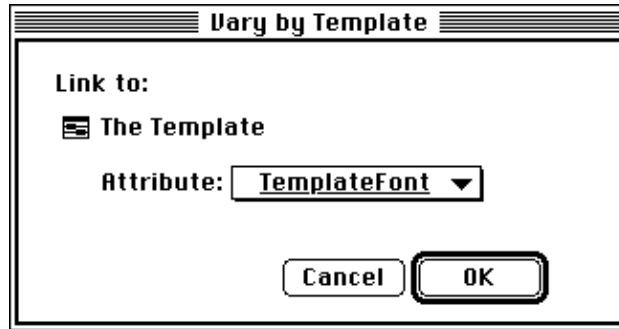


Figure 146 – The Vary by Template dialog

Choosing **Vary by Template** will open the Vary by Template dialog, which lets you select which custom template attribute to link to. A pop-up menu selects the attribute.

For an attribute to use **Vary by Template**, all of the owning templates must have a custom attribute of the same name. The pop-up menu will contain only custom attribute names that are common to all of the connected templates.

Vary by Block, Group

Vary by Block and **Vary by Group** work in the same way as **Vary by Template**, except that custom attributes at the block and group levels are used, respectively. See the previous section for information on **Vary by Template**. Custom block attributes are described in “5.8.5.1 Custom Block Attributes”, p171, and custom group attributes are described in “5.8.4.1 Custom Group Attributes”, p169.

Vary by Run Mode

Vary by Run Mode works in the same way as **Vary by Template**, except that custom run and practice attributes are used (see “5.8.3.1 Custom Run and Custom Practice Attributes”, p158).

Script...

The **Script...** class of settings allows you to link the value of the attribute to entities defined in PsyScript. (PsyScript is described in “, Chapter 12. PsyScript Reference”, p319.)

Script Linked

The **Script Linked** setting is used to link the attribute's value to the value of a script entry. Generally, this means that the two values (the attribute that is being set and the one that it is linked to) will be the same.

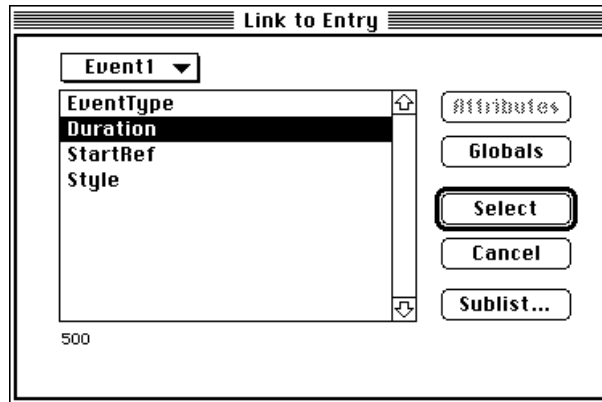


Figure 147 – The Link to Entry dialog

Choosing **Script Linked** will usually open the Link to Entry dialog; this dialog contains a list of all of the entries in the script and lets you choose which entry to link to. If the attribute is already linked to a sublist, the Sublist dialog will be opened instead (see below).

Selecting an entry shows its content (value) at the bottom of the dialog. Clicking the **Attributes** button (or double-clicking on the name of the entry) will “go into” the entry and change the list to show all of attributes for that entry, in much the same way that a folder is opened in the standard file dialog. The **Attributes** button is grayed if the entry or attribute does not have sub-attributes.

The **Sublist...** button switches the link mode and opens the Sublist dialog, which allows you to assign the attribute value to a fixed sublist of an entry value. Sublists are described in “Part 4: Scripting Reference, 12.8.7 Sublisting”, p334.

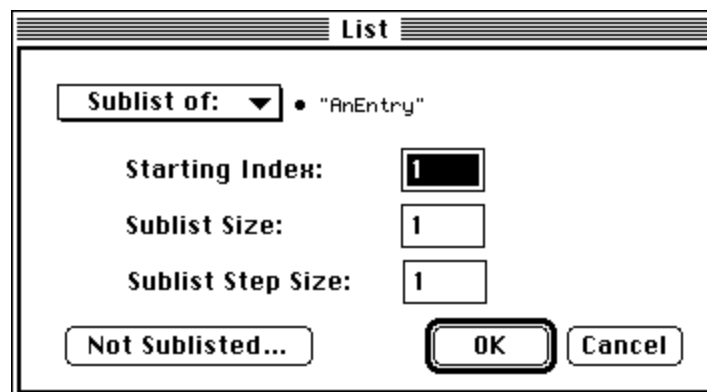


Figure 148 – The Sublist dialog

The Sublist dialog lets you link an attribute to only part of the values of another entry. Clicking the **Not Sublisted...** button switches the link mode back to standard and opens the Link to Entry dialog.

The type of sublist — **Sublist of:**, **Row of:**, or **Column of:** — is selected through the pop-up menu at the top of the dialog.

The entry that is sublisted is set by clicking on the bulleted entry name at the top of the dialog; this opens the Link to Entry dialog described above to select the entry. Unlike clicking on **Not Sublisted...**, no mode change takes place in relation to the attribute being linked, and closing the Link to Entry dialog will return to this dialog.

The text boxes that are below the entry name set parameters for the sublist, and vary depending on the type of sublist being made. See “Part 4: Scripting Reference, 12.8.7 Sublisting”, p334 for information on these parameters.

Script Access

The **Script Access** setting gets the value for the attribute by accessing a PsyScript-defined list (executing the `Access()` script function). PsyScript lists are described in “Part 4: Scripting Reference, 12.8 Lists”, p329.

Choosing **Script Access** opens the same dialog(s) as **Script Linked** (see above).

Script Current

The **Script Current** setting gets the value for the attribute as the current item of a PsyScript-defined list (executing the `GetCurrent()` script function). PsyScript lists are described in “Part 4: Scripting Reference, 12.8 Lists”, p329.

Choosing **Script Current** opens the same dialog(s) as **Script Linked** (see above).

Multiple

This attribute setting (display-only) indicates that the attribute has different values (of equal precedence) among the cells that are currently selected in the Factor Table window. This attribute setting will never appear when only one cell, a whole row, or a whole column is selected in the table.

Other

This attribute setting (display-only) indicates that the attribute value is determined by something that the graphic environment cannot read. This setting should only appear when using the graphic environment with scripts that were created through PsyScript directly.

5.8.2.2 Custom Attribute Sets

Experiment, group, block, and template custom attributes are defined in their respective attribute dialogs.

To define a new custom attribute, you must first select the custom attribute set (through the pop-up menu near the top of the dialog). When the custom attributes are the current set, a pair of buttons appear in the top left of the dialog: **New...**, and **Delete**. These buttons are used to add or remove custom attributes to the list. Custom attributes can also be cut and pasted in the list.

Clicking the **New...** button opens the New Attribute dialog, described in “5.8.2.4 New/Rename/Retype Attribute Dialog”, p157. The name or type of a custom attribute can be changed by double-clicking on the attribute name in the list.

5.8.2.3 Special Keyboard Shortcuts

When the text box at the top of the dialog is the current item (as opposed to the list of attributes), you can switch attribute sets using Command-Down arrow and Command-Up arrow.

When the attribute list is the active item and an attribute is selected, Command-Up arrow or Command-= changes the state of the attribute to **Set To**. For certain types of attributes — ones whose values are a single token — you can set the value through a text box in the Attribute dialog without opening a separate window; to do this, hit Command-Right arrow.

5.8.2.4 New/Rename/Retype Attribute Dialog

The New Attribute dialog is called when either: a) a custom attribute is created in an experiment, group, block, or trial template, or b) a new field is created in a free factor or list. The Rename/Retype Attribute dialog is called when an attribute created in one of the above ways is double-clicked (see “5.8.2.2 Custom Attribute Sets”, p156, “5.7.3.1 List Dialog”, p145).

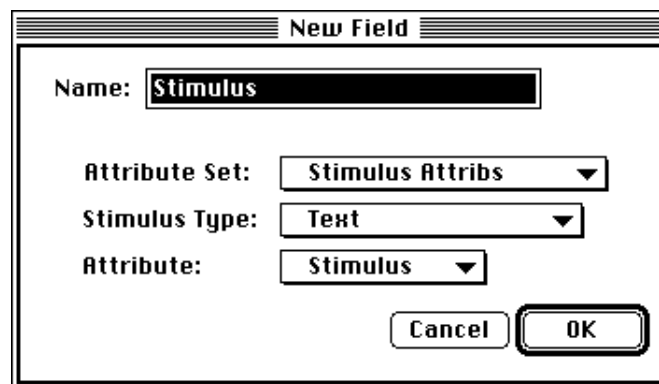


Figure 149 – The New/Rename/Retype Attribute dialog

This dialog has a text field for the attribute name and two or three pop-up menus that specify the type for the attribute. (Attributes and fields must have a type so that the proper dialog can be opened to set values.)

*Note: Factors or lists created by selecting **New...** in the Vary by List dialog will automatically have a field of the correct type; the user is prompted for the field name only.*

The pop-up menus do not represent independent parameters: the top one (or top two, for stimulus attribute or action parameter sets) navigates among attribute sets that contain the actual attribute types.

When creating custom attributes, PsyScope may warn you that the suggested name is a standard attribute name; this means that you have tried to create some attribute that is already available as a standard or default attribute. You should change the name so that is a non-standard attribute name.

When you change the type of a field (in a non-file, non-extended list), the attribute types you can change to will be constrained by the initial type of the field. The constraint is based on whether the original type could have multiple tokens as a value.

5.8.3 Experiment Attributes

Experiment attributes control features of the experiment that span all of the trials and events, as well as features that pertain to the overall running of the experiment.

5.8.3.1 Custom Run and Custom Practice Attributes

Custom run and custom practice attributes can be defined for use with the **Vary by Run Mode** setting in the Attribute dialogs (see “5.8.1.3 Attribute Dialogs”, p150). This allows an attribute to have a value that depends on whether the experiment is executed using **Run** or **Practice** commands. See “5.8.2.2 Custom Attribute Sets”, p156 for a description of how to define custom attributes.

The “RunMode” Attribute

The “RunMode” attribute is not directly adjustable from the graphic environment. PsyScope automatically sets the value of this attribute to `Run`, `Practice` or `Check` when the experiment is run. This can be used in PsyScript expressions to vary how aspects of the experiment are run in the different modes. It also determines the value of any attributes that have been varied by Run Mode (see “Vary by Run Mode”, p154).

The value of “RunMode” is set to `Practice` whenever you use the **Practice** command (from the **Run** menu or the Console) or the **Preview** button in a block or template dialog. When the value is `Practice`, the experiment is run the same as if the **Run** command were used, however no data is recorded in the data file, and the “RunNumber” Subject Info item value is not incremented (see “6.2.3 Subject Number Calculation”, p230).

The value of “RunMode” is set to `Check` whenever you run the experiment (in the Run or Practice mode) and the **Check** box is checked in the Trials Monitor (see “6.3 The Trial Monitor”, p238). When the value is `Check`, PsyScope does not actually run the experiment.

It goes through all of the steps involved in constructing each trial of the experiment (including loading stimuli if the **Load Stims** box is checked), and then reports statistics about the experiment in the Check Statistics window. For more information on the Trial Monitor, see “6.3 The Trial Monitor”, p238.

5.8.3.2 Default Stimulus/Event/Trial Attributes

Values for attributes used by trials and events can be specified at the experiment level. Unless these defaults are overridden at lower levels of the hierarchy, these value will be inherited by all objects belonging the experiment. See “5.8.1.1 Attribute Inheritance”, p150 for more information.

5.8.3.3 Experiment Attributes Dialog

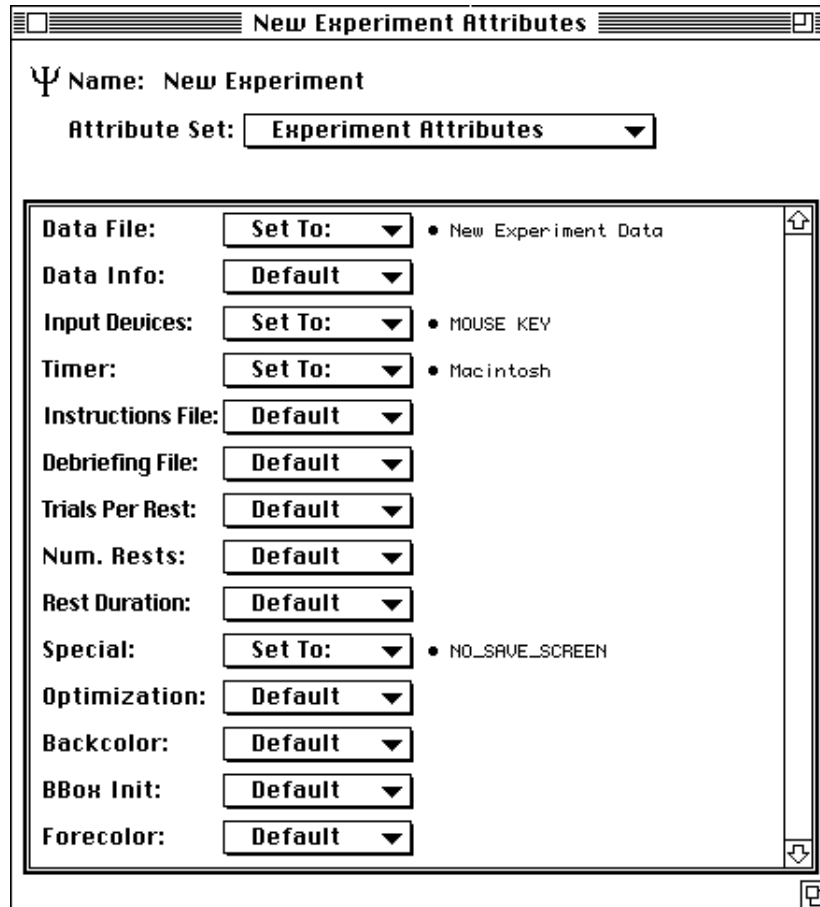


Figure 150 – The Experiment Attributes dialog



Figure 151 – Experiment attribute sets menu

The Experiment Attributes dialog is one of the standard attribute dialogs, described in “5.8.2 The Standard Attributes Dialog”, p151. The attribute sets in the Experiment Attributes dialog are:

Experiment Attributes (the standard experiment attributes, as described below)

Custom Run and Practice Attributes (discussed in “5.8.3.1 Custom Run and Custom Practice Attributes”, p158?)

Default Stimulus Attribs (described in “5.8.7.3 Stimulus Attributes”, p181, creation described in “5.8.2.2 Custom Attribute Sets”, p156)

Default Event Attributes (described in “5.8.7 Event Attributes”, p176)

Default Trial Attributes (described in “5.8.6.3 Trial Attributes Dialog”, p174).

5.8.3.4 Standard Experiment Attributes

Listed below are all of the standard experiment attributes. Other experiment attributes can appear in the list if you use a PsyScope Extension that uses experiment attributes. Consult the extension’s documentation for more information.

Data File

Data File specifies the file that stores data generated during an experiment run and other information about the experiment.

In brief, the data file is a text file that can contain information about the subject (e.g., name, age, number, etc.) and the experiment (e.g., title, time, machine type, etc.), as well as a line for each response that is recorded during the run of experiment. See “6.1.4 The Data File”, p217 for more information about the data file and its content.

The dialog that is opened to set **Data File** is the standard file dialog.

Data Info

Data Info specifies which data elements related to subject responses should be recorded, e.g. whether to record the name of the event that received the response. It also specifies whether or not to record information about the experiment and subject in the data file.

Experiment and subject information are recorded as a “header” at the beginning of the file. Following the header, a line is added each time a response is recorded using the RT[] action (see “RT[]”, p203); each response line is made up of a set of tab-delimited entries recording information about the response.

For a complete description of the data file and its contents, see “6.1.4 The Data File”, p217.

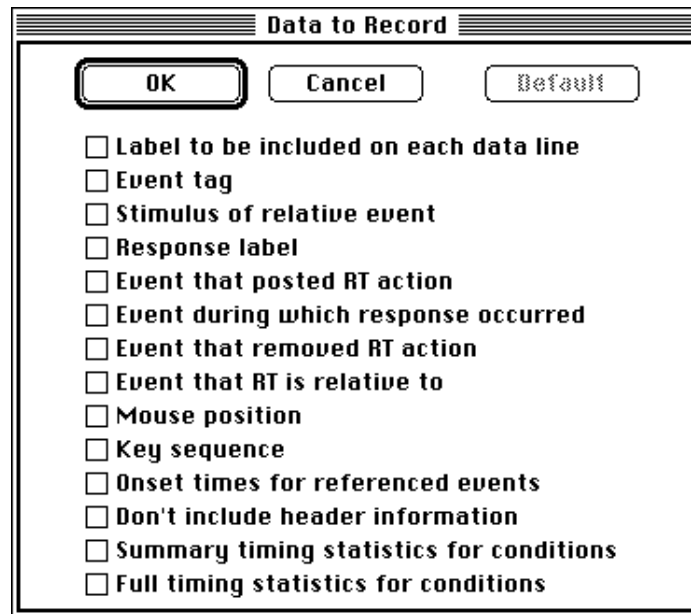


Figure 152 – Data Info dialog

The dialog that is opened to set **Data Info** is a shown above. The data recording flags are:

Label to be included on each data line: Enables recording of the “RunLabel” items at the beginning of every data line. This content of this label is completely up to the experiment designer, but currently must be set through PsyScript in the “RunLabel” experiment attribute; it can be used to identify the data with the current subject, run, script, etc.

Event tag: Enables recording of the **Tag** event attribute for the event that posted the RT[] action. This can be used to label events other than by their name and/or type. This string will appear on each line for which a response is recorded.

Stimulus: Enables recording of the stimulus (in an appropriate text form) of the event that posted the RT[] action. If the event was a **Text** event, the text is the stimulus text itself; otherwise, the text is either the name of the file, the name of a resource, or some other element within the file that identifies the stimulus.

Response Label: Enables recording of the **Label** parameter of the RT[] action (see “5.9.4.1 Available Actions”, p198 for a description of the RT[] action). This is used by the experiment designer to label responses for analysis purposes.

Event that posted RT action: Enables recording of the name of the event that posted the RT[] action.

Event during which response occurred: Enables recording of the name of the event that was most recently started and was still active when the response occurred. This

may be different than the event that posted the RT[] action if: 1) another event has started since the RT[] action was posted, or 2) the posting event ended, leaving the RT[] action active (using the **Active Until** parameter; see “5.9.1 Conditions and Actions Dialog”, p193).

Event that removed RT action: Enables recording of the name of the event that would have removed the RT[] action (had it not been triggered). This may be different than the event that posted the RT[] action, if the **Active Until** parameter was used (see “5.9.1 Conditions and Actions Dialog”, p193). If the RT[] action is scheduled to be active until the end of the trial, TRIAL_END will be reported.

Event that RT is relative to: Enables recording of the name of the event that was used for timing the response. By default, this is the event that posted the RT[] action, and the time recorded is the number of milliseconds elapsed since the start of the event. However, the **RelativeToEvent** parameter of the RT[] action can be used to time a response relative to the start of another event. This allows RT[] actions posted by different events to time their responses relative to the start of the same event.

Mouse position: Enables recording of the X and Y display coordinates of the mouse position at the time the response occurred. X is measured in pixels from the left of the screen, and Y from the top. (If multiple monitors are in use, X and Y are measured relative to the screen with the menu bar.)

Key sequence: Enables recording of the sequence of keys recorded by the last **Key Sequence** event. See also “Key Sequence Event Type”, p178.

Onset times for referenced events: Enables recording of the time, relative to the start of the trial, at which each recorded event began. The “recorded events” are events whose name has been recorded using one of the data items described above.

Don’t include header information: Disables recording of subject and experiment information at the beginning of the data file. This creates “pure” data files, which can facilitate importing data directly into other programs. The subject and experiment information can also be recorded in the log file (see “6.1.4 The Data File”, p217 and “6.1.5 The Log File”, p222).

Summary timing statistics for conditions: Enables recording of the average duration of each event in each different type of trial (i.e. condition) in the experiment. This information is recorded at the end of the data file.

Full timing statistics for conditions: Enables recording of the onset and duration of every event in the experiment, sorted by condition. This information is recorded at the end of the data file.

Data Variables

Data Variables specifies a list of variables that should be recorded with each line in the data file. The variables are defined using the Trial Manager Variables dialog (see “5.10 Trial Manager Variables”, p205, and “5.10.5 Trial Manager Variables Dialog”, p208).

Input Devices

Input Devices specifies which of the available input devices will be checked for input while the experiment is running. If a device is not included here, input from that device will be ignored during the experiment.

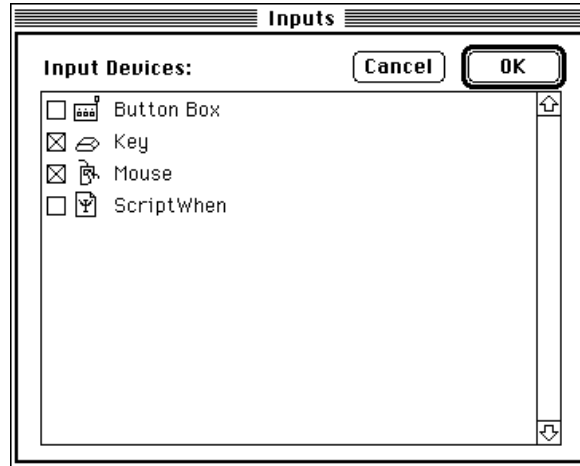


Figure 153 – The Input Devices dialog

Timer

Timer specifies the device that PsyScope will use for timing events. The default timing device is the Macintosh itself, providing 17-millisecond accuracy. If a CMU button box is available, then **Button Box** can be selected for millisecond accuracy. Extensions can be written to use other timing devices.

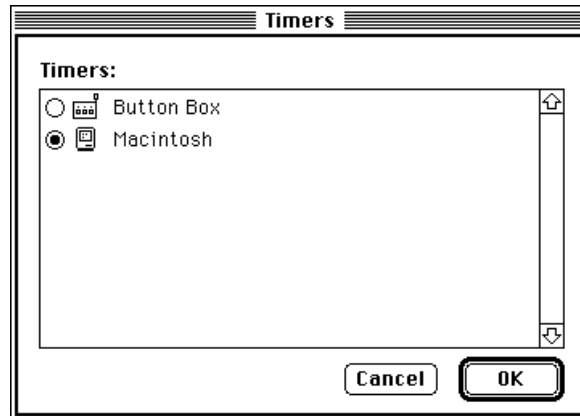


Figure 154 – The Timer dialog

Instructions File

Instructions File specifies a file, the contents of which will be displayed to the subject at the beginning of the experiment. The file must be a plain text file. If the value of this attribute is not set, no starting message will be shown.

While the subject is reading the instructions, the experiment can be precompiled. (See the description of the **Precompile** attribute below, and “6.3 The Trial Monitor”, p238 for more information on precompiling experiments.)

Debriefing File

Debriefing File specifies a file, the contents of which will be displayed to the subject at the end of the experiment. The file must be a plain text file. If the value of this attribute is not set, no ending message will be shown.

Trials per Rest/Num. Rests

These attributes specify the number of rest periods that will occur during the run of the experiment; one or the other of these attributes can be used, but not both.

Trials per Rest schedules rest periods to occur after the specified number of trials. **Num Rests** schedules the specified number of rest periods to occur during the entire experiment; these are spaced evenly across the total number of trials in the experiment.

Rest Duration

Rest Duration specifies the minimum duration (in milliseconds) of the rest period. During this period, the screen will display the message “You may take a rest”. The keyboard and mouse are deactivated during this period as well (unless Command- is pressed, in which case the Break dialog will appear).

Once the rest period has elapsed, the message “Press any key to continue...” will be displayed. Then, pressing a key will begin the next trial of the experiment.

Special

Special sets the state (on/off) of a number of options that modify the way the experiment is run. These are:

Don't show instructions: Causes the instructions and debriefing messages to be skipped, even if the attributes for these messages have been specified.

Don't save screens: Disables PsyScope's default behavior of saving the state of the display before rest periods and when the experiment is interrupted (by pressing Command-), and fully restoring the display when the experiment continues. Saving the display preserves background elements (such as a ports, as well as any additional elements that have been added during the course of the experiment). Disabling this behavior saves memory, especially on large screens, which require a significant amount of memory to store the display. However, the experiment will resume with a blank display, unless actions have been used to explicitly restore its contents.

Don't automatically draw ports: Disables PsyScope's default behavior to draw all port borders at the beginning of each trial; this flag also overrides the **Draw ports when events start** flag. When this option is selected, port borders will only be drawn when the `DrawPortBorders[]` or `DrawAllPortBorders[]` actions are used (see "5.9.4.1 Available Actions", p198).

Draw ports when events start: Causes PsyScope to draw the borders of a port at the beginning of the first event that uses that port, rather than at the beginning of the trial. This option is overridden by the **Don't automatically draw ports** flag.

Don't automatically clear event stimuli: Changes the default value of the **Clearing** event attribute from **Always Clear** to **Don't Clear** (see "Clearing", p180). This causes PsyScope to leave stimuli active until they are cleared using the `ClearStim[]` action (see "5.9.4.1 Available Actions", p198).

Debugging mode: Used by each input and output device as appropriate. The only effects to the standard devices are for the display: the experiment is run in a reduced-size window and some timing interrupts are turned off.

Startrefs default from start: When the starting time for an event is unspecified, it is scheduled to start at the end of the preceding event (i.e. the preceding event in the template's list of events). With this flag on, the default start reference is from the *start* of the preceding event. You should not use this flag with the graphic environment, since it assumes the usual default.

Don't hide cursor: Disables PsyScope's default behavior, which is to hide the cursor while the experiment is running.

Attempt to correct for unusual screen: Disables PsyScope's default behavior, which is to wait for a retrace signal from the display device before writing to the screen. This option allows PsyScope to run with display devices that do not generate a retrace signal. However, it should be used with caution, as it will prevent PsyScope from synchronizing stimulus displays with the screen refresh.

Button Box input only: Disables CMU button box output, even when the button box device is active. This is used to correct for an error in old versions of the button box cable, and should not be used otherwise.

Store data at end of experiment: Causes all experiment data to be stored to disk at the end of the experiment (or when it is interrupted, using `Command-`), rather than after the completion of each trial (which is the default). This saves time during the intertrial interval (by limiting disk access). However, it introduces a risk of data loss if the experiment does not end normally.

Preload all stimuli with precompiling: Causes PsyScope to load (and/or construct) all stimuli used in the experiment into memory at the time the experiment is precompiled, rather than loading the appropriate stimuli at the beginning of each trial. This saves time during the intertrial interval. However, depending upon the number of different stimuli in the experiment, and their size, this may require large amounts of

memory, and may add substantially to the precompile time. (See “6.5.1 Precompiling”, p246 and “ Preloading All Stimuli”, p250.)

Beep at start of rest period: Causes the system beep to be sounded at the start of each rest period. This is useful for alerting the subject or experimenter that a rest period has occurred.

Precompile

Precompile specifies the number of trials to compile before running the experiment. Selecting **All** will cause all of the trials to be compiled; specifying a number will cause PsyScope to precompile that number of trials before beginning the experiment, and to compile the remainder on a trial-by-trial basis. (See “6.3 The Trial Monitor”, p238, for additional information on precompiling experiments).

Resources

Resources specifies a list of files, containing Macintosh resources, to be opened before the experiment is run. The resources contained in these files (e.g., ‘PICT’ and ‘snd’ resources) will be available for use as stimuli in the experiment. See also “6.1.3 Resources”, p216.

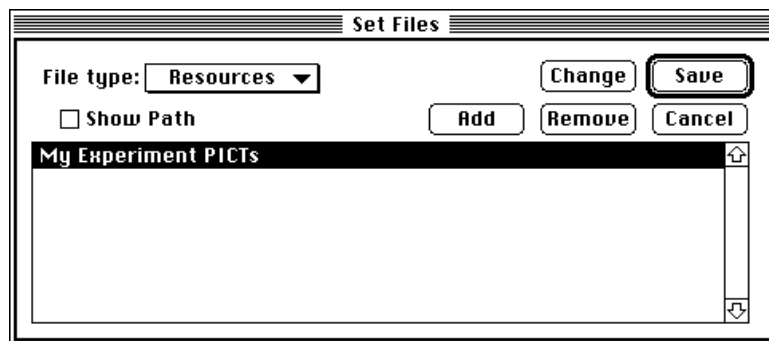


Figure 155 – The Resources dialog

Optimization

Optimization specifies the level of optimization that PsyScope should use in compiling the experiment. This influences both the speed of compilation and the amount of memory that is used. By default, a very limited form of optimization is used — **Optimize constant-declared events only** — in which the experiment designer must specifically an-

notate constant events (and this can be done only through PsyScript; see “Part 4: Scripting Reference, 13.3.6.5 Constant Events in Factor Format”, p387).

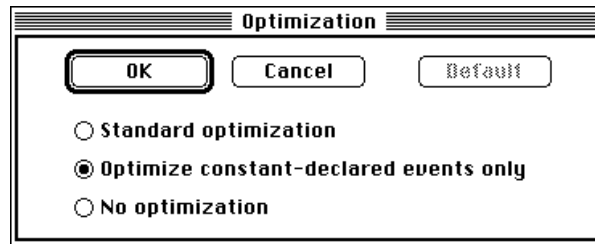


Figure 156 – The Optimization dialog

The more general form of optimization is **Standard optimization**, which attempts to optimize every trial based on table cells selections (and the absence of side-effect operations).

In some cases, **Standard optimization** can actually cost more than it saves (both in terms of speed and memory). This is most likely to be true of complex experiments with very little redundancy across trials, or for experiments where PsyScripted assignment statements are used. In such cases, optimization should be disabled entirely (**No optimization**). (See also “6.3.1 Trial Compilation Statistics”, p241 and “Part 4: Scripting Reference, 13.3.11.3 Factor Format Optimization”, p404.)

Decimal Places

Decimal Places specifies the number of digits that should be printed after the decimal place when a floating-point trial variable is converted to a string (e.g. when a floating-point variable is written to the data file).

BackColor

BackColor specifies the background color in the experiment window. This can be one of the standard colors, or an arbitrary color specified through the standard Macintosh color wheel.

Forecolor

Forecolor specifies the drawing color in the experiment window. This can be one of the standard colors, or an arbitrary color specified through the standard Macintosh color wheel.

BBox Init

BBox Init specifies the state of the CMU button box output lines (including the three LED lights) that will be set when the experiment begins.

5.8.4 Group Attributes

There are no built-in attributes that are particular to groups; only custom and default attributes are specified for a group.

5.8.4.1 Custom Group Attributes

Custom group attributes are used with the **Vary by Group** setting in the Attribute dialogs (see “5.8.1.3 Attribute Dialogs”, p150). When a template or event is owned by more than one group, **Vary by Group** gets a value for the attribute based on which group is currently being used.

For an attribute to use **Vary by Group**, all of the owning groups must have a custom attribute of the same name. This name will appear as one of the group attributes that **Vary by Group** can link to. See also “Vary by Block, Group”, p154.

5.8.4.2 Default Stimulus/Event/Trial Attributes

Values for attributes used by trials and events can be specified at the group level. Unless these defaults are overridden at lower levels of the hierarchy, these value will be inherited by all objects belonging the group. See “5.8.1.1 Attribute Inheritance”, p150 for more information.

5.8.4.3 Group Attributes Dialog

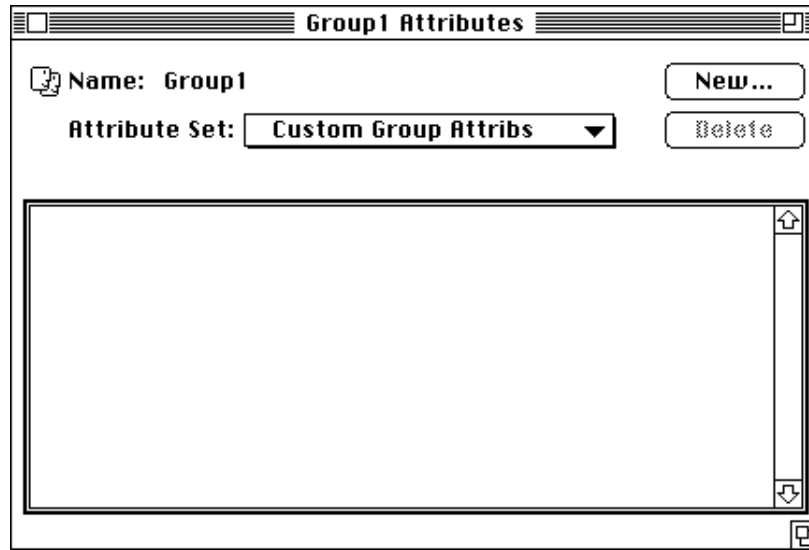


Figure 157 – The Group Attributes window



Figure 158 – Group attribute sets menu

The Group Attributes dialog is one of the standard attribute dialogs, described in “5.8.2 The Standard Attributes Dialog”, p151. The attribute sets in the Group Attributes dialog are as follows:

- Custom Group Attributes** (discussed in “5.8.4.1 Custom Group Attributes”, p169, creation described in “5.8.2.2 Custom Attribute Sets”, p156)
- Default Stimulus Attributes** (described in “5.8.7.3 Stimulus Attributes”, p181)
- Default Event Attributes** (described in “5.8.7 Event Attributes”, p176)
- Default Trial Attributes** (described in “5.8.6.3 Trial Attributes Dialog”, p174)

5.8.5 Block Attributes

There are no built-in attributes that are particular to blocks; only custom and default attributes are specified for a block.

5.8.5.1 Custom Block Attributes

Custom block attributes are used with the **Vary by Block** setting in the Attribute dialogs (see “5.8.1.3 Attribute Dialogs”, p150). When a template or event is owned by more than one block, **Vary by Block** gets a value for the attribute based on which block is currently being used.

For an attribute to use **Vary by Block**, all of the owning blocks must have a custom attribute of the same name. This name will appear as one of the block attributes that **Vary by Block** can link to. See also “Vary by Block, Group”, p154.

5.8.5.2 Default Stimulus/Event/Trial Attributes

Values for attributes used by trials and events can be specified at the block level. Unless these defaults are overridden at lower levels of the hierarchy, these value will be inherited by all objects belonging the block. See “5.8.1.1 Attribute Inheritance”, p150 for more information.

5.8.5.3 Block Attributes Dialog

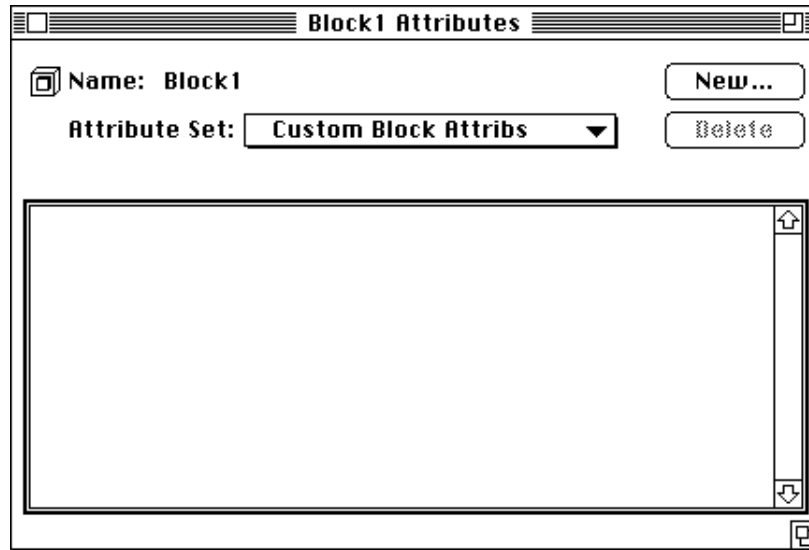


Figure 159 – The Block Attributes dialog



Figure 160 – Block attribute sets menu

The Block Attributes dialog is one of the standard attribute dialogs, described in “5.8.2 The Standard Attributes Dialog”, p151. The attribute sets in the Block Attributes dialog are as follows:

- Custom Block Attributes** (described in “5.8.5.1 Custom Block Attributes”, p171, creation described in “5.8.2.2 Custom Attribute Sets”, p156)
- Default Stimulus Attributes** (described in “5.8.7.3 Stimulus Attributes”, p181)
- Default Event Attributes** (described in “5.8.7 Event Attributes”, p176)
- Default Trial Attributes** (described in “5.8.6.3 Trial Attributes Dialog”, p174)

5.8.6 Trial Attributes

Trial attributes control features of individual trials in the experiment. Custom and default attributes may also be set in templates.

5.8.6.1 Custom Template Attributes

Custom template attributes are used with the **Vary by Template** setting in the Attribute dialogs (see “5.8.1.3 Attribute Dialogs”, p150). When an event is owned by more than one template, **Vary by Template** gets a value for the attribute based on which template (or factor table) is currently being used.

For an attribute to use **Vary by Template**, all of the owning templates must have a custom attribute of the same name. This name will appear as one of the template attributes that **Vary by Template** can link to. See also “Vary by Template”, p154.

5.8.6.2 Default Stimulus/Event Attributes

Values for attributes used by events can be specified at the template level. Unless these defaults are overridden at the event level, these values will be inherited by all events belonging the template. See “5.8.1.1 Attribute Inheritance”, p150 for more information.

5.8.6.3 Trial Attributes Dialog

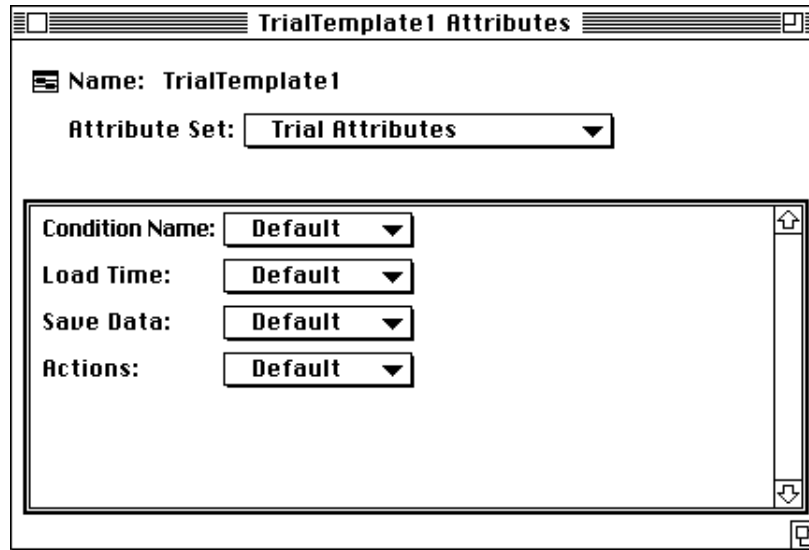


Figure 161 – The Trial Attributes dialog

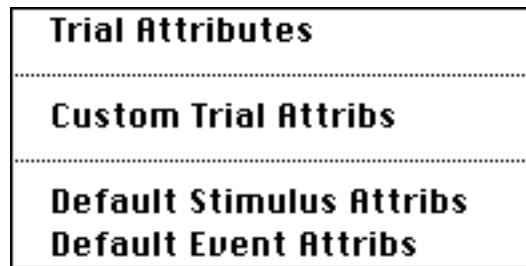


Figure 162 – Trial attribute sets menu

The Template Attributes dialog is one of the standard attribute dialogs, described in “5.8.2 The Standard Attributes Dialog”, p151. The attribute sets in the Template Attributes dialog are as follows:

Trial Attributes (described below)

Custom Trial Attributes (described in “5.8.6.1 Custom Template Attributes”, p173, creation described in “5.8.2.2 Custom Attribute Sets”, p156)

Default Stimulus Attributes (described in “5.8.7.3 Stimulus Attributes”, p181)

Default Event Attributes (described in “5.8.7 Event Attributes”, p176)

5.8.6.4 Standard Trial Attributes

Condition Name

Condition Name specifies a pattern that will be used to generate a condition name to be written to the data file. By default, the condition name of a trial is the concatenation of the names of all the levels of factors (and items of lists) that are current for the trial.

The syntax of a condition name pattern is:

prefix<separators>suffix.

The names of the current levels will be inserted in place of the angle brackets (“<nb>”); the level names will be concatenated, separating the names with *separator*.

For example, a crossing with levels “green”, “small”, and “verb” and a condition name pattern “A-<.>-Z” would give “A-green.small.verb-Z” for the condition name.

If the angle-brackets do not appear in the condition name pattern, level names will not be used at all.

ITI

ITI specifies the *minimum* time to wait from the end of the previous trial to the start of the current trial. During this time, the trial will be built in memory, and all stimuli will be loaded in. If this internal overhead is greater than the time specified in **ITI**, the start of the trial will be delayed even further until all overhead has been completed.

Actions

Actions specifies condition-action pairs for trial actions; these actions will be active for the duration of the trial. Trial actions with **Start** and **End** conditions will occur at the start and end of the trial, respectively. See “5.9 Conditions and Actions”, p192.

5.8.7 Event Attributes

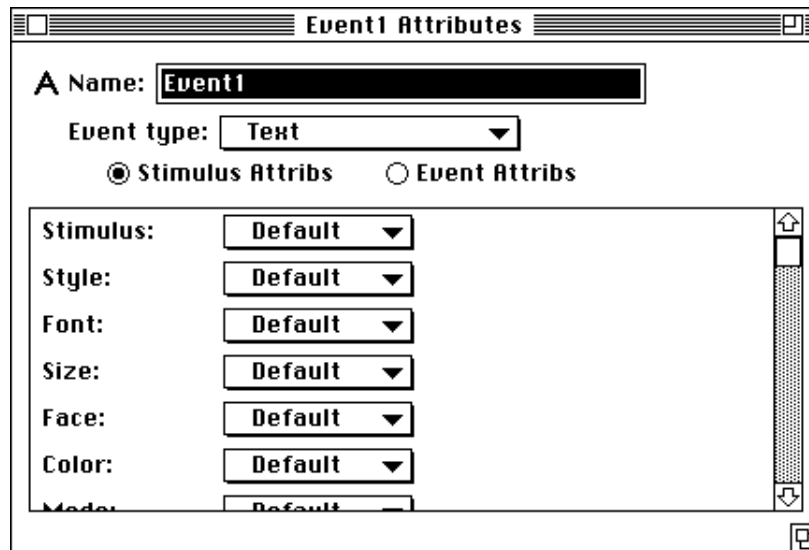


Figure 163 – Event Attributes dialog

The Event Attributes dialog is one of the Standard Attributes dialogs, described in “5.8.1.3 Attribute Dialogs”, p150. The two attribute sets — stimulus attributes and event attributes — are represented by the radio buttons **Stimulus Attribs** and **Event Attribs**.

Event attributes are common to all events. They include the event’s duration and any actions that will occur during the event (see “5.9 Conditions and Actions”, p192).

Stimulus attributes are specific to the type of the event. They include the actual stimulus that will be presented and its features. For example, **Text** events have stimulus attributes for the font, size, color, etc. of the displayed text. Sound event stimulus attributes include the sound’s volume and channel.

The event attributes are described below in “5.8.7.2 Event Attributes”, p179. Stimulus attributes are described in “5.8.7.3 Stimulus Attributes”, p181.

5.8.7.1 Event Types

Clicking on the **Event type** pop-up menu opens the menu shown below. To change the current event type, select a new event type from the menu. Each event type is described below.

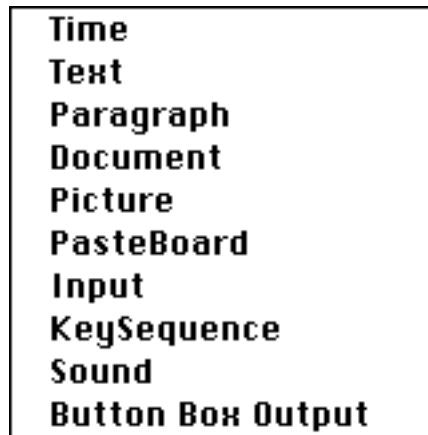


Figure 164 – Event types menu

Time Event Type

Events of type **Time** present no stimulus. They are used for timed delays, or for null events to which actions can be attached.

Text Event Type

Events of type **Text** present a line of text on the screen. The size, font, color, style, and position of the text can be manipulated through the event's stimulus attributes.

PICT Event Type

Events of type **PICT** present a picture stored in a PICT file or resource on the screen.

If the picture is in a resource, the resource file is usually specified in the **Resources** experiment attribute (see “5.8.3.4 Standard Experiment Attributes”, p161 and “6.1.3 Resources”, p216).

Document Event Type

Document events present text from a file on the screen; unlike the **Text** event type, the text will be flowed within its display port (i.e. the text is drawn starting from the upper left corner of the port, and lines longer than the width of the port will be wrapped to the next line).

The file used by a **Document** event should be a plain text file. However, the file text can contain special formatting commands to change the face of parts of the text (e.g. underline

or bold a section). The formatting commands are part of the text, and will only be used when the text is displayed by the event. The available commands are:

- @b – Start making text bold
- @i – Start making text italic
- @u – Start making text underlined
- @e – End all formatting

For example, given this text:

The @iquick@e sly @bfox@e jumped @uover@e the lazy @b@i@udog@e.

the **Document** event type will display:

The *quick* sly **fox** jumper over the lazy ***dog***.

No effort is made to present **Document** text quickly or with millisecond accuracy. To present **Documents** multi-line text stimuli quickly, use the **Pasteboard** event type.

Paragraph Event Type

Paragraph events are identical to **Document** events, except that the text to be displayed is specified directly, instead of through separate file.

Pasteboard Event Type

Pasteboard events present a combination of screen sub-stimuli with many event types — **Text**, **PICT**, **Document**, and/or **Paragraph** — as a single event. Each sub-stimulus in a pasteboard still has the full set of stimulus attributes appropriate to its type, but no event attributes.

A **Pasteboard** event is essentially a picture of its constituent stimuli. PsyScope presents it like a **PICT** stimulus — as quickly as possible. If you need to present a **Paragraph** or **Document** stimulus quickly, you can do so by putting it in a pasteboard.

A pasteboard has a display port like any other screen event type. When a pasteboard picture is being generated, sub-stimuli are drawn in their own ports, but the drawing for each sub-stimulus is clipped to the area of its port that overlaps the pasteboard's port.

Key Sequence Event Type

A **Key Sequence** event is like a **Paragraph** event, except that the subject can type during the event (using the keyboard) and the typed characters are displayed after the prompt paragraph. You will usually set the duration of a key sequence to be `Key[Return]`; this way, the subject can type a response and terminate it by hitting the Return key.

The characters typed by a subject during a **Key Sequence** event are stored in a key sequence buffer. The contents of this buffer can be recorded on every line of the data file by turning on **Key sequence** in the **Data Info** experiment attribute (see “Data Info”, p161). The buffer is erased each time a **Key Sequence** event starts, and a character is added after every key press by the subject.

The best way to record a **Key Sequence** response is to add an RT[] action to the event with the **End** condition. This way, the key sequence buffer contains the subject's entire response when the RT is recorded.

Sound Event Type

Sound events play labeled sounds from SoundEdit™ or Sound Designer II™ sound files. Unless otherwise specified, the duration of a **Sound** even is **Self Terminate**.

If another duration is specified, it will either cause the sound to be cut off — unless **Don't Clear** is specified for the **Clearing** attribute of the event (see “Clearing”, p180) — or it will cause the length of the event to extend beyond the end of the sound. If no label is given, the entire file will be played.

5.8.7.2 Event Attributes

Duration

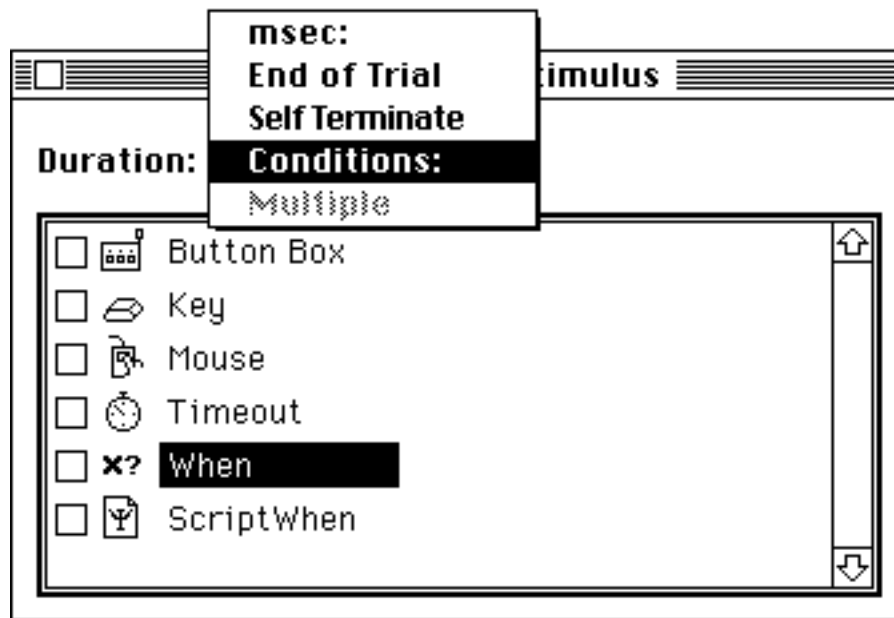


Figure 165 – The Duration dialog

You can set an event's duration in four ways:

- a) by assigning it an absolute value (**msec!**).
- b) by allowing it to last until the end of the trial (**End of Trial**).
- c) by allowing the stimulus to run its course (**Self Terminate**). This is only appropriate for certain types of events (e.g. **Sound**).

- d) by allowing the event to continue until the subject has entered an input, or some other condition has occurred (**Conditions**). The required input or condition is called a *terminating condition*.

In the last case, you must specify a set of conditions which terminate the event. In this mode, the Duration dialog is much like the Conditions dialog (see “5.9.2 Conditions Dialog”, p194). There is additionally, however, a **Timeout** condition, which specifies how long to wait, in milliseconds, for the rest of the conditions to occur. (Of course, **Start** and **End** are not valid duration conditions.)

When a new event is created, PsyScope assigns it a default duration value, which depends on the event’s type. For example, **Time** events are assigned a default duration of 500 msec, while **Sound** events play out the sound.

Actions

Actions specifies a list of sets of conditions-actions action pairs that will be activated when the event runs. See “5.9 Conditions and Actions”, p192.

Clearing

Clearing specifies the manner in which the stimulus will be cleared. The possible values are:

Always Clear: The stimulus will be cleared automatically at the end of the event. This is the default.

Don't Clear: The stimulus will not be cleared automatically. In this case, the stimulus will never be cleared, or it can be cleared using the `ClearStim[]` action (see “5.9.4.1 Available Actions”, p198).

Use Mask: When the stimulus would normally be cleared, it will be masked instead.

The **Use Mask** option is valid only for event types that include a notion of masking. The **Text** event type most often utilizes masking; **Text** events have a **Text Mask** attribute, which is a character used to replace all of the characters of the displayed text.

Tag

Tag is useful for labeling events in the data file. PsyScope does not use the **Tag** attribute for any other purpose. See also “Data Info”, p161.

Load Time

When **Load Time** is not specified, the stimulus is loaded at the beginning of the trial and unloaded at the end of the trial. Specifying **Load Time** causes the stimulus to be loaded immediately before it is displayed, and unloaded immediately after.

The value of the **Load Time** attribute specifies how much time — in milliseconds — to allow for loading the stimulus. If loading takes less than this amount of time, everything will

run on schedule; if loading actually takes longer than this, event execution will be delayed until the stimulus can be loaded.

See also “6.5.2.1 Load Time”, p247.

5.8.7.3 Stimulus Attributes

The stimulus attributes for an event determine features of the stimulus that are particular to the type of the event. Below is a list of the stimulus attributes for each event type, along with a short description of what each attribute does.

Time Attributes

The **Time** event type has no stimulus attributes.

Text Attributes

Stimulus: A string containing the text to be displayed.

Style: The style of the text. Specifies font, size, face, color, and mode as one attribute; this specification is overridden by the individual attributes for these qualities. See “5.8.8.3 Style Dialog”, p185.

Font: The font in which the text will be drawn.

Size: The point size of the text.

Face: The face of the text (e.g. bold, italic).

Color: The color in which the text will be drawn.

Mode: The drawing transfer mode of the text.

Port: The stimulus port in which the text will be drawn. (See “5.8.8.4 Ports and Positions Dialogs”, p187.)

Position: The position of the text within the stimulus port. (See “5.8.8.4 Ports and Positions Dialogs”, p187.)

Degrade: The amount by which the text should be degraded.

Special: Special qualities of the stimulus to turn on or off:

Follow previous text: If this flag is on, the **Position** attribute is not used directly to position the stimulus; instead the text is positioned just after the end of the previous text stimulus. The **Position** attribute is used indirectly: the **Position** horizontal value is used to specify an extra horizontal offset. (The Positions dialog does not currently recognize the different uses of the **Position** attribute.) When **Follow previous text** is on, lines of text are automatically “wrapped” within the port.

Same as previous text This flag works like **Follow previous text**, but the stimulus is positioned at the same place as the previous text stimulus. In this case, both the horizontal and vertical values in the **Position** attribute are used as offsets.

Draw in masked form: If this flag is on, the text is always drawn in masked form. See “Clearing”, p180.

Text Mask: The character to use when masking the stimulus. See “Clearing”, p180.

Flip: Whether the stimulus should be flipped horizontally or vertically before being displayed.

PICT Attributes

Picture: The name of the PICT file or resource to be displayed.

Port: The stimulus port in which the picture will be drawn (see “5.8.8.4 Ports and Positions Dialogs”, p187). Unless the **Draw actual size** flag is on (in the **Special** attribute; see below), the PICT will be scaled to fit into this port.

Mode: The drawing transfer mode for the PICT.

Degrade: The amount by which the picture should be degraded.

Special: Special qualities of the stimulus to turn on or off:

Keep picture in memory: Specifies that the pict should be maintained in memory after it is read from disk. See “Keeping Stimuli in Memory”, p251.

Use default colors: If the default drawing colors have been changed (see “Forecolor”, p168 and “Backcolor”, p168), color PICTs may need this flag to display in reasonable colors (by ignoring the foreground and background color settings).

Draw actual size: If this flag is on, the PICT will be drawn in its native size, centered in its port. PICTs are usually scaled to fit in their ports.

Depth: The depth in pixels of the stimulus; this controls the number of colors to use when drawing the picture and the amount of memory it will use when stored.

Flip: Whether the stimulus should be flipped horizontally or vertically before being displayed.

Document Attributes

File: The name of the document to be displayed.

Style: The style of the text. Specifies font, size, face, color, and mode as one attribute; this specification is overridden by the individual attributes for these qualities. See “5.8.8.3 Style Dialog”, p185.

Font: The font in which the text will be drawn.

Size: The point size of the text.

Face: The face of the text (e.g. bold, italic)

Color: The color in which the text will be drawn.

Mode: The drawing mode of the text.

Port: The stimulus port in which the text will be drawn. (See “5.8.8.4 Ports and Positions Dialogs”, p187.)

Paragraph Attributes

Paragraph: The paragraph of text to be displayed.

Style: The style of the text. Specifies font, size, face, color, and mode as one attribute; this specification is overridden by the individual attributes for these qualities. See “5.8.8.3 Style Dialog”, p185.

Font: The font in which the text will be drawn.

Size: The point size of the text.

Face: The face of the text (e.g. bold, italic)

Color: The color in which the text will be drawn.

Mode: The drawing mode of the text.

Port: The stimulus port in which the text will be drawn. (See “5.8.8.4 Ports and Positions Dialogs”, p187.)

Pasteboard Attributes

Stimuli: A list containing **Text**, **PICT**, **Document**, and **Paragraph** stimuli to be displayed in this pasteboard. See “5.8.8.2 Stimuli Dialog”, p185.

Port: The stimulus port in which the pasteboard will be drawn. (See “5.8.8.4 Ports and Positions Dialogs”, p187.)

Degrade: The amount by which the pasteboard should be degraded.

Depth: The depth in pixels of the stimulus; this controls the number of colors to use when drawing the pasteboard and the amount of memory it will use when played.

Flip: Whether the stimulus should be flipped horizontally or vertically before being displayed.

Key Sequence Attributes

Prompt: The prompt text to be shown to the subject. This is in the same format as a **Paragraph** event's **Paragraph** attribute.

Style: The style of the text. Specifies font, size, face, color, and mode as one attribute; this specification is overridden by the individual attributes for these qualities. See "5.8.8.3 Style Dialog", p185.

Font: The font in which the text will be drawn.

Size: The point size of the text.

Face: The face of the text (e.g. bold, italic)

Color: The color in which the text will be drawn.

Mode: The drawing mode of the text.

Port: The stimulus port in which the prompt and response text will be drawn. (See "5.8.8.4 Ports and Positions Dialogs", p187.)

Sound Attributes

Sound: The label (or region) of the sound to play. If this is not specified or empty (""), the whole file will be played.

File: The SoundEdit™ or Sound Designer II™ file which contains the sound.

Volume: The volume of the sound. 0 = lowest; 255 = loudest.

Channel: Whether to play the sound through the left or right channel.

Feature: Special features of the sound:

Keep sound in memory: Specifies that the sound should be maintained in memory after it is read from disk. See "Keeping Stimuli in Memory", p251.

Play in parallel: Allows the sound to be played simultaneously with other sounds. If you try to run two sound events simultaneously without this flag, one sound will wait for the other to end.

5.8.8 Stimulus Attribute Dialogs

The various stimulus attributes of events require a number of different dialogs. These dialogs are opened in the event attributes dialog for the event.

5.8.8.1 Stimulus Dialog

The Stimulus dialog sets the value of the stimulus for an event. The actual dialog that appears differs according to the event type. It can be standard single line text, free text field, standard Macintosh file finding dialog, or the picture finding dialog.

5.8.8.2 Stimuli Dialog

Certain kinds of events (such as the **Pasteboard**) are made up of several stimulus components. The set of components is modified through the **Stimuli** attribute of the event.

Setting the stimulus list opens the Stimuli dialog. This dialog simply displays a list of stimuli with **New**, **Delete**, and **Select** buttons. Stimuli are added and removed from this list in the standard way. Double-clicking on an item in the list or clicking **Select** opens the Stimulus Attribute dialog for that stimulus. (The Stimulus Attribute dialog is like the Event Attributes dialog, except that it only displays stimulus attributes.)

5.8.8.3 Style Dialog

The attributes for several types of events use the Style dialog. In particular, the **Style**, **Font**, **Size**, **Face**, **Color**, and **Mode** attributes are all set via the Style dialog. For attributes other than **Style** only the part of the dialog pertaining to the selected attribute will be active.

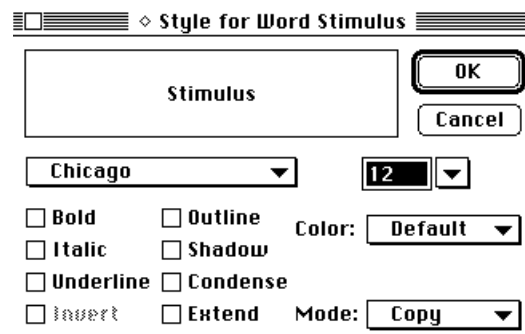


Figure 166 – The Style dialog

The main component of the Style dialog is a box displaying a sample stimulus, in the style that is currently set. Below this box are several controls, namely, the font menu, checkboxes for various text faces, the point size and size menu, the color menu, and the drawing mode menu.

The Font Menu

The font menu is a pop-up menu of all the fonts currently available to PsyScope. There is no guarantee that a font selected in this list will be available on another machine. (If a font cannot be found, the system font (**Chicago**) is used.)

Text Face Checkboxes

These eight checkboxes — **Bold**, **Italic**, **Underline**, **Invert**, **Outline**, **Shadow**, **Condense**, and **Extend** — affect the style of text in the usual way.

The Point Size

Size of the text is set through a text box to the right of the font menu. The size can be changed by pulling down the menu beside the box and selecting one of the standard sizes, or by typing a new size directly into the box.

The Color Menu

This menu selects the color of the stimulus. Possible selections are **Default** (see “Forecolor”, p168), any of the standard colors, or **Other...**

Selecting **Default** causes the sample stimulus to be displayed in black, and the actual stimulus to be displayed in whatever color is specified for in the **ForeColor** attribute of the experiment.

Selecting **Other...** opens the standard system color picker dialog, from which you can select any color available to the system.

The Mode Menu

This menu selects the drawing mode for the stimulus, which affects the way the stimulus drawing interacts with the background or acts when it is drawn over another stimulus. The choices are **Copy**, **Or**, **Xor**, **Erase**, **Inverse**, **InuOr**, **InuXor**, and **InuErase**:

Copy simply replaces anything in the destination screen area with the text stimulus, writing over that area without regard for what was already there.

Or draws the text without affecting screen pixels except where the pixels for a letter is placed, thus “overlaying” the destination area with the text.

Xor draws the text without affecting pixels except where the pixels for a letter is placed; for these pixels, pixels which are off will be turned on, and pixels which are already on will be turned off.

Erase inverts the text before it draws it. In the standard black-on-white mode, it essentially writes in white instead of black.

Inverse, **InuOr**, **InuXor**, and **InuErase** perform similar operations, but invert the text first.

5.8.8.4 Ports and Positions Dialogs

Screen events (**Text**, **PICT**, **Document**, **Paragraph**, and **Pasteboard** are the standard ones) require a display position. This is accomplished in PsyScope by giving the event a *port*.

Ports are independent shapes that you manipulate with the Ports/Positions dialog. A port is basically a rectangle, round-cornered rectangle, or oval on the screen. It also has a frame and a collection of positions. You can specify a port size in absolute pixels or percentages of the screen. You can specify its location using pixel offsets, percentages, or alignments with an edge on the screen.

Text events need both a port and a *position* within the port. A single port can own many positions.

A position is can be specified relative to its owning port or to the screen, using absolute pixels or with percentages. A position also has an alignment parameter which specifies how text is drawn relative to the position (centered, to the right, to the left, above, or below).

The default port for stimuli in a pasteboard is the pasteboard's port. When a port is specified for a stimuli, the port is still defined relative to the screen, and then all stimulus ports are clipped to the pasteboard's port.

*Advanced Note: Ports have a built-in position called the **hot spot**; it is always specified relative to the port. When you set the location for a port, you are actually positioning the hot spot relative to the screen. The hot spot's position relative to the screen and relative to the port give the port its location. An example use of the hot spot is to align one of the port's vertical edges to the center of the screen.*

The port and position of a stimulus must be set separately, since they are controlled by separate attributes. The default port is the entire screen and the default position is centered within the port.

The same dialog is used to set ports and positions, but it operates in two modes. When you open it as the Ports dialog, you can create ports and positions. When you open it as the Positions dialog, you can only create positions and only for the stimulus' current port.

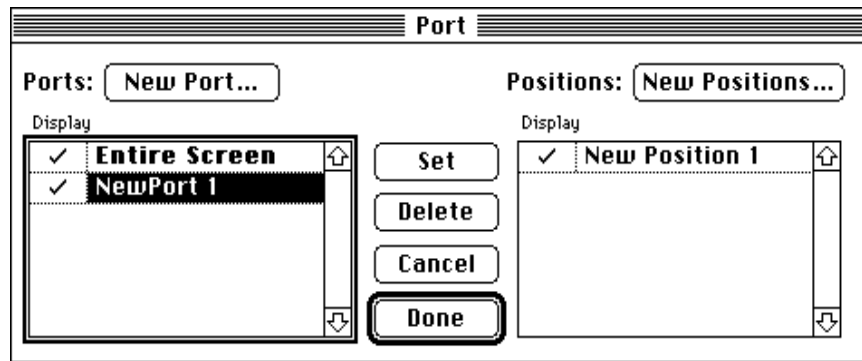


Figure 167 – The main Ports dialog

When the Ports/Positions dialog is opened, it blanks the screen and opens the dialog shown in the figure above. Behind the dialog, the points and positions are drawn exactly where they would be in a run of the experiment.

On the left is a list of all ports in the script. When a port is selected, the list of positions owned by that port is shown on the right. In positions mode, the port selection in the left list cannot be changed.

The **Display** checkmark lets you toggle the display of the port in the background area. This has no affect on how the experiment is executed.

The background graphic versions of ports and positions can be dragged and resized window-style.

Double-clicking on a position or port in the display or in the list opens a dialog to control the port or position attributes more precisely (see below).

New ports and positions are created with the **New Port...** and **New Positions...** buttons, both of which open the New Port/Positions dialog (see below). Positions that you create are added to the currently selected port.

To actually assign a value to the attribute that you clicked to open the dialog, you must click the **Set** button. In port mode, the currently selected port will be made bold in the list, and that port will be written out for the attribute. Similarly, in positions mode, the currently selected position is used.

Hitting **Cancel** will revert all changes made in the dialog since it was opened, including the creation of new ports and positions.

New Port Dialog

Hitting the **New Port...** button in the Positions dialog opens the New Port dialog. This dialog allows you to name the new port and add some initial positions to the port.

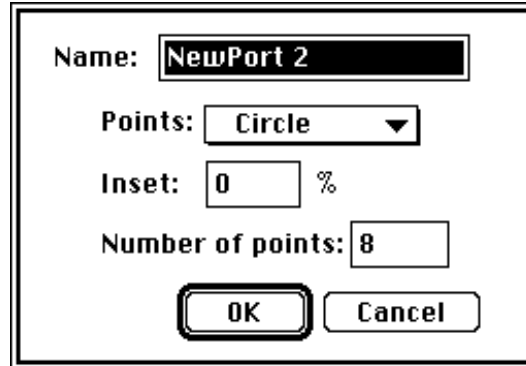


Figure 168 – The New Port dialog

The top text field in the dialog specifies the new port's name. The **Points** pop-up menu sets the number and arrangement of points to be created for the port.

If an arrangement besides **No Points** or **One Point** is selected, then an **Inset** field will appear; this percentage value specifies how far in from the edge of the port, as a percentage of the port's width, the points should be placed. For instance, a circle at 25% will have a radius that is 1/4 the size of an edge of the port (for a square port). Negative values for the **Inset** are OK.

If the **Circle** arrangement is selected, a **Number of points** field will appear. The number you type in this field is the number of points that will be used in making the circle.

New Positions Dialog

The New Positions dialog works just like the New Port dialog, except that the name field is usually missing. If **One Point** is selected in the pop-up, however, that point can be named with the text field.

Port Info Dialog

The Port Info dialog is opened by double-clicking on a port in the list in the Positions dialog, or by double-clicking on its graphic display in the background. This dialog gives you greater control over the details of a port.

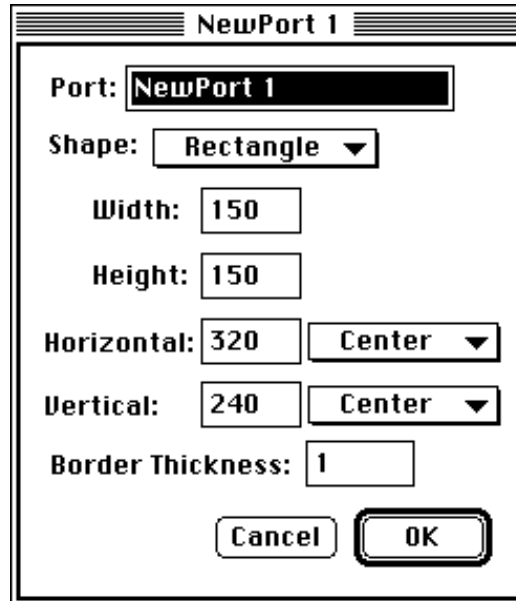


Figure 169 – The Port Info dialog

The text field at the top of the port contains the port name. Below the name is a Shape pop-up, which controls the shape of the port. The possible shapes are **Rectangle**, **Rounded**, and **Oval**.

The **Width** and **Height** of the port can be specified by an absolute number of pixels, or as a percentage of the screen. For instance, to make the port half as wide as the screen, set **Width** to “50%”.

The **Horizontal** and **Vertical** position of the port can also be in absolute pixels or percentage, or the port can be aligned to an edge of the screen. If the port is not aligned to an edge of the screen, the absolute pixel offset or percentage offsets are applied to the center of the port (unless the hot spot has been modified, see “5.8.8.4 Ports and Positions Dialogs”, p187).

The text box at the bottom is used to specify the thickness of the border that is drawn around a port. A thickness of “0” makes the border invisible.

Position Info Dialog

The Position Info dialog is opened by double-clicking on a position in the list in the Positions dialog, or by double-clicking on its graphic display in the background. This dialog gives you greater control over the position definition.

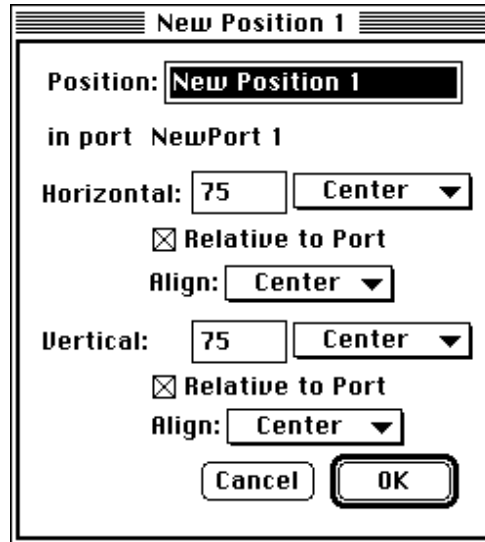


Figure 170 – The Position Info dialog

The point name is shown in the top text field; the port that owns the position is also shown.

The **Horizontal** and **Vertical** values for a point are handled the same as for ports. The location of positions can be specified relative to its owning port, or relative to the screen; this is controlled with the **Relative to Port** checkboxes.

The **Align** pop-up menus specify how text drawn at the position should be located relative to the position; the alignment is demonstrated in the graphic display using the position name.

Hot Spot Dialog

The hot spot (see note at “5.8.8.4 Ports and Positions Dialogs”, p187) has parameters like a regular position, so the dialog is mostly the same. The hot spot is always positioned relative to the port. Since there is no alignment value, the **Relative to Port** checkboxes and **Align** pop-ups are missing in the dialog.

There is an extra checkbox, **Auto-position**. If this is checked, then the hot spot is automatically moved to give the expected port alignments when values in the Port Info dialog are changed. Unchecking this box keeps the hot spot definition fixed.

5.9 Conditions and Actions

Actions are operations that can be performed during the running of a trial. Unlike events, they have no duration, and they trigger when one of a set of **conditions** occurs. Anything that can be done by the Trial Manager can be done via an action— running or ending an event, showing, clearing, or masking a stimulus, retrieving and storing response information, setting variable values, as well as miscellaneous other things. (For the complete list of available actions, see “5.9.4.1 Available Actions”, p198.)

Most actions take parameters which determine how they will be performed. The parameters are shown within brackets. For example, the `Beep[]` action plays “correct beep” if no parameters are specified, or any available ‘snd ’ resource that is given as a parameter (e.g., `Beep[“Wild Eep”]`). `RT[]` is an action that records a response in the data file (or to a trial variable) with the time at which it was received and the state of all active input devices at that time (see “ Input Devices”, p164 for a description of how to specify which input devices are active).

Each action has a set of conditions under which it will be performed. The types of conditions are as follows:

- 1) Input states, pertaining to any of the available input devices (**Mouse**, **Key**, and **Button Box**, plus any custom input devices that have been installed in the version of PsychoScope that you are running).
- 2) The start of the event or the trial (**End**).
- 3) The end of the event or the trial (**Start**).
- 4) A trial variable expression (**When**); see “5.10 Trial Manager Variables”, p205.
- 5) A script evaluation (**ScriptWhen**).

Each action has a specific time during which it is **active**, or waiting for one of its conditions to occur so that it can execute. An action becomes active when it is **posted** by an event or trial that has the action in its **Actions** attribute; events and trials post all of their actions when they are themselves started.

Unless otherwise specified, an action will be active until the end of the event/trial that posted it, or until it is triggered once. This can be changed in a couple of ways: by modifying the *active until* setting for the action, or by allowing multiple *instances* of the action.

The **active until** setting for an action controls the maximum time frame in which the action will be active. An action can be made active until:

- **End of This Event** – the end of the posting event
- **End of [Event]** – the end of some arbitrary event in the trial
- **End of Trial** – the end of the entire trial

- **At Least One Instance** – after at least one instance is triggered or the end of the posting event, whichever is later
- **All Instances** – after all of its instances have been triggered

The *instances* setting of an action determines how many times it can be triggered by its conditions. By default, the number of instances is set to 1, but you can specify as many as 32,767. “-1” can be used to specify an infinite number of instances. (In this case, the active until setting should not require all instances to be triggered!) If an action can be triggered by more than one condition, instances are counted independent of which condition triggers the action.

The meaning of the **Start** and **End** conditions changes depending on whether an action is posted by an event or by a trial. If it is posted by an event (i.e. the action is in an **Actions event** attribute), **Start** means the end of the event; if the action is posted by a trial (i.e. the action is in an **Actions trial** attribute), **Start** means the start of the trial. **End** changes meaning similarly.

5.9.1 Conditions and Actions Dialog

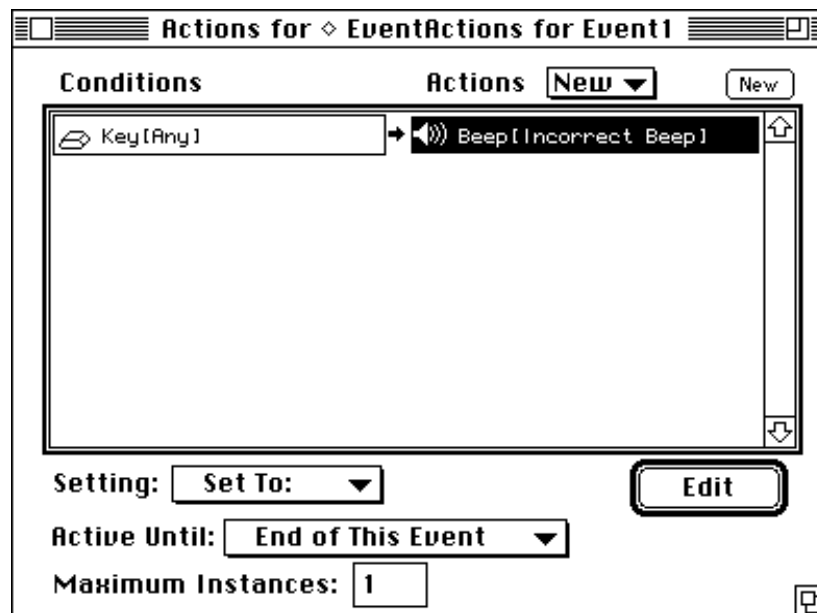


Figure 171 – The Actions dialog

The Conditions and Actions dialog is used to specify condition-action pairs for an event or a trial. The main element of the dialog is a list containing condition-action pairs.

Each condition-action pair consists of two boxes: the left box contains a list of conditions and the right box contains the list of actions to be triggered by these conditions. To add an empty condition-action pair to the list, click the **New** button.

To add, delete, change a condition, double-click on conditions box; this will open the Conditions dialog (see “5.9.2 Conditions Dialog”, p194).

To add individual actions for a condition-action pair, select either box and click on the **New** menu that appears above the actions column. The **New** menu contains all available actions. Selecting one of the actions will add it to the condition-action pair.

Double-clicking on an action lets you to set the parameters of the action through the Parameters dialog (see “5.9.5 Parameters dialogs”, p205).

To delete an action, select it and press the Delete key. Actions can be cut and pasted using the standard **Edit** menu items.

The **Setting** pop-up menu appears at the bottom of the dialog when something is selected in the condition-action list; it refers to either the entire condition list or to a particular action — whichever is selected. This menu is just like the pop-up menu for items in the Attribute dialog (see “5.8.2 The Standard Attributes Dialog”, p151).

Also, when an action is selected, an **Active Until** pop-up and **Maximum Instances** field appear at the bottom of the dialog for editing active until and instances for the action. See “5.9 Conditions and Actions”, p192 for a discussion of these parameters.

5.9.2 Conditions Dialog

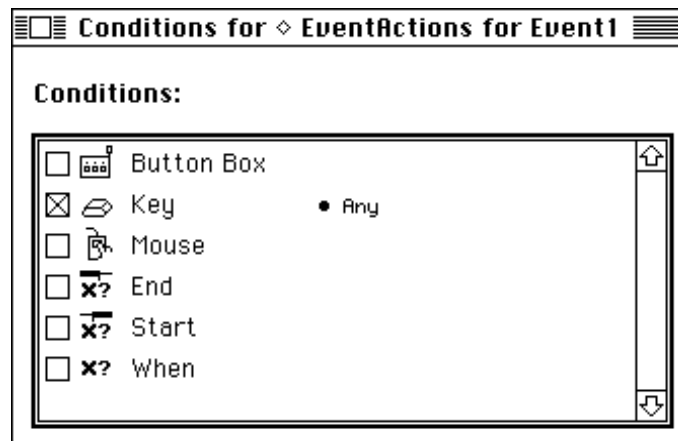


Figure 172 – The Conditions dialog

The Conditions dialog allows you to specify the conditions under which actions will occur. It lists all of the possible triggering “devices”, with checkboxes for turning them on and off. The standard available devices are **Mouse**, **Key**, and **Button Box**, plus the “virtual” devices **Start**, **End**, **When**, and **ScriptWhen**. More devices may appear in your list if you have Extensions installed in your copy of PsyScope.

When a device is checked, bulleted information appears to the right of the device name; this is the device's input parameter. For example, the **Key** device takes parameters specifying which keys to look for. The device together with its parameters make a single condition.

To change the device parameter, click on the bulleted information. The dialog that is opened will be specific to the input device. These dialogs are described below for each of the standard devices.

5.9.3 Condition Parameter Dialogs

5.9.3.1 Button Box Parameter Dialog

The **Button Box** condition is used to detect responses through the CMU button box. The button box can detect button presses and releases, voice start and stops (through a microphone plugged into the button box), and external line inputs.

A single button box condition can be used to watch multiple button, voice, and line inputs. However, the on/off “direction” of the response must be the same for all of the inputs; e.g. you can look for either of two buttons to be pressed, but a condition cannot look both for one button to be pressed and another to be released.

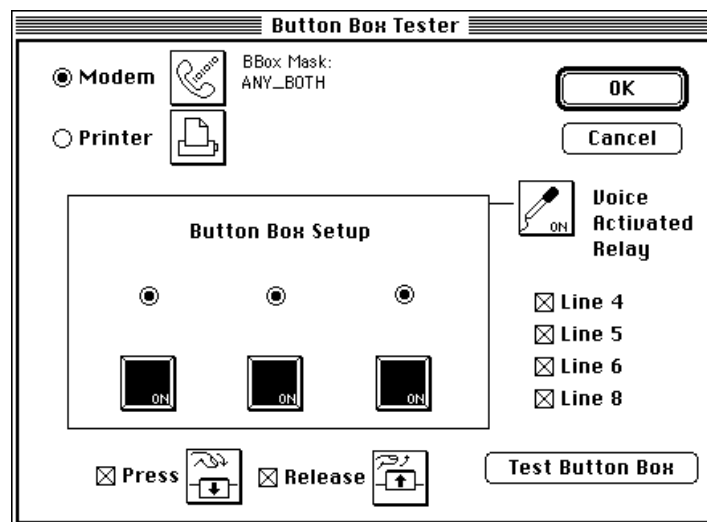


Figure 173 – The Button Box Condition Parameter dialog

The **Press** and **Release** checkboxes at the bottom of the dialog control whether the condition watches for button presses or releases (or voice starts or ends, or line input ons or offs).

The button icons, microphone icon, and line input checkboxes control which parts of the dialog will be watched. To toggle a button on or off, click on it.

The **Modem** and **Printer** radio buttons at the top of the dialog set where PsyScope expects to find the button box connected to the machine. Unlike the rest of the dialog, which applies

to the particular condition being set, this setting is stored globally (as an experiment attribute).

The **Test Button Box** button at the bottom right will put the dialog into testing mode. The button box should be connected and turned on before this button is hit. In testing mode, you can hit buttons and get feedback in the dialog to test the button box. You can also test the output lights using the light icons in the dialog above the buttons; clicking on a light should toggle its state both in the dialog and on the real button box. Click on **End Testing Mode** to return to the normal mode of the dialog.

5.9.3.2 Key Parameter Dialog

The **Key** device is used to watch for input from the standard Macintosh keyboard. This device can detect single key presses or combinations of keys using Command-, Shift-, Control-, and Option- modifiers.

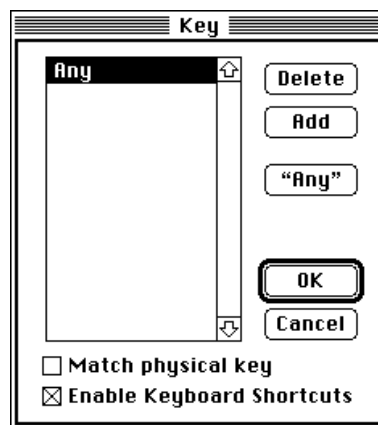


Figure 174 – The Key Condition Parameter dialog

For each **Key** condition, you can detect for multiple key combinations; these are listed in the main part of the dialog. The keyword “Any” (added to the list by hitting the “Any” button) can be used to match any (non-modifier) key press.

Clicking the **Add** button adds a new line to the list of key combinations (setting it to **Any** as the default).

If no items are selected in the list, you can also add a combination by typing it. You can change a combination by selecting it in the list and typing a new combination.

Clicking on the **Delete** button when a combination is selected will delete the combination.

When you set a key combination through the **Key** condition parameter dialog, it usually reads your input combination in a keyboard-independent manner. When the **Match physical key** checkbox is on, the dialog reads the keyboard directly and stores information for exactly the key you pressed. This can be useful if you want to use the function keys on an extended keyboard.

When the **Enable keyboard shortcuts** checkbox is on, you can use the usual keyboard commands to manipulate items in the dialog: e.g. hit Return for the **OK** button, hit the Delete key to delete a combination, etc. You may need to turn this checkbox off in order to set certain key combinations.

5.9.3.3 Mouse Parameter Dialog

The **Mouse** device is used to watch for input from the standard Macintosh mouse. This device can detect button clicks and mouse movement.

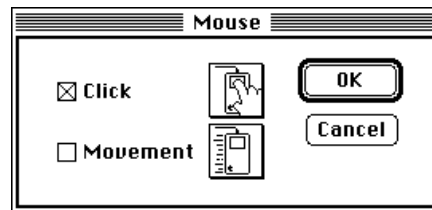


Figure 175 – The Mouse Condition Parameter dialog

Turn on the **Click** checkbox to detect mouse button presses. Turn on the **Movement** checkbox to detect mouse movement.

5.9.3.4 Start/End Parameter Dialog

The **Start** and **End** virtual devices are triggered at the start or end of the posting event or trial (see “5.9 Conditions and Actions”, p192).

Parameters are not usually assigned for conditions using these devices, but you can specify a trial variable expression as a parameter; when a trial variable expression is specified, the condition will trigger only if the expression evaluates to `true`.

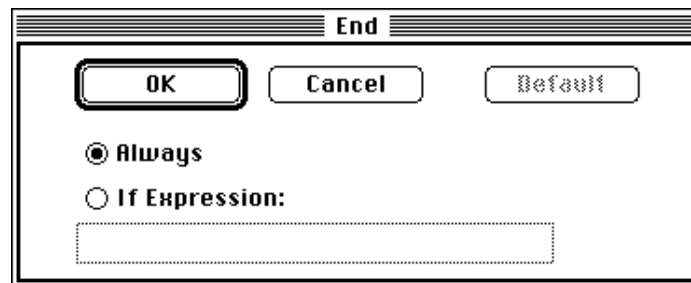


Figure 176 – The Start/End Condition Parameter dialog

Clicking the **Always** radio button in the Start/End Condition Parameter dialog causes the condition to always trigger when the trial/event starts/ends. Clicking **If Expression:** lets you specify a trial variable expression.

5.9.3.5 When Parameter Dialog

The **When** virtual device is used to poll the value of a trial variable expression (see “5.10 Trial Manager Variables”, p205); it triggers when this expression evaluates to `true`.

The When Condition Parameter dialog is a simple text dialog that lets you enter the trial variable expression.

5.9.3.6 ScriptWhen Parameter Dialog

The **ScriptWhen** virtual device is used to poll the value of a PsyScript entry. The condition triggers when the content of the entry has the value `true`. See “, Chapter 12. PsyScript Reference”, p319 for more information on PsyScript.

The Link to Entry dialog is used to set the parameter for a **ScriptWhen** condition. (See “Script Linked”, p155.)

5.9.4 Actions List Dialog

The Actions List dialog is used to describe a list of actions; it is used for setting the value of a field or custom attribute with type **Action List**.

The Actions List dialog works just like the actions box of a condition-action pair in the Conditions and Actions dialog (see “5.9.1 Conditions and Actions Dialog”, p193).

5.9.4.1 Available Actions

This is a list of all of the standard actions available in PsyScope. Your copy of PsyScope may have more actions if an extension has been installed.

AbortEvent []
Parameters: **Event**

This action aborts the specified event, clearing the stimulus, if appropriate. It also:

- deactivates any actions that are linked to the event.
- marks the time the event ended, and computes the event's duration.

AddToList []
Parameters: **LValue**, **Expression**

This action is used to add a value to a trial variable of **List** type. **LValue** should be a variable expression that evaluates to an array variable, and the result of evaluating **Expression** is appended to this array. (See also “5.10 Trial Manager Variables”, p205.)

BBoxOut []Parameters: **Value**, **Mode**

This action sends bbox output codes directly to the box, without the need for a BBox event. The syntax is:

Value an integer value in [0..255] representing the decimal value of the binary lines to be manipulated

Mode (optional: defaults to copy_mode)

"copy_mode" - change the bbox output state to *Value*.

"assert_mode" - turn on the bits that are on in *Value*.

"deassert_mode" - turn off the bits that are on in *Value*.

"xor_mode" - exclusive-or the current state with *Value*.

Beep []Parameters: **Beep**

This action plays a sound stored in an 'snd' resource, with the name **Beep**. If **Beep** is not specified, "correct beep" is used.

The 'snd' resource must be in an open resource file. See "6.1.3 Resources", p216.

CancelAction []Parameters: **Action**, **Stored with**, **Type**

This action removes either a specified action or all actions (controlled by the **Action** parameter) from the action list of either a specified event or all events (controlled by the **Stored with** parameter) triggered by either a particular condition or any condition (controlled by the **Type**).

If an action name is specified in **Action** (e.g. RT), only actions with that name will be removed.

If an event name is given in **Event**, only actions to be posted by that event will be removed.

If a condition device name is given in **Type** (e.g. Start, End, Mouse), only actions that depend on that device will be removed.

There is a technical interaction between `CancelAction` [] and the way in which actions are grouped into action-condition pairs: when any action of a particular condition-action pair is cancelled, all of the actions in the pair are cancelled.

Note: CancelAction[] can only remove actions that have not yet been executed.

ChanceEvent[]

Parameters: **Event**, **Chance**

This action is like `RunEvent[]`, except that the event is run only with the probability given in **Chance**. **Chance** can be a number between 0 and 1, or a trial variable expression that evaluates to a number.

ClearScreen[]

Parameters: *None*

This action erases the screen, using the global background color. (See also “Back-color”, p168.)

ClearPort[]

Parameters: **Event**

This action erases the port that is used by the specified event. **Event** must be of type **Text**, **PICT**, **Document**, **Paragraph**, **Pasteboard**, or **Key Sequence**. The port boarder is unaffected.

ClearStim[]

Parameters: **Event**

This action clears the stimulus associated with the specified event. It also marks the time at which this occurs, computing and storing the actual duration for the event.

It does *not* perform the scheduling operations of `EndEvent[]`; these will be performed when the scheduled end of the event is reached (if the event is currently running).

DrawAllPortBorders[]

Parameters: *None*

This action draws the borders of all stimulus ports that are used by screen events in the experiment.

If different trials use different stimulus ports, then this action can only draw ports that it knows are going to be used. If the experiment is precompiled (see “6.5.1 Pre-compiled”, p246), then all ports will always be known at runtime. Otherwise, the only ports that can be drawn are those used by events which have been compiled so far.

DrawPortBorder[]

Parameters: **Event**

This action draws the port border for the stimulus port of the specified event. **Event** must be of type **Text**, **PICT**, **Document**, **Paragraph**, **Pasteboard**, or **Key Sequence**.

EndEvent []
Parameters: **Event**

This action ends the specified event, just as if its **Duration** terminating condition had been met.

MaskStim []
Parameters: **Stimulus Event**, **Mask**, **Attrib Event**

This action masks the stimulus associated with **Stimulus Event**.

By default, **MaskStim**[] uses the mask specified as the **Mask** attribute for the event; however, an optional a mask stimulus (for **Text** stimuli, this is a character) can be specified as the **Mask** parameter.

Also, an optional **Attrib Event** can be specified as a source for non-**Stimulus** attributes, just as with **ShowEvent** [].

NewListItem []
Parameters: **LValue**

This action extends the size of the trial variable array given in **LValue**; the value of the new item in the list is undefined. (See also “5.10 Trial Manager Variables”, p205.)

NextCrossing []
Parameters: **Factor**

This action causes a new cell to be selected in a factor set — whichever set includes **Factor**. The new cell will be used for the next trial.

This action is intended for use on factor sets with crossing type **Fixed** (see “5.7.2.2 Table Info Dialog”, p140).

QuitBlock []
Parameters: **Block Name**, **Forward Lists**

This action is used to skip any remaining trials in the current block. The current trial continues to execute normally.

If there are multiple levels of blocks in the experiment hierarchy, then **QuitBlock**[] quits within the lowest-level block by default, continuing within that block’s owner (if there are more blocks to execute). To quit a higher-level block, you can specify which block to quit in **Block Name**. Alternatively, you can specify which block to quit as a number; this number specifies how many hierarchical levels of blocks to

quit (thus, the default behavior is equivalent to specifying “1” in **Block Name**).

By default, when trials in the block are skipped, any lists connected to the blocks are left unaccessed for the trials which are not executed. If **Forward Lists** is set to “True”, then for each factor set connected to the block and its owners, a cell is selected for each trial that is not executed; this insures that cells are assigned to trials consistently, whether or not they are run.

QuitTrial []

Parameters: *None*

This action ends the current trial. Any trial actions which execute on the `End` [] condition will be performed.

RemoveFromList []

Parameters: **LValue**, **Expression**

Given an array trial variable in **LValue** and an index into this list in **Expression**, the indexed item is removed from the list. The list is indexed starting with 1. (See also “5.10 Trial Manager Variables”, p205.)

RemovePortBorder []

Parameters: **Event**

This action removes the port border for the stimulus port of the specified event. **Event** must be of type **Text**, **PICT**, **Document**, **Paragraph**, **Pasteboard**, or **Key Sequence**.

RerunTrial []

Parameters: **Trial Number**, **When**, **Arrange**

This action tags the trial numbered **Trial Number** (or the current trial if none is specified) to be run again.

Trial Number is the Trial’s absolute trial number; this specification can be in the form of a number or a trial variable expression (see also “5.10 Trial Manager Variables”, p205).

The **When** parameter specifies when the trial should be re-run; the possible values are **Mix** and **End**. **Mix** specifies that the trial re-run should be mixed in with the remaining first-run trials, while **End** specifies that the re-run should be delayed until all of the first-runs are done. If a trial is re-re-run with **End**, the second-time re-runs will be performed after the first-time re-runs are complete.

The **Arrange** parameter specifies an order within the two **When** types; the possible values are **Start**, **End**, and **Random**. **Start** specifies that the trial should be re-run before any other trials currently scheduled for re-run in its set (**Mix** or **End**). **End** specifies that it should be re-run after the other re-run trials. **Random** specifies that it should be rescheduled at a random position within its re-run set.

ReverseVideo []Parameters: *None*

This function switches the default foreground and background colors for screen stimuli. See also “Backcolor”, p168 and “Forecolor”, p168. If no colors have been set, then screen stimuli will appear white against a black background, as opposed to the Macintosh’s usual black-against-white.

RT []Parameters: **Label**, **Relative to**, **Storage Variable**, **Flag**

This action records the state of all input devices and stores this either: a) in the data file, or b) in a specified trial variable of type **Response**, or c) in both the data file and a variable. Where the information is stored depends on the values of the **Storage Variable** and **Flag** parameters; the default is to the data file.

The **Label** string is stored along with the regular response time information; this label is used by the experiment designer to mark the recorded data and is meaningless to PsyScope. The default label is “”.

The **Relative to** parameter changes which event is used to calculate response time; the recorded response time will be the difference between start time of this event and the time at which a response was received. The default **Relative to** event is the one that posted the action.

The **Storage Variable** parameter specifies a trial variable with either a numerical or **Response** type (see also “5.10 Trial Manager Variables”, p205). If the variable’s type is **Response**, the response data is copied into this variable. If the variable’s type is numerical (**Integer**, **Long Integer**, etc.), the variable is filled in with an index into the standard response list – **RTData** – when the response was recorded.

The **Flag** optional parameter is used when a trial variable of type **Response** is specified for **Storage Variable**; if it is **VAR_ONLY**, the response information is not written to the data file (only to the trial variable).

RunEvent []Parameters: **Event**

This action initiates the specified event, just as if it had been started by the regular scheduling mechanism.

If the event has been run already, but ended, then it will be run again. If the event is still running, then **RunEvent** [] does nothing.

Set []Parameters: **LValue**, **Expression**

Given a trial variable in **LValue**, its value is set to the result of evaluating **Expression**. (See also “5.10 Trial Manager Variables”, p205.)

SetBackColor []

Parameters: **Color**

This function changes the default background color for screen stimuli. See also “Backcolor”, p168.

SetDefaultColor []

Parameters: **Color**

This function changes the default foreground color for screen stimuli. See also “Forecolor”, p168.

ScheduleEvent []

Parameters: **Event, Start**

This action schedules the specified event to be run at the specified starting time, just as if it had been scheduled according to **Start**. **Start** is a string that is the same as the PsyScript expression of a start dependency.

See also “RunEvent[]”, p203.

ScriptEval []

Parameters: **Entry Name**

This action is used to directly evaluate a PsyScript entry. Usually, evaluating the entry will change some value in the script, and that value will be used for compiling future trials.

For more information in PsyScript, see “, Chapter 12. PsyScript Reference”, p319.

ShowStim []

Parameters: **Stimulus Event, Attrib Event**

This action presents the stimulus for **Stimulus Event** without actually executing the event (i.e. its duration condition is not watched, actions are not posted, etc.).

Usually, you will not specify the second event. If you specify two different events, the **Stimulus** value for the first is combined with the non-**Stimulus** attributes from the second.

ShowStim [] does mark the time at which the stimulus began, and records this as the start time for **Stimulus Event**; it does not, however, perform the scheduling operations of RunEvent [] and ScheduleEvent [].

UnscheduleEvent []

Parameters: **Event**

This action removes an event from the run schedule. If the event has already been executed, UnscheduleEvent [] has no effect.

5.9.5 Parameters dialogs

To open a Parameters dialog, double-click on an action in either the Actions dialog or the Action List dialog. The controls in the Parameters dialog work the same as in a Standard Attribute dialog (See “5.8.2 The Standard Attributes Dialog”, p151), with the action parameters as the only available set of “attributes”. The number and type of the parameters in the dialog depends on the action.

Any parameter can be left as **Default**; in this case, a built-in default will be used, since there is no notion of inheritance for action parameters.

5.10 Trial Manager Variables

Trial Manager variables (or simply *trial variables*) are named run-time variables that can be used to vary an event based on user inputs and/or previous trials in the experiment.

The graphic environment is rather limited in its handling of trial variables, so that anyone who needs to use variables in a serious capacity may need to learn a small amount of scripting. It should be noted, however, that the PsyScript and trial variable “universes” are *separate*: trial variables are described in the syntax of PsyScript, but they are manipulated by processes outside of the PsyScript interpreter.

Trial variables are created by using the Trial Manager Variables dialog (see “5.10.5 Trial Manager Variables Dialog”, p208). Each variable has a type, which can be:

Character – Stores a single text character.

Integer – Stores an integer number between -2^{15} and $2^{15}-1$.

Long Integer – Stores an integer number between -2^{31} and $2^{31}-1$

Real Number – Stores a “real” (floating-point) number.

Point – Stores a pair of integers.

Response – Stores a response record (from the RT[] action).

There is also are also array trial variable types, which cannot be used with the graphic environment.

Trial variables values are modified through actions (see “5.9 Conditions and Actions”, p192). In general, values can be modified with the `AddToList[]`, `NewListItem[]`, `RemoveFromList[]`, and `Set[]` actions are used to modify trial variable values. Of these, only `Set[]` can be used with variables defined in the graphic environment. (The other actions work on variables with array types.)

The `RT[]` action also works with variables. A variable of type **Response** may be passed to the `RT[]` action, in which case the input data will be written to the variable as well as the data file. The data file output is actually stored in a built-in variable called **RTData**; this trial variable is a list of **Response** records. See also “`RT[]`”, p203 and “5.10.3 Built-in Variables”, p207.

The value of a trial variable is used by either:

- including the variable in a condition or action parameter *trial variable expression*, or
- linking an attribute value to the variable.

The former usage is the most common. The **Start**, **When**, and **End** conditions use trial variable expressions — specifically, expressions which have a “true” or “false” value — to decide when to trigger. Various actions — such as `ChanceEvent[]` — accept trial variable expressions as a parameter values. Trial variable expressions are discussed in “5.10.2 Trial Manager Variable Expressions”, p206.

The latter usage — linking attributes to variables — is discussed in “5.10.4 Linking to Variable Values”, p207.

5.10.1 How Trial Manager Variables Work

When the experiment is started, PsyScope reads all of the trial variables that are defined. If the **Initialize** flag is on for a variable (see “5.10.5 Trial Manager Variables Dialog”, p208), it is set to the given initial value; otherwise, the initial value of the variable may be the value that it had at the end of the previous run of the experiment.

During each trial of the experiment, built-in variables are updated and user-defined variables may change due to the execution of `Set[]` and `RT[]` actions. Variable values can change many times within a trial, and the values are retained across trials.

The graphic environment’s Trial Manager Variable dialog outputs trial definitions in PsyScript form. Once a variable has been read in for the execution of an experiment, this PsyScript definition will be ignored, unless the **Update in Script** flag is set for the variable (see “5.10.5 Trial Manager Variables Dialog”, p208). In this case, the trial variable’s current value will be written back out to the script after each trial. See also “5.10.4 Linking to Variable Values”, p207.

5.10.2 Trial Manager Variable Expressions

Trial variable expressions are used to retrieve and compute values from trial manager variables. These expressions are evaluated a run time, so that the dynamic nature of trial variables is captured in the evaluation.

Although the use of trial variable expressions is potentially independent from the use of PsyScript, it is considered an advanced feature, documented in “Part 4: Scripting Reference, 13.5 Trial Manager Variables”, p412.

5.10.3 Built-in Variables

There are three built-in Trial Manager Variables:

RTData — an array of response data collected during this execution of the experiment.

TrialNum — the number for the currently executing trial.

RunNumTrials — the number of trials to be executed in this run of the experiment.

These variables can only be used with trial variable expressions. See “5.10.2 Trial Manager Variable Expressions”, p206, above.

5.10.4 Linking to Variable Values

If you want to link some attribute in the experiment to the value of a particular variable, you can do it this way:

- 1) Make sure that the **Update in script** flag is on for the variable (see “5.10.5 Trial Manager Variables Dialog”, p208).
- 2) Use the **Script Linked** attribute setting, and choose the variable from the Link to Entry dialog (see “Script Linked”, p155).

The **Update in Script** bridges the gap between the variable and PsyScript worlds — somewhat. It is important to remember that trial variables are free to change while a trial is being executed; however, a trial’s complete description is compiled before any part of the trial is executed. This means that linking to a variable (in the above manner) lets you use only the value that the variable has *before the trial is executed*. On the other hand, variable references within trial variable expressions (as condition or action parameters) have access to the dynamic behavior of the variable within the trial execution.

This subtlety of linking to trial variables is closely related to the subtleties of precompiling; see also “6.5.1 Precompiling”, p246.

5.10.5 Trial Manager Variables Dialog

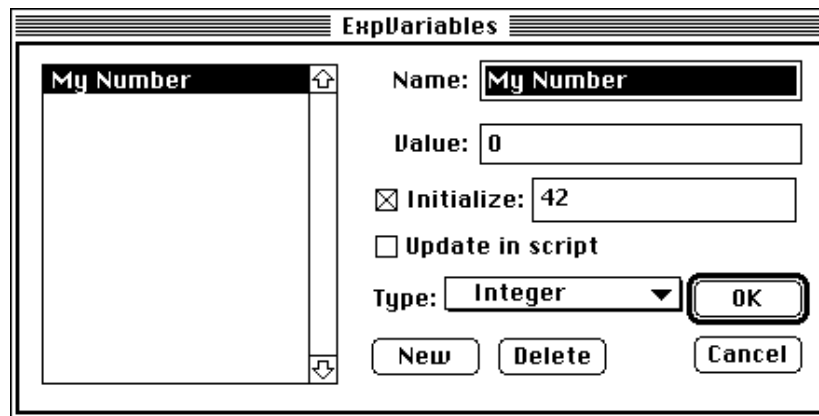


Figure 177 – The Trial Manager Variables dialog

The Trial Manager Variables Dialog allows you to create, delete, and edit trial manager variables (see “5.10 Trial Manager Variables”, p205). To open it, click the **V**ariables icon at the top of the View window.

Clicking the **N**ew button creates a new trial variable. (By default, the new variable will have the same type as the last selected variable, but this can be changed with the **T**ype pop-up menu; see below).

When one or more trial variables are selected in the list, hitting the **D**ellete button removes the variables from the script.

When one trial variable is selected in the list, the name of the variable can be edited in the **N**ame text box at the top right of the dialog. The variable’s current value can be set in the **V**alue text box.

When the **I**nitialize checkbox is on, the selected trial variable’s value will always be reset when the experiment starts; the initial value is set in the text box to the right of the checkbox.

If **U**ppdate in script is checked, PsyScope will update the value of the variable in the script after each trial when the experiment is run. This lets you use trial variable values for scripting, as long as the experiment is not precompiled (see “6.5.1 Precompiling”, p246).

The **T**ype pop-up menu determines the type of the currently selected trial variable. See “5.10 Trial Manager Variables”, p205 for a discussion of the available variable types.

Hitting **O**K closes the dialog. Hitting **C**ancel also closes the dialog, but changes made to variables are discarded, including the creation or deletion of variables.

5.11 Trial Chooser Floating Window

The Trial Chooser floating window is used with the Block dialog, Template window, and some Attributes dialogs. It is used to select a path in the hierarchy — from the experiment object to the dialog’s object — so you can preview trials and attribute values.

For example, suppose a template is owned by two blocks, and some event in the template uses **Vary by Block**; the event will be different depending on which block is used to run the trial. You can preview the trial by hitting the **Preview** button in the Template window, but you need to be able to control which block is used for previewing. To set this block *context*, you use the Trial Chooser.

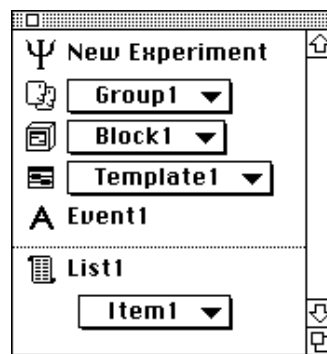


Figure 178 – The Trial Chooser

A context is selected in the Trial Chooser by using pop-up menus: there is one pop-up menu for each level of the hierarchy. In the above example of choosing a block, there would be a block pop-up menu (with a block icon next to the menu) that would let you select one block or the other.

Depending on the structure of your experiment, changing an object in one pop-up menu can affect the allowed objects in other pop-ups. It may even change the number of object pop-up menus, if different hierarchical paths are of different lengths.

If lists are connected to the block or template, the Trial Chooser lets you select a current item for the list; lists are shown in a separate area at the bottom of the Trial Chooser. When previewing a template, the Trial Chooser sets the list item that will be used; when previewing a block, the Trial Chooser sets the list item used for the first trial, and then item-selection proceeds normally for the rest of the trials.

Besides previewing, the Trial Chooser is used to select a context for Attribute dialogs. When the Trial Chooser is open, attributes which are **Vary by...** will display the value they take in the given context (as opposed to the **Vary by...** parameterization, which is shown otherwise).

When factor tables are used, the currently selected cell in the factor table is used for previewing a trial.

Shortcut: Clicking on an object icon in the Trial Chooser (to the left of the object pop-up menu) opens the dialog for the object that is currently selected in that pop-up menu.

5.12 Additional Concepts

5.12.1 List Ordering

When a trial is being generated for the experiment, there are many places in the hierarchy where something needs to be chosen from a list of items; e.g. from a list of blocks, templates, or levels. The manner in which a choice is made depends on the list's *access type*.

The three possible list access types are:

- **Sequential** - starts with the first item, and always chooses the next item in the list. If an item is given a weight > 1 (see below), the same item is used as many times as the weight specifies.
- **Random** - chooses an item from the list randomly. An *access history* is kept, so that after an item (with weight = 1) has been selected, it is ineligible for re-selection until all items have been selected at least once. If an item is given weight > 1 (see below), it can be used as many times as the weight specifies before it is ineligible for re-selection. When no items are left to be selected in the list, the access history is erased and selection starts again.
- **Random with Replacement** - chooses an item from the list randomly. Unlike **Random**, each item selection is independent of previous selections, so there is no guarantee that all items will be used before an item is used multiple times. Item weights (see below) affect only the *probability* that an item will be selected.

5.12.1.1 Weights

The access type of a list can be modified by assigning weights to items in the list. A weight effectively multiplies the number of occurrences of the item in the list.

5.12.1.2 Other Modifiers

The list selection mechanism can be further modified through parameters available only through PsyScript. See “Part 4: Scripting Reference, 12.8 Lists”, p329 for more information on these parameters.

5.12.2 Trial Counting

5.12.2.1 Experiments without Blocks

In an experiment without blocks, the number of trials to be run is either fixed or is based on a total running time for the experiment (i.e. a “block” duration).

If there are no groups in the experiment, you can set the trial count through the Experiment dialog. If there are groups in the experiment, you must set the trial count for each group separately, using the Group dialog.

5.12.2.2 Experiment with Blocks

In an experiment which contains blocks, the number of trials to be run depends on:

- The number of blocks in the experiment/group.
- The number of trials to be run within each block.
- The number of times the entire list of blocks is to be executed.
- The block scaling number of the experiment/group.

At the block level, the number of trials to be run *within* the block is set in much the same way as the total trial count for an experiment without blocks (see above). The only additional feature — the one that makes trial counting seem complicated — is related to the block scaling value, discussed below.

Given the number of trials to be run within each block, the number of trials that are run in one execution of all the blocks of the experiment/group is the sum of the block counts. The experiment/group contains a *cycles* count, which determines how many times the whole block list will be executed. Thus, the total number of trials to be run is the cycle count times the sum of the block counts.

5.12.2.3 Block Scaling

An experiment or group that owns blocks has a *block scaling value*, which is used to scale the number of trials to be run within each “scalable” block. The scaling value affects the trial count of the immediately owned blocks only. Any sub-blocks or factors that are owned by the blocks are unaffected.

In defining the number of trials to be run within a block, the trial count can be *fixed* or *scalable*. Fixed-count blocks are not affected by the block scaling number, while scalable blocks are.

The fixed/scalable nature of a block is set in the Block dialog. Blocks whose trial counts depend on a block duration are never scaled.

5.12.2.4 Superblocks

When counting trials, superblocks work the same as groups or experiments which own blocks. The only difference is that the superblock's cycle count can be fixed or scalable, just like a block's trial count.

5.12.2.5 Trial Counts and Crossing Factors

It is currently still the responsibility of the experiment designer to set up trial counts so that the right number of cells will be used from factor tables and lists. Future versions of PsyScope will support a more automatic count-setting facility.

5.12.2.6 Trial Counts Reported in the Trial Monitor

The Trial Monitor (see “6.3 The Trial Monitor”, p238) displays the number of trials that PsyScope expects to execute for a single run of the experiment. In a design without duration-based blocks (and which does not use `RerunTrial[]`), this trial count is correct.

If a design contains blocks whose trial counts depend on a block running time, the block is assigned only one trial for the purposes of predicting the number of trials to run. In this case, the actual number of trials executed when the experiment is run will be more than the predicted number.



Chapter 6. Running and Managing Experiments

6.1 File System

6.1.1 Using Projects

A *project* is a PsyScope file that keeps a list of scripts. The experiments in these scripts are all put together in one pop-up menu in the Console so that you can switch from any experiment in any of the scripts to any other.

The main use of projects is to group experiments that you plan to run together. However, projects also provide some other features:

- Most options are kept separately for each project.
- An option may be enabled to specify all paths relative to the project (see “6.1.2 Path Names”, p215); this allows you to create a complex file structure — beneath the folder containing the project — which will remain intact when the whole folder is moved.

6.1.1.1 Creating a Project

A project is created by choosing **New Project...** from the **File** menu. Scripts are usually added to the project through the Scripts dialog (see below). Also, when a script is created with **New Script...** in the **Design** menu while a project is open, a Message dialog appears allowing you to automatically add the new script to the open project.

6.1.1.2 The Scripts Dialog

The Scripts dialog is used with projects to add and remove scripts to the project.

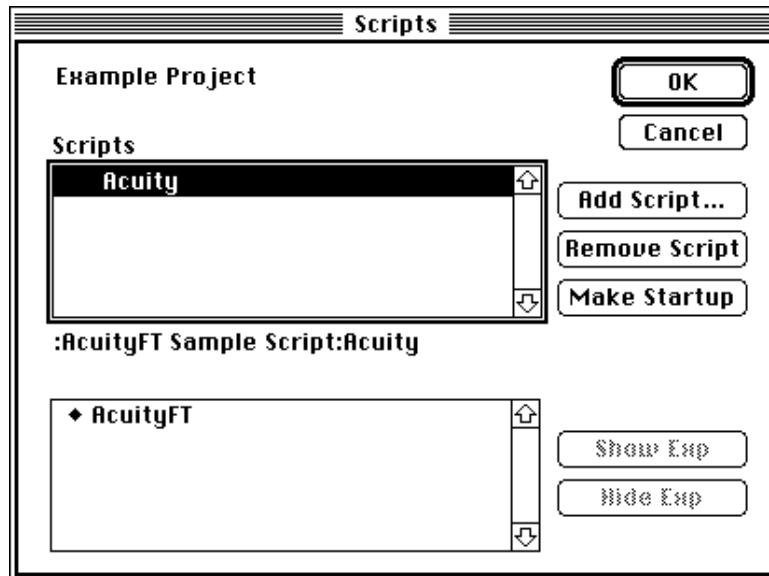


Figure 179 – The Scripts dialog

At the top of the dialog is a list of scripts included in the project. Clicking on one of these displays the (full or relative) pathname of the script and fills the lower list with the names of the experiments in that script. (See also “6.1.2 Path Names”, p215.)

A check mark is placed next to the *current script* — i.e. the script containing the experiment which is currently loaded for the project; if the Scripts dialog is opened just as the project is being loaded (by holding the Shift key down; see “6.1.7 Start-up Shortcuts”, p224), the current script can be changed by clicking to the left of a script name.

A bullet (●) placed beside the name of a script means that it is the *start-up script*; it will be made the current script whenever the project is first opened.

In the experiment list, a filled-in diamond next to an experiment name means that the experiment is visible, i.e. it can be selected from the **Switch Experiment** submenu or from the Console pop-up menu. The visibility of an experiment can be set with the **Show Exp** and **Hide Exp** buttons, or by clicking on the diamond to toggle it.

You can add new scripts to the project with the **Add Script** button; a sequence of standard file dialogs will get a list of scripts to add. Hit **Cancel** in the file-finding dialog when you are done adding scripts.

Scripts can be removed with the **Remove Script** button. If the currently loaded script is removed, the script will be automatically changed as soon as the dialog is closed.

Hitting the **OK** button closes the dialog and saves changes to the project; **Cancel** returns the project settings to their former state.

6.1.2 Path Names

With the Macintosh's Hierarchical File System (HFS), files used within PsyScope may have a complicated location relative to the application or open script or project. To allow access to any file on the volume, *path names* are used.

A path name includes the name of the file and the folder in which the file is located; this folder may be in another folder, and so on. PsyScope uses the file path notation that is standard to the Macintosh: the volume name is given, followed by a folder in the volume's root, and so on, until the file name is finally reached. The parts of the path are separated by colons. For example, a file "Psy.pict" that is in the folder "Picts" in the top level of the volume "My Disk" would have the full path name "My Disk:Picts:Psy.pict".

To make things simpler, PsyScope recognizes *relative paths* (see below), so that file names can be specified in relation to the current script (or project). Also, because path names can be very long and therefore difficult to display, PsyScope supports a *reverse notation* for a pathname (see below).

6.1.2.1 Relative Paths

In the simplest case, files that are used within a script are placed in the same folder as the script. Then, the file names can be used without further specification of the location.

A slightly more complex case is when files used by the script are in a folder that is within the script's folder. In this case, the full path (starting from the volume) still does not need to be specified; *relative paths* can be used instead.

If a pathname begins with a colon (":"), then it is assumed that the path starts at the script's directory. For example, ":Picts:Psy.pict" is the relative path of the file "Psy.pict" that is in the folder "Picts" which is in the same folder as the script.

Relative paths can be arbitrarily complex — so that folders may be nested within folders, etc. — as long as the whole folder structure is under the script's folder.

Relative Paths with Projects

When a project is used, there is an option in the General options dialog — **Store path names relative to...** — which allows you to change the starting point of relative paths. This option affects all path names in the script. See "7.6.1 General Options", p266.

6.1.2.2 Reverse Notation

Path names are usually specified in the order *folder-path:file*. *Reverse notation* allows you to change this to *file @ folder-path*. There must be exactly one space on each side of the

“@”, and only the file name can appear before the “@”. No extra spaces may be added anywhere in the name.

For example, the file path “My Disk:Picts:Psy.pict” can also be specified as “Psy.pict @ MyDisk:Picts:”.

6.1.3 Resources

In many experiments, you may want to import custom resources — ‘PICT’ or ‘snd’, for example — to be used in running the experiment. You can make these resources available in three different ways:

- Place the resource file in a “PsyScope Extensions” folder that is in the same folder as the PsyScope application (see below). The resources will be available whenever PsyScope is running.
- Put the name(s) of the resource file(s) in a “Resources” entry in your script (see “Part 4: Scripting Reference, 16.6.3 Resources”, p461). The resources will be available whenever the script is open.
- Put the name(s) of the resource file(s) in a “Resources” experiment attribute (see “Resources”, p167). The resources will be available only while the experiment is running.

Resources are accessed within PsyScope by name, never by ID. This means that resources of the same type in different resource files do not have to have unique IDs, only unique names.

6.1.3.1 PsyScope Extensions

PsyScope Extensions are Macintosh code resources that extend the functionality of PsyScope. There are several classes of Extensions, including output device drivers (ODEVs), input device drivers (IDEVs), timers (TIMRs), and dialogs/functions (DCODs).

A PsyScope Extension will typically be bundled in a file with all of the resources it needs. To install the Extension, you must have a folder named “PsyScope Extensions” that is in the same folder as the PsyScope application. The Extension file is simply placed in this folder, and it will be available the next time PsyScope is restarted.

The “PsyScope Extensions” folder can be used to make any kind of resource (e.g. ‘PICT’) available for use within PsyScope. Generally, the resource must be given a name (a unique ID does not suffice; see above).

If a text file is placed in the “PsyScope Extensions” folder, it will be interpreted as a script file to be automatically `#included` at the end of any script loaded into PsyScope. Any ‘TEXT’ resource with a name that starts with “Script.” (in any of the open resource files) is treated the same way. (The ‘TEXT’ resources must also have unique names.)

When PsyScope starts up, a folder icon with a psy on it is drawn at the left end of the menu bar; this indicates that a “PsyScope Extensions” folder was found, and icons representing Extensions will be displayed after the folder (similar to start-up icons for Macintosh INITs). If the folder icon is X-ed out, no “PsyScope Extensions” folder was found.



Figure 180 – PsyScope Extensions Folder Found Icon



Figure 181 – PsyScope Extensions Folder Not Found Icon

6.1.4 The Data File

When you run an experiment, the data that is recorded by any RT[] actions in the script is recorded in a *data file*. PsyScope appends the new data and experiment information at the end of the data file every time the experiment is run. The data file is a plain text file that can be read by any text editor, spreadsheet, or statistics program.

6.1.4.1 Specifying the Data File

The name and location of the data file for an experiment is specified using the **Data File** experiment attribute (see “Data File”, p161), or by using the Data File Dialog (see “Part 2: Graphic Environment Reference, 6.2.5 Data File Dialog”).

If you do not specify a data file in your script, the program will ask you to specify one each time you run the script.

6.1.4.2 Information in the Data File

The information in the data file is divided into three categories: *header information*, *response data*, and *timing statistics*.

The Data File Header

The *data file header* is written at the beginning of a run and contains general information about the experiment run, information about the machine that the experiment was run on, and optional information defined by the experimenter (especially subject information).

The data header is an optional feature of the data file; it is written by default, but can be omitted by checking the **Don’t include header information** box in the Data Info dialog (see “Data Info”, p161).

The default data header information looks something like this:

```
PsyScope 1.0 started: 3/30/93 14:30:04
Script file: My Hello World Script
Run on: Macintosh IIci
Input devices active: Key Mouse
```

It contains five default data header items:

- PsyScope application name
- Start date and time
- Script file name
- Macintosh type
- Active input devices

Subject information can be added to this header through the Subject Info dialog (see “6.2.1.2 Subject Info Dialog”, p226). Scripters can add additional fields by using the “DataHeader” experiment attribute (see “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360).

Response Data

Response data follows the header information in the data file. This is data from the experiment run from the execution of RT[] actions.

The response data is written out in the form of records, where each record represents one execution of the RT[] action. Each record is comprised of fields of data, and the first record contains the name of each field.

By default, each new line represents a new record, and each field is separated from the previous field within the record by a tab. This format is compatible with most spreadsheet and data analysis programs, but it can be changed if necessary; see “6.1.4.3 Formatting the Data File”, p222.

There are four different kinds of data fields that can be in each record:

- Default data fields (always written).
- Optional standard data fields. These can be added via the **Data Info** experiment attribute; see “Data Info”, p161.
- Input device data fields. These contain data from whatever input devices are active for the experiment run. Some fields are optional, and are added via the **Data Info** experiment attribute; see “Data Info”, p161.
- Data variable fields. These contain the values of trial variables that are added via the **Data Variables** experiment attribute; see “Data Variables”, p163. See also “5.10 Trial Manager Variables”, p205.

Default Data Fields

There are three default data fields. These fields are always written in the response data.

Trial Number – The number of the trial in which the data were recorded.

Condition – The condition of the trial in which the data were recorded. This is the value set automatically by the crossing of factors, or through the **Condition Name** trial attribute (see “Condition Name”, p175).

Time – The time at which the data were recorded. This value is the amount of time in milliseconds since the beginning of the event that the RT[] action was relative to (see “RT[]”, p203).

Optional Data Fields

These fields are turned on through the Data Info dialog; see “Data Info”, p161.

Run Label – This field contains the label specified in the “RunLabel” experiment attribute, which can be set only through PsyScript. It will be the same for every line. This field is turned on by the **Label to be included on each data line** checkbox in the Data Info dialog.

Stimulus – This field contains the stimulus of the event that posted the RT[] action. This field is turned on by the **Stimulus of relative event** checkbox in the Data Info dialog.

Event Tag – This field contains the values of the **Tag** attribute for the event that posted the RT[] action, providing a way of writing further information about the event in the data file. This field is turned on by the **Event tag** checkbox in the Data Info dialog.

Put Up By – This is the name of the event that posted the RT[] action. This field is turned on by the **Event that posted RT action** checkbox in the Data Info dialog.

Removed By – This is the event that was scheduled to be active until. This field is turned on by the **Event that removed RT action** checkbox in the Data Info dialog.

During – This is the event during which the action was executed. If more than one event was running, the most recently started is used. This field is turned on by the **Event during which response occurred** checkbox in the Data Info dialog.

Relative To – The event that the time is measured relative to. By default this is the event that posted the action, otherwise it’s the event specified as the **Relative To** parameter of the RT[] action. This field is turned on by the **Event that RT is relative to** checkbox in the Data Info dialog.

Onset – This field contains the onset time, relative to the start of the trial, for the event listed in the field immediately to its left. If more than one event field (e.g. During and Relative To) exists in the response data, then there will be one onset field for each. This field is turned on by the **Onset times for referenced events** checkbox in the Data Info dialog.

Response Label – This field contains the label specified in the first parameter of the RT[] action. This field is turned on by the **Response label** checkbox in the Data Info dialog.

Input Device Data Fields

Each input device that is active has one or more data fields which it writes in the data file. These fields are written in the data file in alphabetical order, as listed in the Input Devices dialog, used by the **Input Devices** experiment attribute (see “Input Devices”, p164).

Some input devices have optional fields which are only included if their corresponding items are checked in the Data Items dialog (see “Data Info”, p161).

Below are the data fields for the three built-in input devices: **BBox**, **Key**, and **Mouse**. Other input devices added as PsyScope Extensions will have their own fields; consult the extension’s documentation for more information.

BBox Data Fields

state – The current input state of the button box. The input state is represented numerically, as the sum of the line value for each bbox input line. The line values are computed as follows: for a line x , the line value is 2^x . For a released line x , the line value is 2^{x+8} . This gives the following line values:

Table 1: Button Box State Values

Line	Button	Press	Release
1	1	1	512
2	2	2	1024
3	3	4	2048
4	n/a	16	4096
5	n/a	32	8192
6	voice key	64	16384
7	n/a	128	32768
8	n/a	256	65536

Key Data Fields

key – The key combination that was pressed at the time of the RT[]. A key combination is represented as the letter representing the key pressed, optionally preceded by one or more modifiers. Possible modifiers are **CMD**, **OPT**, **CTL**, and **SHIFT**, representing the Command, Option, Control, and Shift keys, respectively. If the key that was pressed is non-printable,

then a value of the form ASCII-*n* or CODE-*n* will be printed, where *n* is an ASCII or keyboard code value, respectively.

sequence – (optional; turned on by the **Key sequence** checkbox in the Data Info dialog) The key sequence stored at the time of the RT[]. If a **KeySequence** event is currently running, this field will contain every key pressed since the start of the event. If more than one is running, it will contain every key pressed since the start of the oldest **KeySequence** event. If none are running, it will be empty. See “Part 4: Scripting Reference, 14.2.4 KeySequence”, p432 for more information on the **KeySequence** event type.

Mouse Data Fields

mouse_down – Whether or not the mouse was pressed at the time of the RT[]. The field will contain 1 if it was pressed, 0 if not.

x – (optional; turned on by the **Mouse position** checkbox in the Data Info dialog) The horizontal position of the mouse at the time of the RT[]. This is specified in global screen coordinates, with the origin at the upper left corner of the main monitor (the one with the menu bar).

y – (optional; turned on by the **Mouse position** checkbox in the Data Info dialog) The vertical position of the mouse at the time of the RT[]. This is specified in global screen coordinates, with the origin at the upper left corner of the main monitor (the one with the menu bar).

Data Variables

In experiments using Trial Manager Variables, it is sometimes necessary to record the values of some variables in the data file. The **Data Variables** experiment attribute provides this ability.

Any variable whose name is listed in this attribute will be written in a field in each line of the data file. The value written will be the value of the variable at the time of the RT[]. If more than one variable is listed, they will be written in the data file in the order in which they are listed in the **Data Variables** attribute.

Timing Statistics

Timing statistics can be written to the data file at the end of the response data. Timing statistics are optional data, and must be turned on by the **Summary timing statistics for conditions** or **Full timing statistics for conditions** checkbox in the Data Info dialog (see “Data Info”, p161).

Timing statistics are meant to be used as an experimental design debugging tool. The statistics provide information on the actual onset time and duration of the events in the various conditions for the experimenter to use to confirm that the trials are indeed running as expected.

Note: Both timing statistics options assume that the event structure of each trial in a condition is the same. If this is not the case, the statistics presented will not accurately represent the timing of the experiment.

Summary Timing Statistics

Summary timing statistics present, for each condition in the set of trials run, the average onset time and duration of each event in that condition, along with the number of trials averaged.

Full Timing Statistics

Full timing statistics present, for each condition in the set of trials run, the actual onset time and duration of each event in every trial in that condition.

Note: Full timing statistics list a record for every event executed during the run, and as a result can get rather lengthy — longer, in fact, than the response data.

6.1.4.3 Formatting the Data File

By default the data in the data file are presented with one record per line, and a tab character separating the records. This format is compatible with most data analysis and spreadsheet programs.

Nevertheless, two experiment attributes — “DataFieldDelimiter” and “DataRecordDelimiter” — have been provided for the purpose of changing the separators between fields and records. These attributes are available only through PsyScript; see “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360 for more information.

6.1.5 The Log File

As your scripts are run, PsyScope automatically maintains and updates a record of all the major events and keeps this record in a *log file*. The log file is an ordinary text file that has a #LogFile identifier at the top of the file.

6.1.5.1 Specifying the Log File

The current log file can be specified in the script; if it is not, the default log file is named “PsyScope.log” and is put in the same directory as the PsyScope application.

The current log file can be changed by selecting **Change Log File...** in the **File** menu. In System 7.0, double-clicking on an existing log file in the Finder will also change the log

file. The script will be updated to refer to the new file. (The script must be saved to retain this information.)

6.1.5.2 Viewing and Editing the Log File

The log file can be viewed by selecting **View Log File** in the **File** menu. The log file is put into a read-only window; it does not automatically update if the log file is changed. If the log file is edited in an application other than PsyScope, be sure that it is saved in text format.

6.1.5.3 Information in the Log File

When PsyScope starts up, a line is added to the current log file that records the version of PsyScope as well as the date and time at which it was run. This line looks something like this:

```
#-----
Application:: "PsyScope 1.0" "6/23/93" "2:28:58"
#-----
```

Whenever a new experiment is loaded, a line is added that records the name of the Experiment entry and the time at which it was loaded:

```
ExperimentLoaded: "Acuity Experiment Script" "8:16:44"
```

Whenever an experiment is run, the name of the Experiment entry is recorded again along with the name of the data file:

```
ExperimentRun: "Acuity Eperiment"
DataFile: "My Disk:PsyScope:Sample Script:Acuity Data"
```

If the experiment completes normally, then the number of trials that were run is recorded:

```
NumTrialsRun: 100
```

If an error occurs, then the error message as well as the user's response is recorded:

```
Message:
  {Call to RunEvent requires an event reference as its first
parameter (trial: 1; event: "Recurrent Entry")}
  {*** User Canceled ***}
```

The user can record information in the log file directly, by selecting the **Log Comment** item from the **Utilities** menu. Comments appear in the log as follows:

```
Comment:
  {This is a comment.
  Notice that comments can occupy multiple lines.}
```

The Subject Info system uses the log file to generate subject and run numbers; see “6.2.3 Subject Number Calculation”, p230. Other information can be added to the log file using PsyScript. See also “Part 4: Scripting Reference, 16.5 Log File”, p458.

6.1.6 Safe Saves

When PsyScope saves a script, instead of writing directly over the old file, it uses a special mechanism to insure against losing data to disk errors. To do this, it changes the name of the old file, writes a new file, and then deletes the old file. This way, if there is a disk error or your machine crashes while PsyScope is saving the script, the original file is not destroyed.

The old file's name is changed by adding “.ssbk” to its name; if the file name is already 32 characters long (the maximum length of a name on the Macintosh), a safe-save cannot be performed.

If your machine does crash while PsyScope is saving, you will probably find two files after rebooting. The file with a name ending in “.ssbk” is the original, pre-save file. The other file (which probably does not have the psy-file icon), was the attempt to save the script; this file may or may not be valid.

6.1.7 Start-up Shortcuts

If you hold down the Option key while PsyScope is starting up, then the autoload file option will be ignored and no file will be auto-loaded (see “7.6.1 General Options”, p266). A “no file” icon will be displayed at the end of the Extensions icon list in the menu bar.

When loading a project, hold down the Shift key to get the Script dialog for the project (see “6.1.1.2 The Scripts Dialog”, p214). You can add or remove experiments before the project is opened. Also, when loading a project, hold down the Control key to get the General Options dialog for the project (see “7.6.1 General Options”, p266).

6.2 Subject Info

PsyScope's Subject Info system helps you keep track of your experiment subjects and change parameters of the experiment based on the current subject.

The basic elements of Subject Info system are:

- Subject Info items – These define the information that you will keep about each subject. They can be simply stored in the log file, or they might be used to select a group for the subject. See “6.2.1 Subject Info Items”, p225.
- Log file – Most subject information is stored here across runs of the experiment. Subject information can also be stored in the data file, but information relevant to grouping must be stored in the log file. See “6.1.5 The Log File”, p222.

- Subject number calculation – This is a scheduled process which reads the log file and the current subject information to generate a subject number (and some other indices) for the current subject. See “6.2.3 Subject Number Calculation”, p230.
- Automatic grouping – This is another scheduled process which is almost always performed along with Subject number calculation. (The processes can be separated through PsyScript, but the graphic environment forces them to be the same.) Automatic grouping selects a group in the experiment hierarchy to be used for the current subject. See “6.2.4 Automatic Grouping”, p233.

6.2.1 Subject Info Items

Subject Info items are created and maintained through the Subject Info Item dialog (see “6.2.1.2 Subject Info Dialog”, p226). A Subject Info item can be any piece of information about the subject that is relevant to your experiment — e.g. the subject’s name, age, or sex — but items generally fall into one of two classes:

- Items which are used to determine which group is used in executing the experiment
- Items which are not used directly in executing the experiment, but are recorded in the log file or data file for later analysis

Items of the first type will be involved with automatic grouping and possibly subject number calculation. Items of the second type are not involved with automatic grouping, but must be explicitly configured to be logged or written to the data file.

In the simplest case, a subject info item simply exists, and you can change its value through the Subject Info dialog. Usually, however, you will schedule a point in your experiment session (through the Subject Info Schedule dialog) to prompt the experimenter for the item’s value. The type and definition of an info item — as set in the Subject Info dialog — determines the kind of dialog that will be given for a prompt.

Note: The Subject Info Schedule dialog also controls when a Subject Info item is written to the log file or data file. Be sure to schedule the prompt before the item is logged, recorded in the data file, used for auto-grouping, or used in a subject number calculation. See also “Part 2: Graphic Environment Reference, 6.2.3.1 Logging and Scheduling Correctly”, p231 and “Part 2: Graphic Environment Reference, 6.2.6 Subject Info Schedule Dialog”, p237.

6.2.1.1 Special Items

There are seven special Subject Info items: SubjectName, SubjectNumber, RunNumber, GroupNumber, SubjectCount, RunCount, and GroupRunCount. The first three are created by default in any new experiment. The last three can be created in the Subject Info dialog

just like any other item; PsyScope will automatically recognize items with these names as special items.

The SubjectName item is special because it is required by subject number calculation and automatic grouping (see “6.2.3 Subject Number Calculation”, p230 and “6.2.4 Automatic Grouping”, p233).

The rest are special because they correspond to values that are calculated by the subject number calculation process; whenever a subject number calculation is performed, the values of these items are changed to match the computed values.

6.2.1.2 Subject Info Dialog

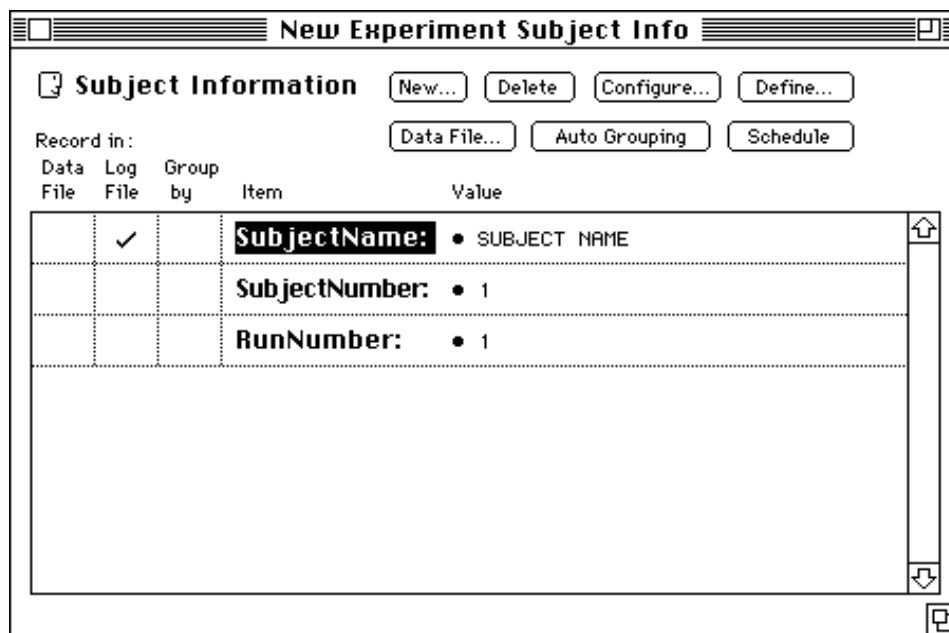


Figure 182 – The Subject Info dialog

Subject Info dialog is the central dialog for setting up subject information to be tracked in PsyScope. The Subject Info dialog is opened by clicking on the **Subject Info** icon in the Design window, or by selecting **Subject Info** from the **Windows** menu.

The main part of the dialog contains a list of items which hold information about the current subject; “SubjectName”, “SubjectNumber” and “RunNumber” are the default items. New subject info items can be created by clicking on the **New...** button, or by selecting **New Info Item...** from the Edit menu.

When you create a new info item, you must specify the type of information that the item contains. This is done through the New Info Item dialog, which is automatically opened when you create a new item (see “New/Reconfigure Info Item Dialog”, p228).

For some item types, you will need to further define the item. When you click **OK** in the New Info Item dialog, a Define Info Item dialog will open (see “Define Subject Info Item Dialog”, p229).

To reopen the New Info Item dialog or Define Info Item dialog, select the item and click on the **Configure...** or **Define...** button in the Subject Info dialog.

In the main list of the Subject Info dialog, each Subject Info item is shown with its current value to the right. Just as in the standard Attributes dialog, the value of an item can be changed by clicking on it.

To the left of each Subject Info item is a row of checkboxes. The checkboxes control whether the item is recorded in the data file and log file, and whether it is used as a grouping criterion.

The **Record in Log File** column is checked if the item has a **Log when** configuration is set to anything other **Never** (through either the New/Reconfigure Info Item dialog or Subject Info Schedule dialog; see below and “Part 2: Graphic Environment Reference, 6.2.6 Subject Info Schedule Dialog”, p237). If the column is initially unchecked and is checked by the user in the Subject Info dialog, the item will be logged just before running the experiment.

If the **Record in Data File** column is checked, the Subject Info item and its value will be stored in the header of the data file.

The **Group By** column is checked when the item is used as a grouping criterion (see “6.2.4 Automatic Grouping”, p233 and “6.2.4.1 Automatic Grouping Dialog”, p234).

The **Data File...** button opens the Data File dialog, which is used to configure automatic data file name generation (see “6.2.5 Data File Dialog”, p236).

The **Auto Grouping** button opens the Automatic Grouping dialog, which is used to configure automatic subject assignment to a group (see “6.2.4.1 Automatic Grouping Dialog”, p234).

The **Schedule** button opens the Subject Info Schedule dialog, which is used to control when items are prompted and logged (see “6.2.6 Subject Info Schedule Dialog”, p237).

New/Reconfigure Info Item Dialog

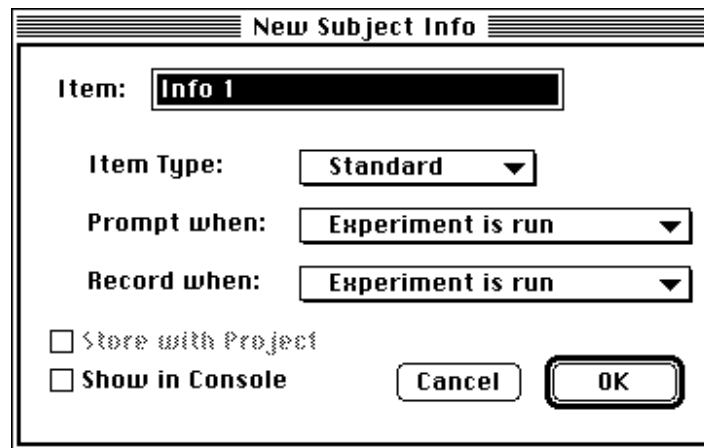


Figure 183 – The New/Reconfigure Info Item dialog

This dialog is opened when the **New...** or **Configure...** button in the Subject Info window is hit. Hitting **Configure...** for an item in the Subject Info window allows you to change anything about the info item except its type (which shown in the **Item Type:** pop-up menu).

The first item in the dialog is a text box for the item's name. As with the names of objects in the experiment hierarchy, this name must be unique, and cannot include certain symbols or keywords (see “5.2.5 New Object Name Dialog”, p114).

The **Item Type:** pop-up menu selects the type of information stored by the item. The available types are:

- **Standard** – Simple text.
- **Value** – A (possibly bounded) numerical value.
- **Buttons** – A one-of-many value selected with radio buttons.
- **Checkboxes** – A many-of-many value selected with a list of checkboxes.

The **Prompt when:** pop-up menu sets a time for automatic prompting. At the specified time (e.g., **Start of Experiment**), the user will be prompted to set the value of the item.

The **Record when:** pop-up menu sets when the item will be recorded to the log file. (This setting is reflected by the **Record with Log** checkmark in the Subject Info Window).

The **Prompt when:** and **Record when:** parameters can also be controlled through the Subject Info Schedule dialog (see “6.2.6 Subject Info Schedule Dialog”, p237).

The **Store In Project** checkbox sets whether the info is to be stored in the project (if a project is being used) or in the local script. *Not yet supported.*

Checking the **Show In Console** checkbox causes the info to be shown in the main console. The user will be able to change the item's value by clicking on it in the console.

For a new item of any type except **Standard**, a Define Subject Info dialog will automatically follow this dialog to further define the item (e.g. to set ranges for a **Value** item or to set the possible values for a **Buttons** or **Checkboxes** item). For an existing item, the Define Info Item dialog can be opened with the **Define...** button in the Subject Info dialog.

Define Subject Info Item Dialog

This dialog is opened in one of three ways:

- Automatically after **OK** is hit in the New Subject Info Item dialog.
- By clicking on the **Define...** button in the Subject Info window.
- By double-clicking on an item in the Subject Info Window.

The purpose of the dialog is to further specify the type of information stored by the item. Hitting **Cancel** in this dialog when it was opened by the New Subject Info Item dialog also cancels the item's creation.

There is no one Define Subject Info Item dialog; the actual dialog that will be used depends on the item's value type. The possible dialogs are described below.

Value Item Definition Dialog

The radio buttons at the left of the dialog determine the type of number stored by the item: **Integer**, **Real**, or **Rational** (fractions). To specify a range of numbers, click on the **Range** checkbox and set upper and lower bounds on the number.

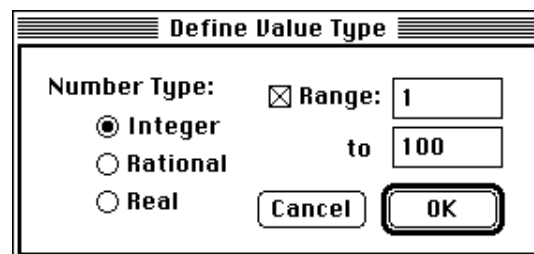


Figure 184 – The Value Item Definition dialog

Buttons Item Definition Dialog

This dialog gets a list of button names. Items are created and deleted in the usual way, using the **New...** and **Delete** buttons.

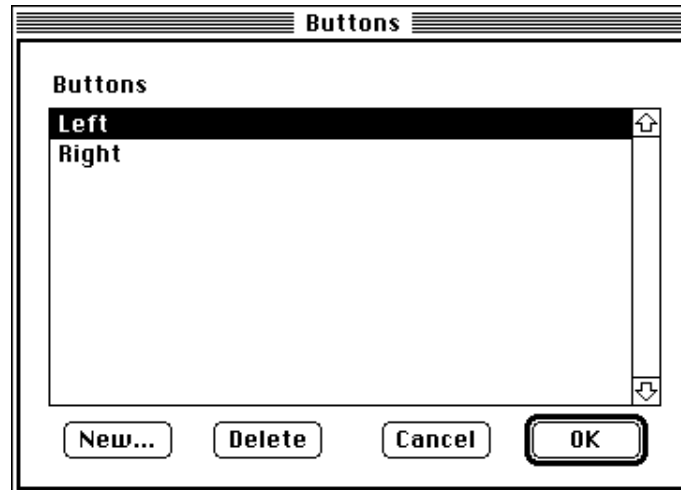


Figure 185 – The Buttons Item Definition dialog

Checkboxes Item Definition Dialog

This dialog gets a list of checkbox names. It is the same as the Buttons Item Definition dialog (see above).

6.2.2 Subject Info and the Log File

The log file is described in “6.1.5 The Log File”, p222. It plays crucial role in tracking subject information for subject number calculations (see “6.2.3 Subject Number Calculation”, p230).

When you create Subject Info items, you will need to record the data for a particular subject in the log file or data file; otherwise, this information will not be recoverable after the next subject is run. Where Subject Info items are stored is controlled by checkboxes in the Subject Info dialog (see “6.2.1.2 Subject Info Dialog”, p226).

6.2.3 Subject Number Calculation

It is often useful to assign each subject in the experiment a number; this number might be used to vary the experiment design, or to name a data file. One way to assign a number would be to create a numerical Subject Info and force the experimenter to type in a number for each subject that is run. Fortunately, you do not have to do this: given the current sub-

ject's name and a history of previous subjects (in the log file), PsyScope can calculate a variety of indices for you:

Subject number – An index for the subject *within* the subject's groups.

Run number – An index for the run of the subject within this experiment, in case the same subject is run multiple times.

Group run number – A run index that is counted across all subjects within this subject's group.

Subject count – An index for the subject across all groups.

Run count – A run index across all subjects run in this experiment.

Group number – An index for the subject's group.

In all of the above definitions, “this subject's group” refers to a group determined by the automatic grouping criteria (see “6.2.4 Automatic Grouping”, p233). If there is no automatic grouping, all subjects are placed into the same group.

These numbers are most commonly used to vary an experiment design using the **Latin Square** or **Between** crossing type. These crossing types take an index — set through the Choose Crossing dialog (see “5.7.2.6 Choose Crossing Dialog”, p143) — which determines exactly which cells of a table are going to be used. See also “5.7.2.2 Table Info Dialog”, p140.

The numbers can also be used as group criteria (e.g. to place even-numbered subjects in one group and odd-numbered subjects in another). Scripters may find these numbers to be useful for other purposes.

Note: SubjectNumber, RunNumber, etc. can be used for automatic grouping, but these criteria will not affect the calculation of the above numbers, since the definitions would become circular.

6.2.3.1 Logging and Scheduling Correctly

For the subject numbers to be calculated correctly, you must obey a few scheduling constraints:

- You must set (i.e. prompt for) all of the relevant information about the current subject before the subject number calculation is performed.
- All of the current subject's information must be logged before the experiment is executed.
- When logging information, you must log the subject's name before any other information.

- You must log the subject name, and all items used for determining a subject's group.

The order in which items are prompted, calculated, and logged is controlled through the Subject Info Schedule dialog (see “6.2.6 Subject Info Schedule Dialog”, p237).

It is not important whether you log the current subject's information before or after you calculate the subject number, unless you want to log one of the numbers; then, of course, you must log these items after the calculation is performed.

Note: Logging subject and run numbers is a good idea, since this information can be used by the calculation process if an early part of the log file is deleted (see below).

6.2.3.2 How the Subject Number Calculations are Performed

The subject numbers are calculated by reading through the log from the start, following this procedure:

When a “SubjectName” keyword is encountered (indicating that the subject name was logged) item information is collected until an “ExperimentRun” keyword is encountered (indicating that the subject was in fact run in the experiment).

If the subject was logged for an experiment (indicated in “ExperimentRun”) that is not the current experiment, the subject information is ignored.

Otherwise, this run contributes towards the subject count and run count calculation. If the info values between the “SubjectName” and “ExperimentRun” keywords match the current group, then subject number and group run count will also be affected.

If the value of the “SubjectName” item that was found matches the current subject, then run number will be incremented.

This process is repeated until the entire log file has been read, counting subjects and runs along the way. If the current subject is found to be already in the log file, the subject number assigned to the subject the first time will be used.

If “SubjectNumber” and “RunNumber” keywords are found for subjects in the log file, they can affect the counting of subjects and runs. For instance, if the first subject found in the current group has “SubjectNumber” info with a value of 5, then it is assumed that subjects 1 through 4 were deleted from the beginning of the log file. However, only subject number and run number are treated this way; accurate subject, run, and group run counts require that all subjects be kept in the log file.

6.2.4 Automatic Grouping

“Automatic grouping” refers to the automatic selection of a one group in the experiment structure to be used for the next execution of the experiment. Which group is selected depends on the current values of certain Subject Info items; these items are called the *grouping criteria*.

The Automatic Grouping dialog controls the selection of items as grouping criteria and the mapping from criteria values to groups.

Automatic grouping is generally performed at the same time as the subject number calculation; the time at which this occurs is controlled through the in **Assign Group and Calculate Numbers** item in the Subject Info Schedule dialog. Usually, you will schedule group calculation just before the experiment is run, but after all the Subject Info item values have been set for the current subject. See also “6.2.3.1 Logging and Scheduling Correctly”, p231.

Note: Although the process of selecting a group for the experiment usually occurs at the same time as the subject number calculations, automatic grouping does not involve the log file in any way; only the current values of the Subject Info items are relevant. Psychers may find it useful to disassociate the subject number calculations and group selection.

6.2.4.1 Automatic Grouping Dialog

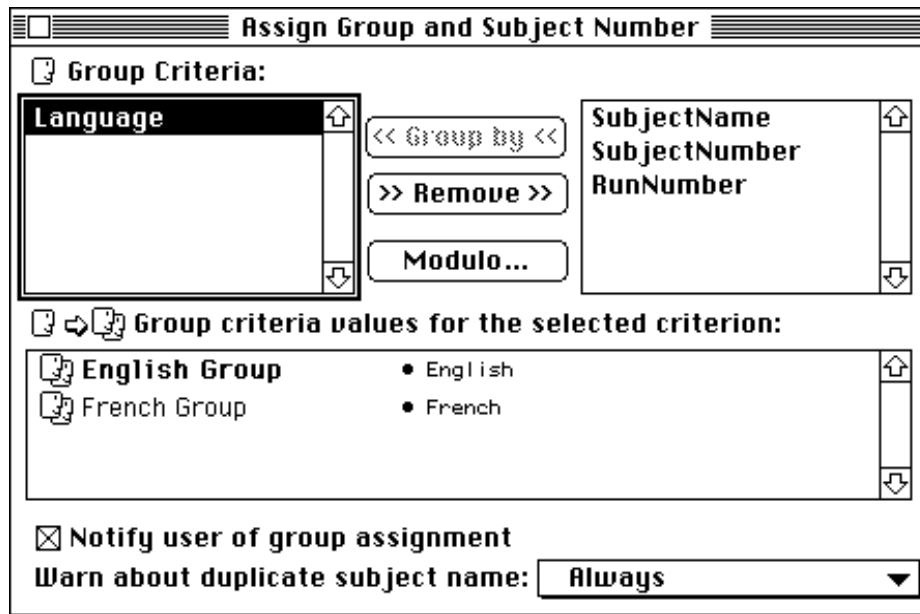


Figure 186 – The Automatic Grouping dialog

The Automatic Grouping dialog is used to configure automatic assignment of a group for the current subject. To open the Automatic Grouping dialog, click on the **Auto Grouping** button in the Subject Info dialog.

At the top of the dialog are two lists: the list on the left shows the subject info items that are used to assign the subject to a group, i.e. the grouping criteria; the list on the right shows subject info items that are not used to assign a subject to a group (i.e. the non-criteria).

To move an item from the non-criteria list (on the right) to the criteria list (on the left), select the item and click on the **<< Group by <<** button. To move items in the other direction, click on the **>> Remove >>** button.

The lower part of the dialog contains a list of all groups currently defined in the experiment. When you select a criterion in the grouping criteria list, the value specific to each group is shown to the right of the group in this list. To change the value for a group, click on it.

To open the Group Criteria dialog and display all criteria values for a single group, double-click on a group in the group list. (See “6.2.4.2 Group Criteria Dialog”, p235.)

Subject Info items (or values read from the log file) are compared to group criteria character-by-character (ignoring capitalization), unless the criteria uses a *modulo comparison*. In a modulo comparison, both the item value and the criterion are converted to numbers; then, the numbers are compared modulo some value. The comparison mode and modulo value for a criterion are set by clicking on the **Modulo...** button.

If the **Notify user of group assignment** box is checked, the user will be notified of the group assignment — through the standard Message dialog — after an automatic assignment is performed.

The **Warn about duplicate subject name** pop-up menu controls a feature to help insure that the subject name is updated before every execution of the experiment. This setting is actually used by the subject number calculation, instead of the automatic grouping process.

When the **Warn about duplicate subject name** pop-up menu is set to **Always**, the user is warned any time the current subject name is found already in the log file. The **For Same Experiment** setting gives the warning only if the subject name was previously logged for the current experiment. The **After Reloading Script** setting gives the warning only if the script has been changed or reloaded since the subject name was logged. The **After Restarting** setting gives the warning only if PsyScope has been restarted since the subject name was logged.

6.2.4.2 Group Criteria Dialog

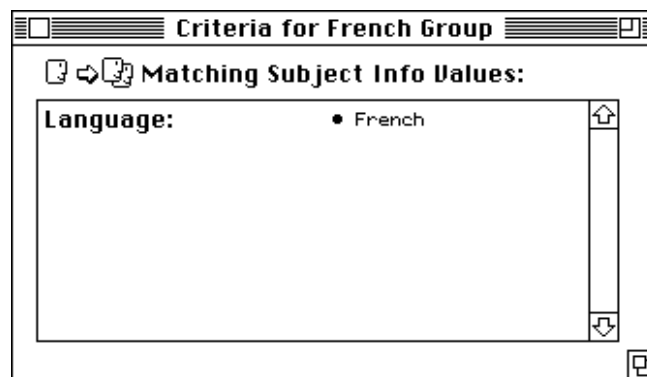


Figure 187 – The Group Criteria dialog

The Group Criteria dialog shows all of the Subject Info grouping criteria and their values for a single group. To open the Group Criteria dialog, click on the **Criteria** button in the Group dialog, or double-click on a group in the Automatic Grouping dialog.

Each criterion is shown as an item in a list. The value for that criterion within the group is shown next to the criterion name. This value is changed in the usual way: by clicking on it.

6.2.5 Data File Dialog

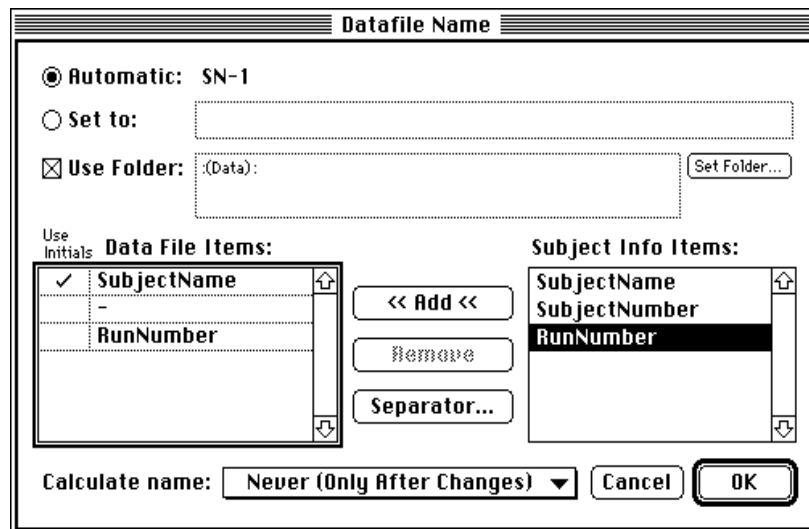


Figure 188 – The Datafile dialog

The Data File dialog is used to configure automatic data file name generation. To open the Data File dialog, click on the **Data File** button in the Subject Info dialog.

The two radio buttons at the top of the dialog control whether a data file name is calculated from Subject Info item values (i.e. **Automatic** is selected) or a fixed name is used (i.e. **Set to** is selected). The rest of the dialog applies only if **Automatic** is selected.

If the data file is to be stored in a directory other than the default directory (which is usually the location of the script; see “6.1.2 Path Names”, p215 for more information), then **Use Folder** should be checked. The folder to be used is shown in the outlined area to the right of the checkbox. The folder can be changed by hitting the **Set Folder...** button.

The list to the left of the dialog controls what information is used to generate an automatic file name; a file name can be built by concatenating Subject Info item values and static separators. The order of the items in the list corresponds to the order they are used in the file name. A checkmark next to an item causes the item’s initials to be used instead of the full item value.

On the right side of the dialog is the current list of Subject Info info items. To add a Subject Info item to the list of items used in the data file name, highlight it and click on the **<< Add <<** button. To add static separator characters, click on the **Separator...** button. The **Remove** button deletes items or separators from the data file items list.

The data file name is automatically recalculated anytime one of the items it uses is changed through a prompt dialog. You can also schedule a recalculation using the **Calculate name:** pop-up menu at the bottom of the dialog. (This can also be set through the **Calculate Data File Name** item in the Subject Info Schedule dialog; see “6.2.6 Subject Info Schedule Dialog”, p237.)

6.2.6 Subject Info Schedule Dialog

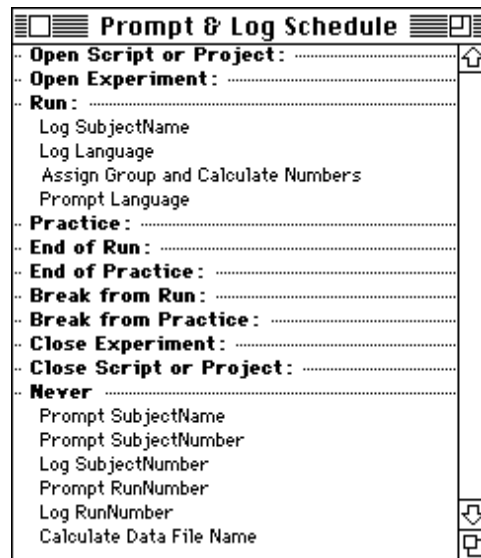


Figure 189 – The Subject Info Schedule dialog

The Subject Info Schedule dialog is used to control a number of *processes*:

- When the user is prompted for the value of a Subject Info item (**Prompt...**).
- When Subject Info items are recorded in the log file (**Log...**).
- When a group is automatically assigned and subject numbers are calculated (**Assign Group and Calculate Numbers**).
- When the automatic data file name is calculated (**Calculate Data File Name**).

(See also “6.2.4 Automatic Grouping”, p233 and “6.2.5 Data File Dialog”, p236.)

To open the Subject Info Schedule dialog, select the **Schedule** button in the Subject Info dialog.

Processes at the bottom of the list — in the **Never** section — are unscheduled. To schedule a process in this list, drag it to one of the time sections. To schedule a process to happen twice, option-drag it.

The schedule times represented by the sections are as follows:

Open Script or Project – Processes in this section are performed when the script is initially loaded into PsyScope (and a project is not used), or when a project is loaded into PsyScope. (See “6.1.1 Using Projects”, p213 for more information on projects).

Open Experiment – Processes in this section are performed when the current experiment is changed *to* this one in PsyScope (by changing the experiment in the Console’s

pop-up menu or by opening the script or project). Note that **Open Script or Project** processes are always followed by **Open Experiment** processes, although **Open Experiment** processes will not be preceded by **Open Script or Project** process if the current script is changed within a project.

Run – Processes in this section are performed just before the experiment is executed in Run mode (as opposed to Practice mode).

Practice – Processes in this section are performed just before the experiment is executed in Practice mode (as opposed to Run mode).

End of Run – Processes in this section are performed just after the experiment execution is completed in Run mode (as opposed to Practice mode or ending by breaking).

End of Practice – Processes in this section are performed just after the experiment execution is completed in Practice mode (as opposed to Run mode or ending by a breaking).

Break from Run – Processes in this section are performed just after the experiment execution is broken (by Command-.) in Run mode (as opposed to Practice mode or a completed execution).

Break from Practice – Processes in this section are performed just after the experiment execution is broken (by Command-.) in Practice mode (as opposed to Run mode or a completed execution).

Close Experiment – Processes in this section are performed when the current experiment is changed *from* this one in PsyScope (by changing the experiment in the Console's pop-up menu or by opening a different script or project). Note that **Close Script or Project** processes are always preceded by **Close Experiment** processes.

Close Script or Project – Processes in this section are performed when the script is closed (when a project is not used), or when a project is closed. (See “6.1.1 Using Projects”, p213 for more information on projects).

6.3 The Trial Monitor

The *Trial Monitor* is a part of the PsyScope user environment that gives the user greater control over the running of an experiment, allowing interactive adjustment of some parameters of the experiment. It also includes features for use in debugging a script.

The Trial Monitor can be accessed in either of two ways:

- Select the **Monitor** item from the **Windows** menu.

- Click on the **Monitor** button in the Console.

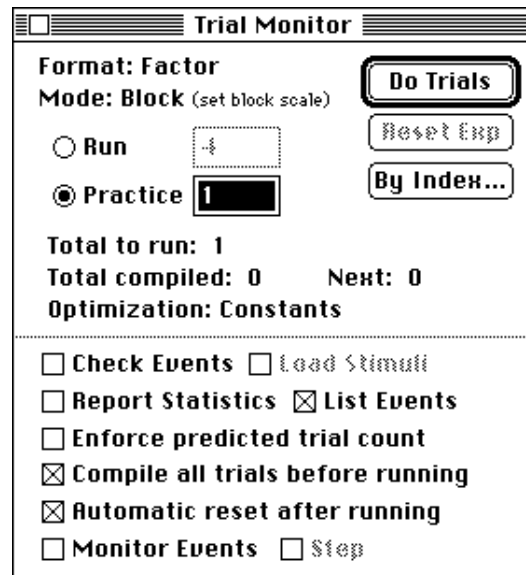


Figure 190 – The Trial Monitor

The **Format** line tells you which scripting format has been specified in the script file. (This is solely for your information, and cannot be changed from the Trial Monitor.) In the example above, the console informs the user that the Factor format has been specified. (Factor is the default format. See “, Chapter 13. Experiment Scripting Reference”, p357 for information on the other available formats.)

The **Mode** line tells you which mode of the format is being used, and how the values in the boxes are used. The exact content of these lines depends on the format.

For Factor format, the mode will be **Block** or **Direct**. **Block** mode means that the experiment contains blocks, and the text boxes below are used to specify a block scaling value for the experiment. **Direct** mode means that there are no blocks in the experiment, so the text boxes are used to specify total trial counts.

Just below the **Format** and **Mode** fields are two radio buttons: **Run** and **Practice**. These are used to select the mode in which the experiment is to be run. The script may be configured to operate differently in these modes. Each mode button has a box next to it in which you can change the number of trials to be run in that mode.

When one of the modes is selected, the actual number of trials to be run is displayed below the boxes. For example, in the trials console shown above, the **Total to run** is set to “1”, since that is the number specified for the “Practice” mode. If the **Run** button were clicked, **Total to run** would change to “4”.

The value of **Total to run** will generally not be correct if block durations are used, if the `RerunTrial[]` action is executed, or if the `QuitBlock[]` action is executed, because these features dynamically change the number of trials which are executed. (You can, however,

use **Enforce predicted trial count** to force the number of trials actually executed to be the same as **Total to run**; see below.)

At the top right of the console are the **Do Trials**, **Reset Exp**, and **By Index...** buttons. **Do Trials** causes the specified number of trials to be run in the selected mode. When the trials are completed, the experiment will still be active — i.e. the experiment is paused, but not ended, so that more trials may be run without restarting the whole experiment. Hitting **Reset** will then end the experiment, possibly writing out the data for the trials that were run.

By Index... may be used to run specific trials by specifying starting and ending trial numbers. If the start number is omitted, it is defaulted to 1. (The indexing of trials starts with 1.) If the end trial number is omitted, it is defaulted so that the number of trials specified in the trials console will be run.

At the bottom of the Trial Monitor are a number of checkbox options for running the trials. The first is **Check Events**, which causes the trials to be compiled, but not run. Check mode is useful for obtaining compile time and optimization statistics without actually running the trials.

Load Stimuli works in conjunction with **Check Events**; **Load Stimuli** specifies that the stimuli for the experiment should be loaded during the check. This may be useful for testing sound file paths, for instance, or for checking the amount of time required to load the stimuli.

Checking **Report Statistics** causes a statistics window to be opened in a standard editing window after the trials are run or checked. See “6.3.1 Trial Compilation Statistics”, p241 for information on the contents of the window.

Checking **List Events** causes a window with debugging information to be displayed after the trials are run or checked. The debugging information includes:

- the condition name for each trial
- the name of each event in the trial
- the stimulus string for each event (in parentheses, following the event name)

If **Enforce predicted trial count** is checked, block durations and special actions are ignored so that the number of trials executed will match the number of trials predicted in **Trials to run**.

If **Compile all trials before running** is checked, the interpreter will process all the trials to be performed before running the experiment. (Usually, one trial is processed and then run at a time.) This option is available in case compiling each trial takes more time than can be spared between each trial, but limits some of the functionality of PsyScope. See “6.5.1 Precompiling”, p246 for more information.

If **Auto reset after running** is checked, the experiment will be automatically reset after the trials are run, rather than falling into the “paused” state described above.

Checking **Monitor Events** turns on the Event Monitor debugging system, which shows the status of events and actions as the trial is run. If **Step** is also checked, the program will run only one event at a time; to go to the next event, hit the space bar. Like **Check Events**, these options are useful for debugging your script. See “6.4 The Event Monitor and Variable Monitor”, p241 for more information.

6.3.1 Trial Compilation Statistics

When trials are run from the Trial Monitor and the **Report Statistics** box is checked, a statistics window opens when the trials are completed. This window contains compile time and optimization reports.

The first section of the statistics report shows the total, average, minimum, and maximum times for:

Compile – This includes the time used to select factor levels and then read and parse attributes from the script. Typically, the first couple of trials in an experiment take longer to compile than later ones, since the script interpreter may have to parse some entries for the first time. Trials that are optimized or that have optimized events will usually compile much faster than unoptimized trials.

Initialize – This includes the time used by the Trial Manager to initialize a trial so that it is ready to run. This time is typically very short.

Load – This includes the time used by output devices to load stimuli into memory. If trials are compiled in **Check Events** mode and **Load Stimuli** is not checked in the Trial Monitor, this value will not reflect the actual load time.

Run – This includes the time spent actually running a trial. This value will be 0 if trials are checked instead of run. If your trials have a fixed duration, it should be reflected in this value.

In Factor format, the second half of the statistics shows how many events were optimized during compilation and how many were not. If very few trials were optimized, you may have optimization turned off, or the experiment may not be very optimizable. (See also “Optimization”, p167.)

6.4 The Event Monitor and Variable Monitor

The *Event Monitor* and Variable Monitor are tools for debugging experiments. They are accessed through the **Monitor** checkbox in the Trial Monitor (see “6.3 The Trial Monitor”, p238). The Event Monitor provides information about a trial while the trial is running so that the experiment designer can see more clearly what is happening while a trial is running. The Variable Monitor similarly provides information about Trial Manager Variables used in the experiment.

6.4.1 The Event Monitor

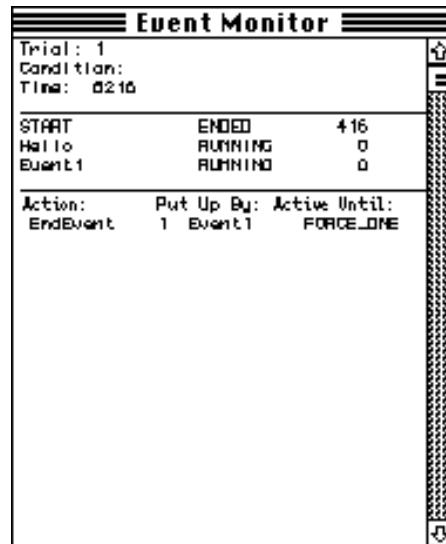


Figure 191 – The Event Monitor

The Event Monitor has three panes which provide three types of information as a trial is running. The upper pane contains trial information. The middle pane contains event information, and the bottom pane provides action information.

6.4.1.1 Trial Information

There are three pieces of information given in the Trial Information pane of the Event Monitor: the trial number, the trial condition, and the current time.

Trial Number

The trial number field gives the number of the current trial. This is the same value that would appear in the data file for any RT[] action executed during the trial.

Trial Condition

The trial condition field gives the condition name of the current trial. This is the same value that would appear in the condition field of the data file. See also “Condition Name”, p175.

Current Time

This is the time of the last Event Monitor update. It is the number of milliseconds of trial time since the start of the trial. Trial time is measured as the time taken to run the trial minus the time taken to update the Event Monitor. See “6.4.2.1 Perceived Times”, p244 for more on the times in the Event Monitor.

6.4.1.2 Event Information

The event information pane of the Event Monitor window lists each event in the current trial, the status of each event, and the actual duration. Each row of the display represents an event, with the name of the event in the first column, the status of the event in the second column, and the duration of the event in the third column.

Event Statuses

The second column of the event information display contains the event status for each event. This tells whether or not an event has been loaded, queued, run, cleared, etc. The possible values are:

- ON DECK** – The event exists in the trial, but its stimulus has not been loaded.
- LOADED** – The stimulus for the event has been loaded into memory.
- ON QUEUE** – The event is queued to run, but has not begun to run yet.
- RUNNING** – The event has begun to run, `START[]` actions have been triggered and executed.
- MASKED** – The event has been masked using `MaskEvent[]`.
- CLEARED** – The event’s stimulus has been cleared. The actual of the duration is now known and will be displayed. `END[]` actions have not been triggered yet.
- ABORTED** – The event was aborted using `AbortEvent[]`. `END[]` actions will not be triggered unless the event is run again.
- ENDED** – The event was ended normally. `END[]` actions have been triggered and executed, and any actions that were active until this event should have been removed.

Event Times

The third column of the event information display contains the time for each event in the current trial. The time displayed for each event is the actual duration of that event in milliseconds. For each event, the display will read 0 until the event status reaches `CLEARED`, `ENDED`, or `ABORTED`, at which point the actual duration stored for that event will be displayed in the Event Monitor.

Like the current time displayed in the trial information pane, the durations shown are in trial time, which does not include the time taken to update the monitor. See “6.4.2.1 Perceived Times”, p244 for more information on trial time.

6.4.1.3 Action Information

The third pane of the Event Monitor window contains action information. It contains a list of each currently active action, the number of instances active for that action, the event that posted the action, and the event or other condition until which the action will be active.

Instances

The second column of the actions information pane contains the number of instances of the action that are still active. If the action has more than one instance active this value will decrease each time the action is triggered until the last instance is triggered, or the action is cancelled.

Put Up By

The third column contains the name of the event that put up the action. If the action is a trial action, the event “START” will be listed here.

Active Until

The last column contains the name of the event or other trial condition until which the action is active; this is the value of the **Active Until** parameter of the action. Possible values are:

the name of any event in the trial; active until the end of that event

TRIAL_END; active until the end of the trial

FORCE_ONE; active until at least one instance has triggered

FORCE_ALL; active until all of the instances have triggered

6.4.2 Event Monitor Operation

The Event Monitor operates by keeping track of the current state of the Trial Manager, and updating the contents of the Event Monitor window when the Trial Managers state changes.

At the beginning of each trial, the event monitor prints the current trial number and condition, and lists all events in the trial with the status ON DECK. Each time an event is run or ended, or any other action occurs, the Event Monitor window is updates with the current state of the Trial Manager.

At each update a number of things occur:

- The current time is written in the trial information pane.
- The status of each event is checked, and changed if necessary.
- The action queue is read, the action list is updated to reflect the current queue.

6.4.2.1 Perceived Times

In order to prevent the time taken to update the event monitor from interfering with the timing of events in the Trial Manager, that time is not taken into account when calculating trial

times while running trials using the Event Monitor. When the event monitor is updated, the time is recorded at the beginning of the update, and the trial clock is reset to that time as soon as the update is finished. In essence, trial time is “frozen” for the duration of the update.

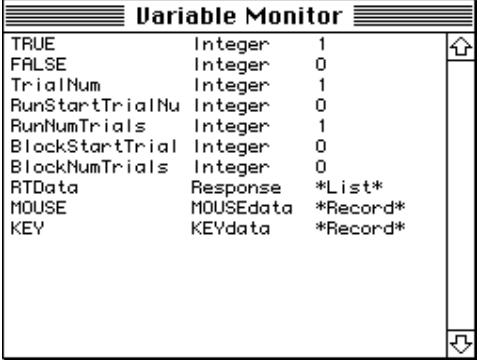
For this reason, event durations of synchronous events, such as text, and pictures, will seem longer than they should be. Each event will be “stretched” by the amount of time that was needed to update the Event monitor. On the other hand, asynchronous events that have an inherent duration, like sounds, will not be stretched in this way, because they are not dependent on the Trial Manager to continue their presentation or to clear them. Actions that depend on sounds and other events with inherent durations, however, *will* be delayed by Event Monitor updates.

6.4.2.2 Step Mode

When **Step** is checked in the Trial Monitor window, it puts the Event Monitor in *step mode*. In step mode, the Event Monitor waits after every update until the space bar is pressed before allowing the Trial Manager to continue executing the trial. This allows the experiment designer to see the state of the trial after each action, and to “step” through the trial at his own pace. This can be helpful when debugging complicated sequences of events that take place in a very short period of time.

In step mode trial time is “frozen” (see above) until the spacebar is pressed. Synchronous event types that depend on the trial manager to end them will be extended as long as the trial is stopped, but asynchronous event types with an inherent duration not dependent on the trial clock will continue to run.

6.4.3 The Variable Monitor



Variable Monitor		
TRUE	Integer	1
FALSE	Integer	0
TrialNum	Integer	1
RunStartTrialNu	Integer	0
RunNumTrials	Integer	1
BlockStartTrial	Integer	0
BlockNumTrials	Integer	0
RTData	Response	*List*
MOUSE	MOUSEdata	*Record*
KEY	KEYdata	*Record*

Figure 192 – The Variable Monitor

The Variable Monitor lists all of the trial variables that are used in the experiment. The figure above shows the Variable Monitor with the standard built-in variables. See also “5.10 Trial Manager Variables”, p205.

The first column of the Variable Monitor shows a variable name. The second column is the type of the variable. The third column shows the variable's current value if it is a scalar, or *List* or *Record* for arrays and records, respectively.

6.5 Space and Speed

In an ideal world, all experiments would run displaying their stimuli instantaneously while using the least possible amount of memory. Unfortunately, the world is not ideal, and very often it is necessary to slow down the displaying stimuli in order to minimize memory use, or to use more memory in order to insure that stimuli are displayed as quickly as possible.

This section is intended to help the experiment designer decide what measures need to be taken in order to achieve optimal memory usage and time overhead for his or her purposes. Of particular interest is the *intertrial interval*, or *ITI*; this is the time that elapses between the end of one trial and the beginning of the next.

This section explains a number of problem areas in which trade-offs between speed and memory need to be made. It also describes some situations in which some other quality (such as picture quality) can be sacrificed in order to reduce memory usage without sacrificing speed.

6.5.1 Precompiling

Before a trial can be executed, it must be *compiled*. Compiling is the process of reading the experiment structure from the script and generating a trial that can be executed by the Trial Manager. The compilation process must choose a block, template, and condition for the trial, and then read all the trial and event attributes in this context.

By default, PsyScope compiles each individual trial just before it is run. For very complex experiments, the time required to compile a single trial may become so significant that it affects the ITI.

One way around this problem is to compile all of the trials before executing any trials; this is called *precompiling*. When trials are precompiled, the compile time does not contribute to the ITI — there is just some waiting time at the beginning of the experiment. (The subject can be reading experiment instructions during this time; see “Instructions File”, p165.)

Precompiling can be turned on in a couple of ways. There is a **Precompile** checkbox in the Trial Monitor (see “6.3 The Trial Monitor”, p238); checking this box will cause all experiments run in that copy of PsyScope to be precompiled before running. (If a project is used, this checkbox applies only to scripts in the project; see “6.1.1 Using Projects”, p213.)

There is also a **Precompile** experiment attribute; in this attribute, you specify a number of trials to be precompiled (the rest will be compiled normally), or **All** trials. This attribute applies only to the current experiment, but it can be overridden by the **Precompile** checkbox in the Trial Monitor (i.e., the checkbox can force all of the trials to be precompiled).

6.5.1.1 Problems with Precompiling

Unfortunately, precompiling is not a completely general solution; precompiling can sometimes produce different results than regular, trial-by-trial compiling. This happens when *compiled* information for one trial depends on the *execution* of another.

Note that this is different from *run* information being execution dependent; for example, trial variable values may be changed from trial to trial, but a reference to a trial variable in an expression cannot be correctly compiled without executing the previous trials. The expression will not be evaluated until the trial is actually executed (by which time the trial variable will have the correct value).

Precompiling does not work under the following circumstances:

- Block durations are used – For most experiments, it is impossible predict in advance how many trial will be run in a block that uses durations; even when it is possible (i.e. the trials all have a fixed duration), PsyScope does not attempt to calculate a count. Instead, duration-based blocks are assigned a trial count of 1.
- Attributes are linked to trial variables – When an attribute is linked to a trial variable (which is different from using the variable in an expression; see “5.10.4 Linking to Variable Values”, p207), the attribute depends on the variable’s value to have changed by the time the trial is compiled. However, with precompiling, no actions will have been run, so the variable values will all stay the same.
- The `ScriptEval[]` action is used, and attributes are linked to the result; in this case, executing a trial could modify the script. If any other trial depends on this modification, it will not have been made at precompile time.

6.5.2 Loading Stimuli

6.5.2.1 Load Time

Before PsyScope presents a stimulus, it must be loaded into a buffer in memory. Loading stimuli not only occupies memory, but also takes time, the amount of which can vary from trial to trial. Depending upon the nature of your stimuli, and the demands of your experiment, you may be concerned about one or the other of these factors.

Loading Text

PsyScope has the capability of displaying text stimuli in any font, size, face and color available on the system. The Macintosh system, however, was not designed with the idea that users would want to have text displayed no more than a few milliseconds after it has been requested.

Because of this, the Macintosh Font Manager only keeps the most recently used font in memory, and when it is not using it, will allow it to be purged if that memory is needed for

something else. The result is that it often takes as long as 60 milliseconds for text to be displayed, and in some extreme conditions can take as long as 250-400 milliseconds.

To get around this problem, the Trial Manager has PsyScope's screen device draw any text that is loaded at the beginning of the trial into a buffer offscreen and then simply copy the buffer to the screen when the stimulus is displayed.

The two main factors in increasing the font loading time are the use of color and the use of resource management utilities such as Suitcase II™. (These utilities allow you to keep a large number of font files open at the same time. The trouble is that PsyScope must then search through all these files in order to find a particular font.) In many cases, running in monochrome mode and without many open font files will reduce the font loading time (which was moved to the ITI) to acceptable levels.

It is possible that an experiment with many large text stimuli in a trial will use great amounts of memory. To avoid this, specify a **Load Time** of 1 ms or more (see “ Load Time”, p180). When text stimuli are loaded just before playing, they are not written into an offscreen buffer, but written directly to the screen, since the font would have to be loaded anyway. Of course, the font loading time will be incurred during the trial instead of at the beginning. If a constant inter-stimulus interval is necessary, then **Load Time** can be specified to be a value greater than the longest incurred font loading time, and the Trial Manager will insure that it loads the stimulus the given amount of time before it plays the stimulus.

Loading PICTs

In general, the time it takes to load a PICT depends on:

- The PICT's size — The larger a PICT file is, the longer it will take to load from disk and draw it off-screen to prepare it to be displayed.
- Whether the PICT is a bitmap or a line drawing — A bitmap will take longer to load than a line drawing.
- The depth of the PICT, and the relation between the PICT definition's depth and the display depth — In general, a PICT will take longer to load when the screen depth is different from the depth of the PICT definition than when they are the same.
- Whether the PICT has to be scaled — By default, PICT stimuli are scaled to fit exactly within the borders of the stimulus port in which they are to be displayed. However, the **Draw picture actual size** feature attribute of a PICT stimulus can be turned on to cause the PICT to be drawn the same size as it was in the original drawing.

The **Keep picture information in memory** feature attribute can reduce the amount of time taken up loading PICTs in some cases. See “ Keeping Stimuli in Memory”, p251 for more information.

Loading Sounds

The main consideration in loading sounds — both in terms of time and memory consumed — is the size of the sound. The larger a sound is, the longer it will take to load and the more memory it will occupy.

In general, higher fidelity means a larger sound. For example, a sound with 16-bit samples will be twice as large as one with 8-bit samples. A stereo sound will be twice as large as a monaural sound, and a sound sampled at 44.1 KHz sampling rate will be twice as large as one sampled at 22.5 KHz sampling rate. A CD-quality stereo sound (16-bit, 44.1 KHz) will be 8 times as large as a sound recorded with MacRecorder™ on SoundEdit™ (8-bit, mono, 22.5 KHz).

The **Keep sound in memory** feature can reduce the time consumed by loading sound stimuli in some cases. See “ Keeping Stimuli in Memory”, p251 for more information.

The **Play in parallel** feature also has an effect on the load time of a sound. When a sound has to be played in parallel with other sounds, it must be played in a separate sound channel. Thus, a new sound channel must be allocated at load time for every sound for which the **Play in parallel** feature is turned on. Allocating a sound channel can take 50ms, more or less depending on the speed of the machine you are running on and how much memory is currently available.

6.5.2.2 Controlling The Load Procedure

By default, PsyScope loads all stimuli for a trial during the ITI, before the trial begins. Once a stimulus is presented, it is cleared from memory, so that when the trial is complete all of the memory has been freed and the stimuli for the next trial are loaded. Each ITI lasts as long as it takes PsyScope to load the stimuli for the next trial.

There are two problems with loading all of the stimuli into memory at the beginning of a trial:

- 1) Keeping several large stimuli in memory may take a great deal of memory, perhaps more than is available.
- 2) Moving the loading to the beginning of the trial does not eliminate the time delay, it just causes it to occur at a (hopefully) more convenient time.

You can control this process in several ways:

- setting the minimum ITI.
- preloading all stimuli at the beginning of the experiment.
- keeping stimuli in memory across trials.
- loading stimuli event by event.

Each of these addresses a different concern, as discussed below.

Setting the Minimum ITI

The **Minimum ITI** trial attribute sets the minimum amount of time that PsyScope waits between trials. PsyScope will load all of the stimuli for the trial during this period, but will wait to begin the trial until the specified amount of time has elapsed.

You can use this attribute to ensure that the amount of time between trials is constant. First, you should use the Trial Monitor to check the maximum amount of time it will take to load stimuli before a trial (see “6.3.1 Trial Compilation Statistics”, p241). Then, assign an equal or greater value to the **Minimum ITI** attribute. Be sure to run the load time test on the slowest machine that you will be using to run the experiment.

Setting **Minimum ITI** is the best option to use if you do not have the memory available to preload all of the stimuli at the beginning of the experiment (discussed next), but need to maintain a consistent interval between trials.

Many experiment designs have an “ITI” event (as the last event in the template) which is supposed to control the time between trials. If you are not checking for input during this ITI, it is much better to use the **Minimum ITI** trial variable, since this allows any sort of compilation or loading work to be done during the ITI. You can combine a shorter “ITI” event with **Minimum ITI** to obtain a variable ITI.

Preloading All Stimuli

For some experiments, it may be important to reduce or eliminate the ITI. You may want to reduce it because the experiment contains large stimuli that are stored on disk (such as long sounds or big PICTs, that take a while to load), or stimuli that must be manipulated in some way (for example, PICTs that are flipped, which also takes a while). In other cases, you may need to eliminate the ITI altogether. Even loading simple text stimuli can take time (the bitmap must be constructed for the characters in the string; see “Part 4: Scripting Reference, 15.2.2 How a Screen Stimulus is Drawn and Cleared”, p445) which may interfere with the experimental design.

If your experiment can be precompiled (see “6.5.1 Precompiling”, p246), then precompiling may greatly reduce the ITI. If you *can* precompile an experiment, you can also load all of the stimuli at the beginning of the experiment by checking the **Preload all stimuli with precompiling** checkbox in the **Special** experiment attribute (see “Special”, p165).

Using the **Preload all stimuli with precompiling** option and **Precompile** mode, you can ensure that there is no delay between trials. (Of course, you can reintroduce any needed delay, by including an explicit “ITI” event in the trials of the experiment.) The cost of this technique is that you must precompile trials, which limits your experimental design and takes time at the beginning of each run of the experiment.

Also, preloading stimuli requires that you have enough memory to store all of the unique stimuli in the experiment in memory at once. This is not likely to be a problem for experiments that use repetitive text stimuli; however, it may be a problem if the stimuli are sounds or PICTs, which can occupy a large amount of memory.

Unlike stimuli that are loaded trial-by-trial, preloaded stimuli are never cleared from memory. This means that you may have enough memory to load everything at the beginning of the experiment, but not enough to keep the stimuli around while data is accumulated. It is therefore important to test the memory usage of your experiment by actually running completely, rather than simply compiling it.

Keeping Stimuli in Memory

Stimuli of type **PICT** and **Sound** are extremely memory intensive. A complicated picture or a long and/or high-quality sound can take up a great deal of space in memory. Unfortunately, it also takes a great deal of space on disk, and thus may take a relatively long time to load into memory.

Because of this, the ability to keep PICTs and sounds in memory from one trial to the next is available. If you have enough memory, you might decide to keep a long sound or complicated picture in memory until the end of the experiment, thus only having to wait for it to be loaded once. This can be useful if a small set of sounds will be used in many trials; e.g., if there is a constant set of sounds which will be used for feedback in every trial in the experiment.

Pictures and sounds which are to be kept in memory are denoted individually. **PICT**, **Sound**, and **SoundDesigner** events have a **Feature** attribute, through which you can set the **Keep picture/sound in memory** flag. The picture or sound which is loaded by the event will be kept for use by later events.

The **Keep picture in memory** feature causes the raw picture information that is loaded from disk to be kept from one trial to the next. This information — which could be a bitmap or line-drawing — is separate from the final, scaled bitmap which is created and stored at load time (and is the bitmap actually seen by the subject).

The **Keep sound in memory** feature causes the buffer of sound samples to be retained across trial boundaries. No additional processing has to be performed on a sound buffer.

Note: Keeping sounds or pictures in memory can be very memory intensive and leaves less memory available for other PsyScope processes and for other stimuli. This feature should only be used for those sounds that will be needed in every, or almost every trial.

Loading Stimuli Event by Event

By default, PsyScope loads all of the stimuli for each trial before it begins. In some experiments, however, trials may contain a large number of stimuli, or the stimuli may be particularly memory intensive, so that it is not possible to load all of them — even for a single trial — into memory at once.

In this case, you will need to direct PsyScope to load stimuli event-by-event. You do this by setting the **Load Time** event attribute. This attribute determines when the stimulus for the event will be loaded. If the value is 0, the stimulus is loaded at the beginning of the trial.

If it is non-zero, then PsyScope will schedule the event stimulus to be loaded that many milliseconds in advance of the event's start time.

Once the event ends, the stimulus will be cleared (unless **Keep picture in memory** or **Keep sound in memory** applies; see “Keeping Stimuli in Memory”, p251), making room for other stimuli to be loaded. The advantage of this technique is that it allows you to use memory in the most efficient possible way, permitting the sequential presentation of memory-intensive stimuli. The cost is the time it takes to load each stimulus, which will take place during the trial, just before it is presented.



Chapter 7. User Environment

7.1 Menus Reference

7.1.1 File Menu

New Project... – Creates a new project. See “6.1.1 Using Projects”, p213.

Open.../Switch... – Opens and loads a script or project, closing any that are currently open. Only one project or script can be open at a time, and if a project is open, only one script owned by the project can be loaded.

Close – Closes the current script or project.

New Text File – Opens a text editor window for a new file. See “7.3 The Editor”, p258.

Open Text File... – Opens a text file to edit in an editor window. See “7.3 The Editor”, p258.

Open Selection – Opens a text file named by the current selection in the frontmost editor window. See “7.3 The Editor”, p258.

Edit This Script – Opens a text editor window for the script currently loaded. See “7.3 The Editor”, p258.

Save Script – Saves to disk any changes made to the currently loaded script.

Save Script As... – Saves the current script under a different name and makes the new file the current script file. If the script belongs to a project, the name will also be changed in the project’s list of scripts.

Save a Copy As... – Saves the current script under a different name but does not switch to the new file.

Revert Script – Disposes of any changes made to the current script since it was last saved and then reloads the script from disk.

Quit – Closes the current script and quits PsyScope.

In text editor mode:

Save File – Saves the file in the frontmost Editor window to disk. See also “7.3 The Editor”, p258.

Save File As... – Saves the text file under a different name. Subsequent saves will go to this new file.

Save a Copy As... – Saves the text file under a different name but does not switch to the new file.

Revert File – Disposes of any changes made to the text file since it was last saved and reloads it from disk.

7.1.2 Edit Menu

Most of these menus are only used with the text editor. See also “7.3 The Editor”, p258.

Undo, Copy, Cut, Paste, Clear – These perform the standard operations.

Find..., Find Again..., Selection to Search, Replace, Replace and Find Again, Replace All, Tools – These items are described below in “7.3 The Editor”, p258.

New Object, Get Object... – These items used by the graphic environment; the name will change depending on the type of object that is appropriate to the active window.

Options – This is a submenu for setting various types of options. See “7.6 Options”, p265.

7.1.3 Run Menu

Run – Runs the current experiment.

Practice – Practices the current experiment.

Monitor – Opens the Trial Monitor or brings it to the front. See “6.3 The Trial Monitor”, p238.

Build Run File..., Do Run File... – These items build and load *run files*, precompiled experiments saved to disk. *Run files are not currently supported.*

7.1.4 Utilities Menu

Project Scripts – Adds or removes scripts to the current project and sets the avail-

ability of the experiments in the scripts. See “6.1.1 Using Projects”, p213 and “6.1.1.2 The Scripts Dialog”, p214 below.

Switch Experiment – Changes the current experiment by selecting from the hierarchical list; this is the same as the pop-up menu in the Console (see “7.2 The Console”, p257). If the selected experiment is in another script of the current project, then the current script will be closed and the new script will be loaded.

Evaluate, Evaluate Again – These items are described in “7.4 The Evaluator”, p263.

Text Tools – The items in this submenu are described in “7.3.1 Editor Menu Items”, p258.

Change Log File... – Changes the name and location of the log file. The location of the log file is stored in a “LogFile” entry in the script, so the new log file name and location is saved when the script is saved. See also “6.1.5 The Log File”, p222.

Log Comment – Opens a text window for a comment to be inserted in the log file. Closing the window automatically writes the window’s contents to the log file. See also “6.1.5 The Log File”, p222.

View Log File – Opens a read-only copy of the current log file in the edit window. As the log is changed, this edit window will *not* be updated.

Change Data File... – Changes the name and location of the data file, as stored in the **Data File** attribute of the current experiment. The name and location of the new data file will be saved when the script is saved. No new files are created until the experiment is run. See also “6.1.4 The Data File”, p217.

View Data File – Opens the data file specified in the “DataFile” attribute of the current experiment. If no data file is specified, a dialog will ask for a file to open. See also “6.1.4 The Data File”, p217.

Experiment Notes – Opens an editor window for typing in notes about the current experiment. This uses the same window as the **Notes** button in the Design window (see “5.2.4 Design Window Control Area”, p113), but the menu item can be used for an experiment in any format.

Reinitialize Script – Reinitializes the log file, menu system, and script resources for the currently loaded script. It does not save the script or reload it from the disk.

List Script Changes – Displays a list of all entries that have changed since the script was loaded or last saved.

7.1.5 Design Menu

This menu will be in the menu bar only if **Design** mode is on (see “7.6.1 General Options”, p266).

New Experiment – Creates a new experiment. You will be prompted for an experiment name, and whether the new experiment should be written to the currently loaded script or to a new file.

Clean Up – Available only when the Design window is active; it is used to reformat object icons in the Work area of the Design window. See “5.2.3.1 Cleaning Up”, p113.

Objects in Script – A submenu of all of the types of objects in PsyScope; each of these items opens an Object List dialog (see “5.2.7 Object List Dialog”, p115) containing a list of all of the objects of that type in the script.

Check Links – A utility that is normally run automatically; it analyzes the script and validates the hierarchical structure. **Check Links** updates the “BuilderData” script entry, which is where information specific to the graphic environment is stored. (This information is not needed for running the experiment.) If the Option key is held down while this item is selected, “BuilderData” will be erased and completely rebuilt.

Put in View – Creates a graphic environment object for an arbitrary entry in the script. *This feature is not currently supported.*

Empty Trash – Permanently deletes all of the objects that are currently in the trash. See “5.2.3.2 Trash”, p113.

Show Trash – Opens a window that lists all of the objects currently in the trash. See “5.2.9 View Trash Dialog”, p115.

7.1.6 Script-Specific Menus

A number of script-specific menus may appear between the **Utilities** (or **Design**) menu and **Windows** menu. The operations performed by these items are entirely dependent on their definition in the script. See “Part 4: Scripting Reference, 16.1 Setting up the Menus”, p449.

7.1.7 Windows Menu

Console – Brings the Console to the front. See “7.2 The Console”, p257.

Design – Opens the Design window or brings it to the front. See “5.2 The Design Window”, p107. This item is only available for scripts using Factor format.

Subject Info – Opens the Subject Info window or brings it to the front. See “6.2.1.2 Subject Info Dialog”, p226.

Palettes – Opens or closes the graphic environment tool palette floating window; this item is enabled if a palette-using window is frontmost and the **Palettes in separate floating window** option is on. See also “5.2.1 Objects and the Experiment Hierarchy”, p107.

Trial Chooser – Opens or closes the Trial Chooser floating window for some graphic environment dialogs. See “5.11 Trial Chooser Floating Window”, p209.

Cell Chooser – Opens or closes the Factor Table floating window for some graphic environment dialogs. See “5.7.2.7 Factor Table Floating Window”, p144.

Help... – Gets help on a word or opens a help page. See “7.5 The Help System”, p264.

Monitor – Opens the Trial Monitor or brings it to the front. See “6.3 The Trial Monitor”, p238.

Evaluator – Opens the Evaluator window or bring it to the front. See “7.4 The Evaluator”, p263.

Zoom – Makes current window as large as possible or returns it to the user size. (Selecting this item is just like clicking the zoom icon in the right corner of the window title bar.)

Iconify/Deiconify – **Iconify** replaces the frontmost window with a small, icon-sized version of the window. Double-clicking on the icon or choosing **Deiconify** restores the window to its normal state. See also “7.6.5 Display Options”, p269.

Close – Closes the frontmost window.

Other items at the end of the **Windows** menu are the names of currently open windows; selecting a name brings that window to the front. These windows are typically editor or graphic environment windows.

7.2 The Console

The Console is always open when a script or project is loaded; closing the Console causes the script or project to be closed. If a project is loaded, the title of the Console will be the name of the project; if a script is loaded outside of a project, the title will be “PsyScope”.

The first item in the Console is a pop-up menu of all the experiments available in the current script or project. This menu is just like the **Switch Experiment** submenu in the **Utilities** menu. If you select an experiment from the menu while pressing the Option key, the experiment’s script file will be opened in a text window, without changing the current experiment.

Note: See “6.1.1.2 The Scripts Dialog”, p214 for information on adding experiments to this menu in a project.

At the right end of the Console are **Run**, **Practice**, **Monitor**, **Design**, and **Quit** buttons. These buttons perform the same operations as their menu counterparts. (See “7.1.3 Run Menu”, p254, “7.1.5 Design Menu”, p256, and “7.1.1 File Menu”, p253.)

At the bottom of the Console is the name of the script file that is currently loaded. Clicking on the script name opens the script in an editing window (see “7.3 The Editor”, p258).

In the middle of the Console may be a number of bulleted items. These are the names of some specially configured entries in the script and their current values. You can change the value of the entry by (single-) clicking on the item in the console. In this way, the Console provides both the ability to display a few important values from the script for the user, and allows quick access to the most used entries. (Information on configuring the Console is given in “Part 4: Scripting Reference, 16.2 The Console”, p455.)

7.3 The Editor

The PsyScope text editor can be used to edit any text file. The Editor’s primary purpose is to allow scripts and other experiment files to be edited from within PsyScope, but it is used by PsyScope for several different things: e.g., editing, messages, help files. The *Action bar* at the top of the editor will change (and sometimes disappear) based on the mode in which it is being used.

Except where noted, the information presented here applies to all modes. In general, the editor will perform like any Macintosh editor, with a vertical scroll bar showing the position of the visible section in the file, and a horizontal scroll bar for scrolling across the fixed-width page.

When a text file is opened, the **File** menu items **Save**, **Save As...**, **Save a Copy As...**, and **Revert** change so that they may be used for the text file rather than the script. If the editor is being used in interactive mode (see below), this does not happen because the file and the script are the same.

7.3.1 Editor Menu Items

The **Edit** menu includes these items:

Undo, **Copy**, **Cut**, **Paste**, **Clear**, and **Select All** – Work in the standard way. If you hold the shift key down as you select **Copy** or **Cut**, however, the copied/cut text will be *appended* to the current clipboard, rather than replacing it.

Find... – Opens a dialog to locate a string of characters in the text. See “7.3.5 The

Find Dialog”, p262.

Find Again – Searches using the parameters set in the Find dialog and starting from the caret position or end of the current selection. See “7.3.5 The Find Dialog”, p262.

Selection to Search – Sets the find string to be the current selection. See “7.3.5 The Find Dialog”, p262.

Replace – Replaces the currently selected text with the replace string. See “7.3.5 The Find Dialog”, p262.

Replace and Find Again – Performs the **Replace** and **Find Again** commands consecutively.

Replace All – Finds all occurrences of the find string in the file (after the caret position in non-wraparound mode) and changes them to the replace string. See “7.3.5 The Find Dialog”, p262.

Text Tools, in the **Utilities** menu – A submenu with these items:

Interactive Mode – A shortcut item for setting the **Interactive Script Editing** option (see “7.6.3 Editor Options”, p267). This item is only enabled if the frontmost editor window is for the current script.

Scroll to Changes – A shortcut item for setting the **Scroll to Changes** option (see “7.6.3 Editor Options”, p267). This item is only enabled if the frontmost editor window is for the current script.

Script Enabled – Enables and disables reading from and writing to the script by background processes. This is useful if you need to do a large amount of editing and wish to suspend background updates. This item is only enabled if the frontmost editor window is for the current script.

Wraparound – Enables and disables wraparound text. This option is not used in interactive editing. (This wraparound is unrelated to the **Wrap-around** searching option.)

Balance – Extends the current selection forward and backward until a balanced pair of brackets is found.

Statistics – Opens another window to report the number of lines, words, and characters in the text and in the current selection range.

Count Finds – Counts the number of occurrences of the find string in the text after the cursor or in the selection range. See “7.3.5 The Find Dialog”, p262.

Shift Left – Shifts all lines in the current selection left by removing a tab from the beginning of each line.

Shift Right – Shifts all lines in the current selection right by adding a tab to

the beginning of each line.

Comment Lines – Places a “#” at the beginning of each selected line if there is not one already.

Uncomment Lines – Removes “#” from the beginning of each selected line if it is there.

Uppercase – Converts all alphabetic characters in the selection range to uppercase letters.

Lowercase – Converts all alphabetic characters in the selection range to lowercase letters.

Script Functions, Script Keywords, Exp Keywords, Actions, and Conditions submenus – Choose an item from one of these submenus and it will be pasted into the current selection in the text.

7.3.2 Keyboard Commands

The following special keyboard commands are supported:

- Control-P: previous line
- Control-N: next line
- Control-B: previous character
- Control-F: next character
- Control-A: beginning of line
- Control-E: end of line
- Control-D: delete next character
- Control-K: cut text from cursor to end of line (append to clipboard)
- Control-Y: paste

The Home, End, Page up, and Page down keys are also supported.

The meanings of Command-arrow and Option-arrow depend on the **Cmd-arrow and Option-arrow same as MPW** option setting (see “7.6.3 Editor Options”, p267). If it is off:

- Option-Up: home
- Option-Down: end of file
- Option-Left: beginning of line
- Option-Right: end of line
- Command-Left: start of word
- Command-Right: end of word

If it is on:

- Command-Up: home
- Command-Down: end of file

Command-Left: beginning of line
 Command-Right: end of line
 Option-Left: start of word
 Option-Right: end of word

7.3.3 Action Bar

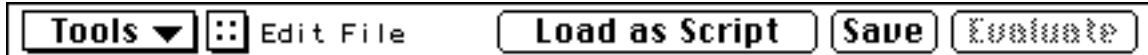


Figure 193 – Editor action bar

The *Action bar* at the top of the editor window is provided for information and convenience. The buttons in the Action bar of an editor window generally perform the same operations as certain menu items. **Load as Script** and **Save And Reload** are provided so that you can edit scripts and quickly load them back in to PsyScope. Editor windows in Help mode have special buttons (see “7.5 The Help System”, p264).

The **Tools** menu at the left of the Action bar is the same as the **Text Tools** submenu in the **Utilities** menu, provided for quicker access.

Clicking on the pop-up with double-colons (::) (next to the **Tools** menu) opens a menu that lists the names of all entry definitions in the text; selecting an item from the menu moves the selection range to the entry’s definition in the text. In addition to entry names, the menu also looks for markers of the format #> marker-name; a marker is indicated in the menu with a > beside the name.

By default, entry names in the :: pop-up menu are listed in the order in which they are found in the script, but the **Sort entry list alphabetically** option (see “7.6.3 Editor Options”, p267) alphabetizes the names, putting marker names at the beginning. This option can be toggled by selecting the last item in the pop-up menu, **Alphabetize**. It can be toggled temporarily by holding down the Option key while clicking on the :: pop-up.

7.3.4 Interactive Mode

Interactive mode is a way of editing the current script so that the text you see in the editor window is a live copy of the script being used: when you make a change to the text, the script is change parsed immediately, and when an entry’s value changes as a result of some other action (e.g., a dialog is used in the graphic environment), the changes are immediately reflected in the text.

Interactive mode is enabled through the Editor options dialog, or by the **Interactive Mode** item in the **Tools** menu.

If the name of an include file is changed in an interactive script, the new file will be loaded the next time the script interpreter searches that part of the script for an entry. The entries that were in the old included file will be forgotten.

Because you are modifying the working copy of the script when in interactive mode, background processes within the PsyScope environment are temporarily suspended while you are typing. The suspension continues until a certain time after your last keystroke. The duration of this suspension is set through the **Script editing suspend time** option (see “7.6.3 Editor Options”, p267).

If you misedit an entry (e.g. forget to close a pair of quotes) and the entry is used by some other open window (such as the Design window), PsyScope may get into a state where it is trying to update the entry and constantly gives an error message. In this case, hit the **Shut Up** button in the error dialog; you will have about five seconds to fix the script (before the dialog shows up again).

7.3.5 The Find Dialog



Figure 194 – The Find dialog

The Find dialog has two fields: one for the *find string* and one for the *replace string*. These strings are used by the buttons at the bottom of this dialog, and are also remembered for use by the other **Find** menu items.

The Find dialog lets you set a number of options that control how a search is performed:

If **Ignore case** is checked, text is found when it matches the letters of the find string, even if the capitalization is different.

If **Match words** is checked, text matches the find string only if it is found as a separate word or phrase, i.e. preceded and followed by spaces.

If **Wraparound** is checked and a search reaches the end of the file, searching continues again from the start of the text file

Like the find and replace strings, these search modes are also remembered. The modes can also be changed through the Editor options dialog (see “7.6.3 Editor Options”, p267).

Hitting the **Search** button begins the search from the caret position or end of the current selection in the frontmost editor window. The rest of the buttons perform the same function as their menu counterparts.

The **Select editor window after finding or replacing** Editor option controls whether hitting any of the buttons moves the searched-in editor window to the front, or keeps the Find dialog active (see “7.6.3 Editor Options”, p267).

The find and replace strings are limited to 255 characters. Tabs and returns may be specified using “\t” and “\r”, respectively. “\\” must be used for “\”.

7.4 The Evaluator

The Evaluator is a special editor window that is in direct communication with the PsyScript interpreter. You can use the Evaluator to type in a scripting expression and evaluate it immediately. The result is returned on a line starting with # `Result:` followed by the value of the expression.

An expression is evaluated when the **Evaluate** menu item is selected (or Command-Space is hit). The expression to be evaluated is determined one of two ways:

- 1) If a range of text is selected, the selected text is used as the expression. This method will work in an arbitrary text editing window; the Evaluator will be opened or brought to the front to return the value of the expression.
- 2) If no range is selected, the current position of the caret is taken to be the end of the expression; the starting position is the first character after the last previous commented line. This method works *only* in the Evaluator. (Note that the Evaluator returns results in a comment format. This means you can type an expression, hit Command-Space for the result, type another expression, etc.)

The **Evaluate Again** menu item re-evaluates the expression that was last evaluated.

New entries can be defined through the Evaluator. To do this, just evaluate a valid entry definition; the evaluator will know that it is supposed to create a new entry because of the double colons (“::”) in the text. Only one entry may be defined at a time in this way.

If you try to define an entry with a name that is already used in the script, an error is returned. If an entry by the name had been defined previously in the Evaluator, its definition is replaced.

Entries defined through the Evaluator are *not* included in the script. Changing scripts or re-loading will erase the entries. Reinitializing the script will not erase the entries.

If an expression is evaluated containing the `THIS` operator (see “12.2.4.1 `THIS` and `OWNER`”, p324), the operator will refer to a temporary entry that is created in order to evaluate the expression. This temporary entry is named “TempEntry”. This convention does not, of course, apply when a new entry is being defined.

7.5 The Help System

The Help system is made up of a number of different *sections* corresponding to different parts of PsyScope. A single section can be opened in any one text editing window — called a *page* — but any number of sections and any number of copies of a section can be opened by using multiple pages.

Help can be opened three ways: the main section can be opened by hitting the Help key on an extended keyboard; a specific section can be opened through the **Help** submenu in the **Apple** menu, or a dialog that asks for a keyword on which to find help can be obtained by choosing **Help** from the **Windows** menu (or hitting Command-H).

When a help page is opened, the text editor Action bar has a special configuration. At the left is a menu containing a list of all the sections; choosing one of these changes the section displayed by the page. If Option is held down as a section is chosen, a new page will be opened. The Action bar buttons will be discussed further below.

7.5.1 The Help Search Dialog

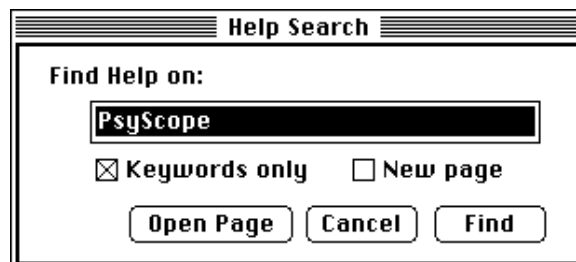


Figure 195 – The Help Search dialog

The usual Find dialog (see “7.3.5 The Find Dialog”, p262) can be used to locate information within a page. To locate help on a topic that may be anywhere in the help system, use the Help Search dialog, which is opened by selecting **Help** from the **Windows** menu. A string typed here will be searched for in the help index to find help on the topic.

If **Keywords only** is checked, the help search will only use items that are specifically listed in the help index. Otherwise, all of the text in all Help sections is searched.

If **New page** is checked, a new page will be opened to view the findings. Otherwise, the frontmost open Help window will be used. (If no Help page is already open, a new one will always be opened.)

If the **Open Page** button is hit instead of **Find**, no search will take place; a new page will simply be opened showing the main Help section.

If you choose **Help** (or hit Command-H) with an active selection range in any editor window, the text in the range will be automatically transferred to the search text box in the Help Search dialog. If you hold the Shift key down at the same time, the dialog will be skipped altogether.

7.5.2 Help Action bar Buttons

Often, more than one reference will be found for a topic. The help finding system gives exact keyword matches the highest priority, then inexact matches on keywords, then matches found in the general text (if the **Keywords only** is not checked).

Multiple findings can be viewed by hitting the **Next** button at the top of a help page, or by typing Return, Enter, or N. If the **Next** button is disabled, no more references were found. You can go back to previous findings by hitting the **Preu** button or by typing P.

If you hit the **Help On Selection** button, the Help system takes the text in the current selection range and attempts to find other help on it. This is the same as copying the selected text and pasting it into the text box of the Help Search dialog.

7.6 Options

PsyScope's options are divided up into five dialogs: General options, Run options, Editor options, Design options, and Display options. Options are set by choosing an item from **Options** submenu at the bottom of the **Edit** menu.

Options are stored in a "PsyScope Preferences" file in the "Preferences" folder in the System folder (or in the System folder itself in pre-System 7.0). If a preferences file is in the same directory as PsyScope, that one will be used instead. If a project is open, most options are stored in the project instead, so that each project has its own set of options. The **Auto-load this script/project** option is always stored in the preferences file.

Hitting the **Save** button in any of the options dialogs stores the options for that dialog. Selecting **Save as Default** from the **Options** submenu stores the current set options specially, so that newly created projects will start with these options.

7.6.1 General Options

User mode – If **Design** is selected, modifications may be made to the script through the graphic environment; if **Run Only** is selected, the graphic environment will not be available.

Autoload this script/project – This sets the script or project currently loaded to be automatically loaded every time PsyScope starts up. (If another script or project is double-clicked to start PsyScope, this option will be ignored.)

Autosave – If this option is active, the script will be automatically saved whenever a change is made to the script. If the **immediately** mode is selected, changes will be saved in the background. If **before run or quit** mode is selected, the changes will only be saved just before running or quitting the experiment.

Click in inactive window used as if active – If this option is active, clicking in an inactive window will bring the window to the front, and then the click will count again as a click in the content of the window.

TimeBar operations backgroundable – Some time-consuming jobs performed by PsyScope use a time bar show to report progress to the user. If this option is on, these jobs can be performed by PsyScope in the background. This is more convenient if you are performing multiple tasks concurrently, but it will slow down the job.

Warning Messages – This option allows the user to specify a “safety level” within PsyScope. If **All** is selected, the user will get all warning messages. If **Most** is selected, messages and warnings with the face icon will not be given. If **Some** is selected, most warnings will not be given, except those which use the extra-bold exclamation point icon, or which require a user decision. If **Very Few** is selected, then only warnings which require a decision will be given. Error messages (indicated by stop sign or bomb icon) are always given. Unless you have a reason to skip certain error messages, you should use the **All** mode.

Store path names relative to... – This option is used only for projects. It allows you to specify the starting point of path names (relative to **Project**, **Script**, or **PsyScope**) for any file referenced anywhere in PsyScope while the project is opened. **Require Full Path** always generates a path name starting with the volume name. Scripts which are not in a project always have path names relative to the script. See “6.1.2.1 Relative Paths”, p215.

7.6.2 Run Options

Run File Default – This options specifies the default name and file format to be used when writing a run file. The name and format can be changed by the user when the run file is being stored. *Run files are not currently supported.*

Use experiment info from file – This option specifies, when running a run file as an experiment, whether such information as the log file and data file specifications should be read from the run file, or reread from the current experiment in the open script. If no script

is loaded, the experiment information is always read from the run file. *Run files are not currently supported.*

Use run file directory as default – This option determines, when a run file is opened for use as an experiment, whether the default volume and directory for locating files should be the open scripts, or the directory in which the run file was located. *Run files are not currently supported.*

Trial monitor count separate from script – If this option is set, the trial count in the Trial Monitor will be stored separate from, and will therefore possibly be different from, the trial count attributes of the experiment in the script.

Show number to be compiled in run time bar – When an experiment is run and the compile time bar is shown, the number of trials that are being compiled will be displayed unless this option is off.

7.6.3 Editor Options

Interactive Script Editing – This option enables live editing of the script currently loaded. See also “7.3.4 Interactive Mode”, p261.

Select editor window after finding or replacing – This option causes the frontmost editor window to be automatically moved in front of the Find dialog when one of the search or replace buttons is hit. See also “7.3.5 The Find Dialog”, p262.

Scroll to Changes – When an entry changes and an interactive editing window is open, the window will scroll to the changed entry and hilite it.

Wraparound – This option is used for searching; when the end of the file is reached and this option is on, searching continues again at the beginning of the text.

Match Words – This option is used for searching; if this option is on, text will be considered a match with the find string only if it is preceded and followed by spaces, tabs, or returns.

Ignore Case – This option is used for searching; if this option is on, upper- and lower-case representations of a letter are considered equal for searching.

Auto tab for newlines – When a Return is typed into a text file and this option is on, tabs and spaces are automatically placed at the start of the new line to match the indentation of the previous line.

Sort entry list alphabetically – This option causes the list of entries obtained by clicking on the :: pop-up in the editor window to be sorted alphabetically, with markers at the top of the list. If this option is off, the items are in the order in which they appear in the script.

Don't wrap text (newlines with returns only) – This option enables wraparound text (i.e. flowing text that runs off the right end of the page back to the left end) in the editor window. An interactive editing window will never use wraparound.

Cmd-arrow and Option-arrow same as MPW – This option controls how Command-arrow and Option-arrow navigate in the text editor. See “7.3.2 Keyboard Commands”, p260.

Script editing suspend time – This option sets the pause time after the user stops typing in an interactive window; during this pause, script reading and writing is suspended for background processes. See also “7.3.4 Interactive Mode”, p261.

7.6.4 Design Options

Automatically open Experiment View at startup – If this option is on, PsyScope opens the Design window automatically whenever a script is loaded.

Always ask for new object names – If this option is on, PsyScope always requests a name for any new object created. If this option is off, a default object name will be usually generated.

Use horizontal arrangement for structure display – This option is for the Work area of the Design window: in horizontal arrangement, the experiment hierarchy proceeds left to right; in vertical arrangement, the hierarchy proceeds top to bottom.

Trash always in lower right corner – This option makes the trash can always float in the lower right corner of the Design window. Otherwise, the trash can will be a moveable object like any other in the window.

Palettes in separate floating window – This option moves the palettes out of the Design and Template windows and into a separate floating window. See “5.2.2 Design Window Palettes”, p109.

Auto-change to link tool after object creation – If this option is on, PsyScope automatically changes the tool to the link tool after an object is created in the Design Window. See “5.2.2 Design Window Palettes”, p109.

Template time step – This option specifies the minimum granularity used for duration and start times in the Template window.

Name Change Pause – When the name field of a dialog is edited, this specifies the time (in seconds) that PsyScope should wait before actually changing the object's name. This lets you to finish typing in the name text box before the name is changed. This pause also applies in the Factor Table window (but not the Cell Chooser) for new cell selections. The recommended value is 2 to 4 seconds.

7.6.5 Display Options

Remember new window positions – When this option is on, the position of a window is remembered when it is closed so that it may be put back at the same place when it is opened. Turning off the option does not return the windows to their default positions, it only stops remembering new positions for the windows.

Animate startup window (in 256+ colors only) – This option controls whether or not the color-cycling animation is used when PsyScope is starting up. You may want to turn the animation off if it flashes your screen unnecessarily (updating background color) while starting up.

Iconify to mini-window instead of Finder-style – This option controls how iconified windows are displayed. The mini-window style may be easier to see on a cluttered screen.

7.6.6 Custom Options

This is a submenu for script-specified options. See “Part 4: Scripting Reference, 16.3 Custom Options”, p456 for information on configuring this submenu.

Part 3:

Scripting

User Manual

Chapter 8. Introduction 273

Chapter 9. Scripting Overview 275

Chapter 10. Scripting an Experiment 281



Chapter 8. Introduction

The purpose of this part of the documentation is to describe how the scripting language is used to define an experiment. A little knowledge of scripting is needed, so a little scripting starter information is presented in “Chapter 9. Scripting Overview”.

It is assumed that the reader is familiar with “Part 1: Introduction to PsyScope” and “Part 2: Graphic Environment Reference”; many details about the use of trials and events are left for those parts, while this part is concerned with expression those constructs in the language of PsyScript.

“Part 4: Scripting Reference” contains an exhaustive reference for PsyScript, experiment descriptions, and user environment configuration.

Ψ

Chapter 9. Scripting Overview

This chapter should give you enough information about PsyScript for you to understand “Chapter 10. Scripting an Experiment”. PsyScript is explained in full in “Chapter 12. PsyScript Reference”.

9.1 A PsyScope Script

A *script* is a text file used to define a PsyScope experiment. The syntax of the script is called *PsyScript*. A script consists mostly of *entries*; they are the basic blocks of information in a script, similar to the records of a database. (For example: each object in the graphic environment is implemented as a separate entry in the script.)

Here is an example of a typical entry:

```
Experiments:: "Hello World"  
  Current: 1
```

A script can also contain *comments*, *modifiers*, and *section markers*. These elements are used to annotate the script in different ways. (Modifiers and sections markers are not directly addressed in this chapter; see “Chapter 12. PsyScript Reference”.)

9.2 Entries

An entry has three parts: an *entry name* (shown as part A in “Figure 196 – The components of an entry”, below), a *content* (part B), and an *attribute block* (part C).

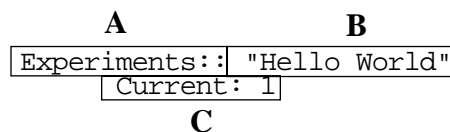


Figure 196 – The components of an entry

9.2.1 Entry Name

Part **A** in Figure 196 is the *entry name*; in the example above, the entry name is “Experiments”. An entry always starts with an entry name followed by double colons. The entry name can be just about anything, but there are a few restrictions:

- Entry names cannot include any colons (“:”).
- Entry names cannot include any greater-than symbols (“>”).
- Entry names *can* include spaces.
- Each entry name in the script should start on its own line.
- An entry name cannot be the same as the name of an input device or action.

9.2.2 Content

Part **B** in Figure 196 is the entry’s *content*, which always starts immediately after the double colons. Content information is important for certain types of entries, such as the “Experiments” entry, but an entry’s content will often be empty.

The content of an entry consists of a list of *expressions*. An expression is sometimes just a word or a quoted phrase. In this example, there is only one expression: “Hello World”.

The content of an entry can be arbitrarily long, and it does not have to be on a single line.

9.2.3 Attribute Blocks

Part **C** in Figure 196 is the *attribute block*. (This use of the word “block” is unrelated to the concept of an experimental block.) The attribute block is made up of one or more individual *attributes*, where each attribute is defined by a name followed by a single colon. (These attributes *are* directly related to “attributes” of an object in the graphic environment.) The entry in Figure 196 has one attribute: “Current”. An entry is not required to have an attribute block.

The relationship of attributes to entries is analogous to the relationship of entries to scripts; just as entries label and distribute information within a script, attributes label and distribute information within an entry.

Attributes are also entries in that they have a *content* and can own *sub-attributes*. Expressions following the single-colon of an entry are in the *content* of the attribute. Ultimately, all information in a script is stored in the content of an entry or an attribute. In Figure 196, the content of the “Current” attribute has one expression: 1.

Sub-attributes are defined by following the sub-attribute name with “:””, as in this example:

```
AnEntry:: a b c
  1stAttrib: 1 2 3
    SubAttribOf1st:> x
  2ndAttrib:
    FirstSubAttribOf2nd:> 4
    SecndSubAttribOf2nd:> 5
    SubSubAttribOfSecondSubAttribOf2nd:>> "Deep!"
```

Sub-sub-attributes are defined using “:>””, and so on, adding a “>” for each deeper layer of attributes. In practice, sub-attributes are used only occasionally; sub-sub-attributes and deeper layers are rarely used in normal scripting.

9.2.4 How Entries Are Used

The sample entry below defines an event that displays the string “Hello World” on the screen in the twelve-point Geneva font:

```
Hello Event::
  Stimulus: "Hello World"
  EventType: Text
  Font: Geneva
  Size: 12
```

The “Stimulus” attribute of the entry tells PsyScope that the stimulus will consist of the string “Hello World”. The “EventType” attribute tells PsyScope that this is a text event — that is, that the event will cause text to be displayed on the screen (as opposed to displaying pictures or playing sounds). The “Font” and “Size” attributes tell PsyScope to display the text in 12-point Geneva font. Other attributes could be included to further specify the features of the stimulus, such as its color, its duration, etc.

9.2.5 Spaces, Blanks, and Quotes

In the script, expressions are separated by *blanks*. A blank can be

- a regular space
- a tab
- a comma
- a return

When an expression contains one or more blanks, the string must be enclosed by quotation marks. We had to put quotation marks around “Juliet Capulet” because we wanted all the words to be treated together as a single item. Without quotes:

```
Hello Event::  
  Stimulus: Hello World  
  EventType: Text  
  Font: Geneva  
  Size: 12
```

the “Stimulus” attribute would have two values: “Hello” and “World”. In this example, PsyScope will display an error message since Text events can have only one stimulus.

You never have to put quotes around an entry name in the entry’s definition, even if it contains blanks. If an attribute name has blanks, however, you *do* need quotes. Here is an example of an entry with blanks in its name and some attribute names:

```
Hello Event::  
  Stimulus: "Hello World"  
  EventType: Text  
  Font: Geneva  
  Size: 12  
  "Extra Attribute": TRUE
```

Note that all of the quotes are straight quotes; *never* use “smart quotes” in a script. “Smart quotes” are those curly quotes you see here: “ ”; straight quotes look like this: " ".

The following table summarizes the rules for quotes that apply to the different parts of an entry:

Table 2: Quoting conventions

	Name Contains Blanks	No Blanks
Entry name in definition	Don’t quote	Don’t quote
Entry name elsewhere	Quote	Don’t quote
Attribute name	Quote	Don’t quote
Content expression	Quote	Don’t quote

9.3 Comments

PsyScope allows you to place comments in your script by using a pound sign (“#”); PsyScope will ignore everything after a “#” until the end of the line is reached. For example:

```
# This is an event definition entry  
Hello Event::  
  Stimulus: "Hello World"
```

```

EventType: Text
# Use 12-pt Geneva
Font: Geneva
Size: 12

```

9.4 Entry References

A *reference* is a “pointer” to an entry. A reference is typically used to perform some operation on the entry or to obtain some value from the entry. The (full) name of an entry in the script can always be used as a reference, but there are also other, more complicated ways of generating a reference.

The most common use of a reference is to get a value stored in the entry; the basic operator is @, which operates on a reference and returns the content of the referenced entry.

For example, given this entry definition in the script:

```

Hello Event:: "useless value"
Stimulus: "Hello World"
EventType: Text
Font: Geneva
Size: 12
"Extra Attribute": 1 2 3

```

Then, the value of @"Hello Event" is "useless value".

To make a reference to an attribute, you concatenate the entry name, the >> symbol, and the attribute name. For example, "Hello Event>>Stimulus" references the “Stimulus” attribute of “Hello Event”. The value of @"Hello Event>>Stimulus" is "Hello World".

The @ operator returns all of the tokens in the content of an entry, so that the value of @"Hello Event>>Extra Attribute" is three tokens: 1, 2, and 3.

9.5 Lists

A *list* is nothing more than an entry that is used in a special context. The values in the content of the entry become the *items* of the list; an entry is used as a list when items need to be selected one at a time.

When an entry is used as a list, there are a few attributes of the list that are given a special meaning; the most important one is “AccessType”. “AccessType” specifies the order in which items are selected from the list; its value is usually one of:

Sequential – items are selected from the list in the same order that they appear in the content of the entry

Random – items are selected from the list without regard to their order in the entry’s content; however, each item will be used only once

The process of selecting an item from the list is called *accessing* the list. If all of the items have been selected once from a list, the accessing the list again will start the selection process fresh, creating a new ordering of items if the value of “AccessType” is Random.

Lists are described in full detail in “Part 4: Scripting Reference, 12.8 Lists”, p329.

9.6 Function Calls

There are many kinds of expressions that can appear in the content of an entry. For the purposes of scripting a Factor format experiment, you only need to use literal string expressions (such as "Hello World") and *function calls*.

A function call has the form *FunctionName(Param1, Param2, ... ParamN)*. There cannot be any spaces between *FunctionName* and the opening parenthesis. The commas separating the parameters are not actually required, but they may reading the script easier.

Here is an example of an entry using a function call to `FactorAttrib()`:

```
Hello Event::  
  Stimulus: FactorAttrib(Language, HelloWorld)  
  EventType: Text  
  Font: Geneva  
  Size: 12
```



Chapter 10. Scripting an Experiment

10.1 Scripting a New Experiment

In practice, most PsyScope experiment designers will set up a basic design using the graphic environment, and then use PsyScript to fine tune the details of the experiment. In this chapter, however, we will demonstrate scripting without using the graphic environment at all.

10.1.1 The Standard Script Template

The easiest way to create a new experiment— even if you plan to implement the experiment by scripting — is to select **New Experiment** from the **Design** menu. When you do this, PsyScope creates a new text file that is the skeleton of a script.

Let's create a new experiment — called “Hello World” — so we can look at the resulting script file, line-by-line.

1. Select **New Experiment** from the **Design** menu. This will open a dialog for naming the new script.
2. Name the new experiment “Hello World”. Leave the **New File** box checked if it is present.
3. Save the script as “Hello World Script”, in whatever folder you like.
4. The new script will be automatically loaded into PsyScope. Open a text copy of the script by selecting **Edit This Script** from the **File** menu.

Looking at a script created by **New Experiment**, we see:

```
#PsyScope 1.0  
# Script template, Version 1.0
```

A script always starts with the modifier `#PsyScope 1.0`. This lets PsyScope know that this text file is a script. See also “12.4.1 #PsyScope”, p324.

`# Script template, Version 1.0` is a comment indicating the version of PsyScope that originally created this script. Of course, since it is only a comment, this line is not necessary for running the script.

```
Experiments:: "Hello World"  
  Current: 1
```

This is the first entry in the script, the “Experiments” entry. This entry contains the names of all experiments defined within the script (a script can implement multiple experiments, although one script usually corresponds to one experiment). PsyScope just created this file for the experiment “Hello World”, so that is all that is initially in the “Experiments” entry.

Note that the “Current” attribute of the “Experiments” entry is set to 1. Whenever a script is loaded, one experiment in the script is designated as the *current experiment*; this is the experiment that is executed when the **Run** button is hit in the Console, or modified when the **Design** button is hit, etc. The value of “Current” is an index into the list of experiments in the “Experiments” entry. Since there is only one experiment in this new script, 1 is the only possible value for “Current”.

```
#> ExperimentDefinitions  
  
Hello World::  
  Format: Factor  
  InputDevices: MOUSE KEY  
  Timer: Macintosh  
  Flags: NO_SAVE_SCREEN  
  DataFile: "Hello World Data"  
  ScaleBlocks: 1
```

For each experiment implemented in a script, there must be an entry — an *experiment entry* — with the same name as the experiment. The experiment entry is the starting point for the definition of an experiment in PsyScript; it corresponds exactly to the psy icon that represents the experiment in the graphic environment’s Design window (see “Part 2: Graphic Environment Reference, 5.2 The Design Window”, p107).

The #> ExperimentDefinitions marker simply indicates that this is the section for defining experiment entries. Like all markers, it is for informational purposes only, and does not affect how the script is read to execute the experiment.

The “Hello World” entry is set up with a number of default attribute values, including the attributes “Format” (Factor format), “InputDevices” (the keyboard and mouse), “Flags” (do not try to save the screen on breaks), and “DataFile” (“Hello World Script.data”, in the same directory as the script).

```
#> GroupDefinitions  
  
#> BlockDefinitions  
  
#> FactorDefinitions  
  
#> LevelDefinitions  
  
#> TemplateDefinitions  
  
#> EventDefinitions  
  
#> StimulusDefinitions  
  
#> TrialManagerVariables
```

#> Group Definitions, #> BlockDefinitions, etc. are more markers (i.e., guidelines for placing entry definitions within the script). The graphic environment uses these guidelines, and the interactive Editor's :: pop-up menu displays them (see "Part 2: Graphic Environment Reference, 7.3.3 Action Bar", p261), but they are otherwise treated as comments.

```
#> PortDefinitions
PortNames:: "Entire Screen"
Entire Screen:: Center 100% Center 100% 0
#> PositionDefinitions
```

The #> PortDefinitions section includes a "PortNames" entry and one port definition: "Entire Screen". This information is used by the Positions extension, which is the part of the graphic environment that interactively defines new ports for screen events. These entries can be deleted if you have no use for the Positions extension; however, if the Positions extension is executed, it will re-create "PortNames" and "Entire Screen". The #> PositionDefinitions section is also used by the Positions extension, but there are no default positions.

```
#> SubjectInfo
SubjectName:: "SUBJECT NAME"
SubjectNumber:: 1
    Type: Integer
RunNumber:: 1
    Type: Integer
```

The SubjectInfo section defines some standard subject-tracking items. See "Part 2: Graphic Environment Reference, 6.2 Subject Info", p224 for more information.

```
#> InterfaceDefinition
Console::
Options::
#> MenuDefinitions
Menus:: Experiment
Experiment::
    @StandardPsyScopeMenuItems
#> DialogDefinitions
```

#> InterfaceDefinition, etc. contain entries which are used to customized the PsyScope environment for this script. See "Chapter 16. Configuring the User Environment", p449.

```
#> LogFile
LogRunStart:: SubjectName
    Dialog: LogInfo
```

```
#> OtherConfiguration
RunStart:: LogRunStart
```

The `LogFile` and `OtherConfiguration` sections contain entries which help with subject-tracking. See “Part 4: Scripting Reference, 16.6.2 Execution Entries”, p460.

```
#> ExperimentNotes
Notes::
  "Hello World": "(There are no notes for this experiment.)"
```

The `#> ExperimentNotes` section contains the “Notes” entry. This entry is used by the graphic environment to store notes about the current experiment (when the **Notes** button is hit from the Design window; see “Part 2: Graphic Environment Reference, 5.2.4 Design Window Control Area”, p113).

```
#> BuilderData
# This information is used by the graphic interface - do not delete

BuilderData::
  "Hello World":
  GroupsInScript:
  BlocksInScript:
  TemplatesInScript:
  FactorsInScript:
  LevelsInScript:
  EventsInScript:
  StimuliInScript:
  Trash:
  Desk:
  SubjectInfo: SubjectName DataTypes Standard ;
               SubjectNumber DataTypes Value ;
               RunNumber DataTypes Value ;
```

The `BuilderData` section contains the “BuilderData” entry. This information is used by the graphic environment only, and is not intended for human consumption (or modification).

10.1.2 Using the Interactive Editor

PsyScope contains a built-in text editor for modifying scripts. This editor is special: when the current script is being edited, the changes take effect immediately within PsyScope; there is no need to save the file and then re-load it into PsyScope, because your changes are incorporated into PsyScope’s working copy of the script as you type.

You can open a copy of the current script for editing by selecting **Edit This Script** from the **File** menu (see “Part 2: Graphic Environment Reference, 7.1.1 File Menu”, p253). You should have interactive editing enabled, by checking **Interactive Script Editing** in the Editor Options dialog. Also, the **Script editing suspend time** option in Editor Options controls how long PsyScope waits after your last click or keypress before reading your edits.

Interactive editing can sometimes cause problems. For example, suppose you edit the experiment entry and accidentally leave out a closing quote. After the prescribed delay, Psy-

Scope will try to incorporate your new experiment entry into its current picture of the script; since a quote is missing, however, the script interpreter will complain. It may keep trying to read the script, and keep complaining. For this reason, there is a **Shut Up** button in most error dialogs which temporarily suspends error messages (for 5 seconds), giving you time to correct the script. See “Part 2: Graphic Environment Reference, 7.3.4 Interactive Mode”, p261 for more information.

PsyScope’s text editor contains many other features which can make scripting easier. Most of these features are accessed through the **Text Tools** submenu of the **Utilities** menu, described in “Part 2: Graphic Environment Reference, 7.3.1 Editor Menu Items”, p258. (This menu is also available as the **Tools** pop-up menu in the editor’s Action bar; see “Part 2: Graphic Environment Reference, 7.3.3 Action Bar”, p261.)

10.1.3 Scripting a New Event

If you create a new script using **New Experiment**, open it up, and hit the **Run** button in the Console, you will get this error: “‘Events’ attribute or some other link missing in trial definition.” This error is reported because no events are defined in the experiment skeleton created by **New Experiment**.

Let’s define an event that prints the words “Hello World” on the screen for two seconds. Like any object in an experiment, an event is defined by creating an entry and adding attributes.

If you have not already done so, load “Hello World Script” into PsyScope and open the script in an editor window. Be sure that you are in interactive editing by checking the **Interactive Script Editing** box of the Editor Options dialog.

1. Scroll down to the section of the script marked #> `EventDefinitions`. Put the cursor at the end of this line and hit Return a couple of times to make space for a new entry.
2. Type in this entry:

```

Hello Event::
    EventType: Text
    Stimulus: "Hello World"
    Duration: 2000

```

Be sure that you put *two* colons after “Hello Event” and *one* colon after “Stimulus” and “Duration”.

The “EventType” attribute specifies what kind of event is defined by the entry. If “EventType” is omitted, `Text` is assumed. All built-in event types are described in “Part 4: Scripting Reference, 14.2 Stimulus Types Reference”, p424. More event types can be made available by using PsyScope Extensions (see “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216).

Almost every event entry will have a “Stimulus” or “Stimuli” attribute. “Stimuli” is used for multi-part events with sub-stimuli. `Null` (**Time**, in the graphic environment) events do

not require a stimulus. For all other event types, a “Stimulus” attribute must be specified. For text events, the value of the “Stimulus” attribute is the text that should be displayed.

The “Duration” attribute here specifies 2000 milliseconds as the event’s duration. More complicated durations (e.g. waiting for a mouse click) are discussed below (see “10.1.3.2 The ‘Duration’ Attribute”, p287).

At this point, we have a valid event definition, but the experiment entry does not yet “know” about the event entry. Events are linked to their parents with the “Events” attribute.

Add “Hello Event” to the “Hello World” experiment entry:

1. Add a new line at the end of the “Hello World” entry (after ScaleBlocks : 1).
2. Type this attribute definition:

```
Events: "Hello Event"
```

Your experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Events: "Hello Event"
```

Now that “Hello Event” is in the list of events for “Hello World”, try running the experiment. The screen should go blank, the words “Hello World” should appear on the screen for two seconds, and then the PsyScope environment should return.

Usually, you will want to run more than one trial in an experiment. In a design without blocks, the number of trials to be run is controlled by the “Cycles” experiment attribute.

Change “Hello World” to run five trials (i.e. display “Hello World” five times in each run):

1. Add a new line to the “Hello Event” entry.
2. Type this attribute definition:

```
Cycles: 5
```

Your “Hello World” entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Events: "Hello Event"
  Cycles: 5
```

When you run now, you will see “Hello World” for a total of ten seconds, flashing briefly every two seconds as it is cleared and redrawn.

There are many attributes which can be specified for a `Text` event, controlling features of the stimulus such as text font and color; the complete list of attribute names — and the format of their values — is given in “Part 4: Scripting Reference, 14.2 Stimulus Types Reference”, p424.

You should be able to open the Design window of the graphic interface at any time, even when you are changing the script through scripting. If you open the Design window now, you will see the experiment icon with one text event connected to it. (You must have the **Show Events** checkbox on to see the event.)

10.1.3.1 Timing and Sequencing Events

Let’s add another event to the “Hello World” script, so that “Goodbye World” follows “Hello World” in each trial.

Add a “Goodbye Event” event entry:

1. Add a couple of lines to the script after “Hello Event” in the `#> EventDefinitions` section of the script.
2. Insert this entry definition:

```
Goodbye Event::
  EventType: Text
  Stimulus: "Goodbye World"
  Duration: 2000
  Color: Red
```

Just for fun (and example), we have made “Goodbye World” print in red instead of black. Now, we’ll link “Goodbye Event” to the experiment.

3. Add “Goodbye Event” after “Hello Event” in the “Hello World” experiment entry. Your experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Events: "Hello Event" "Goodbye Event"
  Cycles: 5
```

When you run the experiment now, “Hello World” (in black) will be followed in every trial by “Goodbye World” (in red).

10.1.3.2 The ‘Duration’ Attribute

The duration of an event can be specified in four ways, by setting the value of “Duration” to:

- a duration in milliseconds; the event will last a fixed amount of time.
- the keyword `TRIAL_END`; the event will end when all other events in the trial have ended.
- the keyword `SELF_TERMINATE`, for `SoundLabel` events; the event will end when the sound is finished playing.
- one or more condition specifications, plus an optional `Time[msec]`; the event will end when one of the given conditions is triggered.

Of these possible duration types, the first and last are the most common. `TRIAL_END` is useful only when events are running in parallel. `SELF_TERMINATE` is almost always used when the event is a `SoundLabel` stimulus.

For condition-based durations, you specify a list of input devices which can terminate the event; then, for each device, you give parameters which specify the kind of input that is interesting.

Make “Hello World Event” last until the mouse is clicked, instead of a fixed two seconds:

- Replace 2000 in the “Duration” attribute of “Hello Event” with `Mouse[Click]`. Make sure there are no spaces between `Mouse` and `[`, but spaces within the brackets are OK.

Your “Hello Event” entry should now look like this:

```
Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Duration: Mouse[Click]
```

If you run the script now, “Hello World” will not go away until you click the mouse button. “Goodbye World” will still appear for two seconds, whether you click the button or not.

The format of each condition in the “Duration” attribute is: `DeviceName[params]`, where `DeviceName` is the input device and `params` specify one or more states of the input device. There cannot be any spaces between the device name and the opening square-bracket (`[`). The value `Mouse[Click]` specifies the standard Macintosh mouse as the input device and the state is with the button clicked.

Any number of input devices and states can be specified for each event. The event will end when any one of the conditions occurs. In addition to the standard input devices, you can use the “virtual device” `Time` to specify a timeout value on the conditions.

Change “Goodbye World” to end when the mouse is clicked, when any key on the keyboard is hit, or when two seconds pass.

- Replace 2000 in the “Duration” attribute of “Goodbye Event” with `Mouse[Click] Key[Any] Time[2000]`.

Your “Goodbye Event” entry should now look like this:

```
Goodbye Event::
  EventType: Text
  Stimulus: "Goodbye World"
  Duration: Mouse[Click] Key[Any]
            Time[2000]
  Color: Red
```

If you run the script now, “Goodbye World” will still go away after two seconds, but it will go away immediately if you click the mouse button or hit a key.

Event durations are discussed further in “Part 4: Scripting Reference, 13.4.3 Duration”, p411. The list of all condition devices and a description of their parameters is in “Part 4: Scripting Reference, 14.3 Conditions and Inputs”, p437.

10.1.3.3 The ‘StartRef’ Attribute

In the “Hello World” experiment so far, “Goodbye World” comes after “Hello World” in each trial because “Goodbye Event” is listed after “Hello Event” in the “Events” attribute of “Hello World”. We can change the order of the events by reversing the order of the references in the “Events” attribute, or we can explicitly set ordering dependencies using the “StartRef” event attribute.

Make “Goodbye World” appear *before* “Hello World”:

1. Add a new line to the “Hello Event” entry.
2. Type this attribute definition:

```
StartRef: 0 after end of "Goodbye Event"
```

Your “Hello Event” entry should now look like this:

```
Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Duration: Mouse[Click]
  StartRef: 0 after end of "Goodbye Event"
```

Now, when you run the experiment, “Goodbye World” will come before “Hello World” within each trial.

The general format for “StartRef” is *milliseconds* after start/end of *Event*, where *milliseconds* is the time to wait after the start or end of the event named *Event*. The special event name *START* can be used for the start of the trial (*START* can only be used with *after end of*).

When “StartRef” is not specified, it defaults depending on the order in which event entries are referenced in the “Events” attribute. If the first event listed does not have a “StartRef” value, the default is *0 after end of START*; for any other event, the default is *0 after end of Previous*, where *Previous* is the event that is listed just before this one in the “Events” attribute.

If you are still running five trials in each run of the experiment, it may not be clear after a few trials whether “Hello World” comes before or after “Goodbye World”. Let’s switch the order of “Hello World” and “Goodbye World” back to normal, and make “Hello World” start 500 milliseconds into the trial. This will give us a slight pause between the words of different trials.

1. Add a new line to the “Goodbye Event” entry.
2. Type this attribute definition:

```
StartRef: 0 after end of "Hello Event"
```

Your “Goodbye Event” entry should now look like this:

```
Goodbye Event::
  EventType: Text
  Stimulus: "Goodbye World"
  Duration: Mouse[Click] Key[Any]
           Time[2000]
  Color: Red
  StartRef: 0 after end of "Hello Event"
```

3. Change the value of “StartRef” in “Hello Event” to 500 after end of START.

Your “Hello Event” entry should now look like this:

```
Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Duration: Mouse[Click]
  StartRef: 500 after end of START
```

Now, when you run the experiment, each trial will do nothing for half a second, and then the “Hello World”-“Goodbye World” sequence will execute.

Start references are discussed further in “Part 4: Scripting Reference, 13.4.2 Start Reference”, p411.

10.1.4 Scripting Conditions and Actions

Conditions and actions for events are specified in the “EventActions” attribute. The value of the attribute is a list of pairs; the first half of each pair contains a list of conditions, and the second half contains a list of actions to be triggered by the conditions.

The “Hello World” text is cleared in our experiment when the subject clicks the mouse button. Let’s add an action that beeps an alert when a key is pressed instead.

1. Add a new line to the “Hello Event” entry.
2. Type this attribute definition:

```
EventActions: Conditions[Key[Any]] =>
              Actions[Beep[Instances: -1]]
```

Your “Hello Event” entry should now look like this

```

Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Duration: Mouse[Click]
  StartRef: 500 after end of START
  EventActions: Conditions[Key[Any]] =>
                Actions[Beep[Instances: -1]]

```

Now, when you run the experiment and press a key while “Hello World” is showing, you should hear a system beep from the Macintosh.

The generic form for a conditions-actions pair is: `Conditions[condition-list] => Actions[action-list]`. *condition-list* has the same syntax as for the “Duration” attribute in a condition-based duration, except that `Time[]` cannot be used; see “10.1.3.2 The ‘Duration’ Attribute”, p287.

action-list has a form similar to *conditions-list*; each action is specified with its name followed by parameters in square brackets (`[]`), but there are also two special attributes which can be specified within the square brackets: “ActiveUntil” and “Instances”. “Instances” and “ActiveUntil” must be specified after all parameters of the action, since an action declaration is really an inline entry (see “Part 4: Scripting Reference, 12.9 Inline Entries”, p335).

“ActiveUntil” specifies when the action should become inactive; this can be an event name — in which case the action is active until the given event starts — or one of a few keywords; see “Part 4: Scripting Reference, 13.4.1.2 Instances and ActiveUntil”, p410 for more information. The default is that the action lasts until the end of its event.

“Instances” specifies the number of times that the action can be triggered. `-1` means that it can be triggered any number of times.

The `Conditions` and `Actions` labels are not required in the generic form `Conditions[condition-list] => Actions[action-list]`, so that a conditions-actions pair can instead be specified as `[condition-list] => [action-list]`. This involves less typing, but may not be as clear to another reader of your script. If the `Conditions` and `Actions` labels are used, there cannot be any spaces between them and the opening square-bracket (`[`). Spaces around `=>` do not matter, but there cannot be any spaces between `=` and `>`.

Let’s record some data during “Hello Event” using the `RT[]` action. When the subject clicks the mouse to end the event, we will record a “Correct” response; when the subject presses a key instead, we will record “Incorrect”.

1. Add a new action to the existing conditions-actions pair in the “EventActions” attribute of “Hello Event”:

```
RT["Incorrect" Instances: -1]
```

2. Add a new conditions-actions pair in the “EventActions” attribute of “Hello Event”:

```
Conditions[End[ ]] => Actions[RT["Correct"]]
```

Your “Hello Event” entry should now look like this

```
Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Duration: Mouse[Click]
  StartRef: 500 after end of START
  EventActions:
    Conditions[Key[Any]] =>
      Actions[Beep[Instances: -1]
              RT["Incorrect" Instances: -1]]
    Conditions[End[]] =>
      Actions[RT["Correct"]]
```

The first parameter of RT[] is a response label that can be recorded to the data file, but the response label is not recorded by default:

3. Add a new attribute “DataFields” to the “Hello World” experiment entry:

```
DataFields: RESPONSE_LABEL
```

Your experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Events: "Hello Event" "Goodbye Event"
  Cycles: 5
  DataFields: RESPONSE_LABEL
```

Now, when you run the experiment, the data file (“Hello World Data”) will contain entries for each time you hit a key or click the mouse during “Hello Event”. (You can view the data file by selecting **View Data File** from the **Utilities** menu.)

Notice that in the conditions list for the “Correct” RT[], we used End[] — which triggers at the end of the action’s event — instead of Mouse[Click]. This lets us change the condition that ends “Hello Event” without also updating the “EventActions” attribute.

Conditions and actions are discussed further in “Part 4: Scripting Reference, 13.4.1 Action Lists”, p409. The list of all condition devices and a description of their parameters is in “Part 4: Scripting Reference, 14.3 Conditions and Inputs”, p437. The list of all actions and their parameters is in “Part 4: Scripting Reference, 14.1 Actions Reference”, p419.

10.1.5 Scripting Templates

So far, we have only worked at the extreme ends of the experiment hierarchy: the experiment as a whole and the individual events which make up a trial. The template definition was implicitly in the experiment entry, and all trials were derived from this implicit definition. In order to have trials with completely different structures, we need to explicitly define templates.

Explicitly define the template entry for the “Hello World” experiment:

1. Add a couple of lines to the script in the #> TemplateDefinitions section of the script.
2. Insert this entry definition:

```
HelloGoodbye Template::
    Events: "Hello Event" "Goodbye Event"
```

The “Events” attribute is the same as in the “Hello World” experiment entry.

3. Delete the “Events” attribute from the “Hello World” experiment entry.
4. Add a “Templates” attribute to the “Hello World” experiment entry:

```
Templates: "HelloGoodbye Template"
```

Your “Hello World” experiment entry should now look like this:

```
Hello World::
    Format: Factor
    InputDevices: MOUSE KEY
    Timer: Macintosh
    Flags: NO_SAVE_SCREEN
    DataFile: "Hello World Data"
    ScaleBlocks: 1
    Cycles: 5
    DataFields: RESPONSE_LABEL
    Templates: "HelloGoodbye Template"
```

If you run the script now, it should execute exactly as before. We have defined the template explicitly, but we did not change its definition.

If you open the graphic environment’s Design window now, you will see that “Hello Event” and “Goodbye Event” are no longer connected directly to the experiment; there is now a template object inserted between them.

The “Templates” attribute specifies the list of templates available for use in building a trial. For each trial, only one template will be used.

Define a new template which only has one event, displaying “Hello World”. There is no need to define a new event entry; we can use “Hello Event” for this template, too.

1. Add a couple of lines to the script after “HelloGoodbye Template” in the #> TemplateDefinitions section of the script.
2. Insert this entry definition:

```
Hello Template::
    Events: "Hello Event"
```

3. Add "Hello Template" after "HelloGoodbye Template" in the "Templates" attribute of the "Hello World" experiment entry. Your "Hello World" experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Cycles: 5
  DataFields: RESPONSE_LABEL
  Templates: "HelloGoodbye Template"
            "Hello Template"
```

Now, when you run the experiment, you will see one trial with "Hello World" followed by "Goodbye World", then a trial with only "Hello World", then another "Hello-Goodbye", then another "Hello", and finally one more "Hello-Goodbye".

When each trial is being compiled, templates are selected from the "Templates" attribute by accessing it as a scripting list. Scripting lists — which are different from the "lists" of the graphic environment — were introduced in "9.5 Lists", p279 and are discussed fully in "Part 4: Scripting Reference, 12.8 Lists", p329.

The default access type of a list is `Sequential`, so trials of "Hello World" were built by picking "HelloGoodbye Template" and "Hello Template" in the order which they are listed in the "Templates" attribute of "Hello World".

Make the ordering of "HelloGoodbye Template" and "Hello Template" random:

- Add an "AccessType" sub-attribute to the "Templates" attribute of "Hello World":

```
AccessType:> Random
```

Your "Hello World" experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Cycles: 5
  DataFields: RESPONSE_LABEL
  Templates: "HelloGoodbye Template"
            "Hello Template"
  AccessType:> Random
```

Now, when you run "Hello World", you will see both "Hello" and "Hello-Goodbye" in each pair of trials, but their relative order will be random.

If you look at the data file after running "Hello World", you will see that "Correct" and "Incorrect" are being recorded for "Hello Event" in both "HelloGoodbye Template" and "Hello Template", but there is no way to tell which template was used to build the trial that contained the response.

The “Condition” field of the data file can be used to make this distinction. By default, the condition name is empty. It can be set using the “ConditionName” attribute.

Set the condition name for “HelloGoodbye Template” and “Hello Template”:

1. Add a “ConditionName” attribute to the “HelloGoodbye Template” entry:

```
ConditionName: "HelloGoodbye"
```

Your “HelloGoodbye Template” entry should now look like this:

```
HelloGoodbye Template::
  Events: "Hello Event" "Goodbye Event"
  ConditionName: "HelloGoodbye"
```

2. Add a “ConditionName” attribute to the “Hello Template” entry:

```
ConditionName: "Hello"
```

Your “Hello Template” entry should now look like this:

```
Hello Template::
  Events: "Hello Event"
  ConditionName: "Hello"
```

Now, when you run the experiment and look at the data file, you can see which template was used for the trial by looking at the “Condition” field of each line.

10.1.5.1 Scripting Trial Actions

Actions can be defined to cover the entire trial using the “TrialActions” attribute in a template entry. The syntax is the same as for the “EventActions” attribute of an event entry.

Add an action to play the built-in “Bloop” sound anytime the space bar is hit during the trial:

- Add a “TrialActions” attribute to both “HelloGoodbye Template” and “Hello Template”:

```
TrialActions: Conditions[Key[SPACE]] =>
  Actions[Beep["Bloop"]]
```

Your “HelloGoodbye Template” entry should now look like this:

```
HelloGoodbye Template::
  Events: "Hello Event" "Goodbye Event"
  ConditionName: "HelloGoodbye"
  TrialActions: Conditions[Key[SPACE]] =>
    Actions[Beep["Bloop"]]
```

Your “Hello Template” entry should now look like this:

```
Hello Template::
  Events: "Hello Event"
  ConditionName: "Hello"
  TrialActions: Conditions[Key[SPACE]] =>
    Actions[Beep["Bloop"]]
```

Now, when you run the experiment and hit the space bar, you will hear the “Bloop” sound (but only once during a given trial).

Trial actions are discussed in more detail in “Part 4: Scripting Reference, 13.1.6.2 Standard Trial Attributes”, p366.

10.1.5.2 Attribute Inheritance

It is unnecessary to define “TrialActions” in both “HelloGoodbye Template” and “Hello Template” when we want the same trial actions definition in both templates. We can simply put the attribute in the experiment entry and let both templates *inherit* the attribute value.

Move the duplicated “TrialActions” definition to the “Hello World” experiment entry, so that it does not have to be defined twice:

1. Copy the “TrialActions” attribute from the “HelloGoodbye Template” and paste it into the “Hello World” experiment entry. Your “Hello World” experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Cycles: 5
  DataFields: RESPONSE_LABEL
  Templates: "HelloGoodbye Template"
            "Hello Template"
            AccessType:> Random
  TrialActions: Conditions[Key[SPACE]] =>
              Actions[Beep["Bloop"]]
```

2. Delete the “TrialActions” attribute from both the “HelloGoodbye Template” and “Hello Template” entries.

Your “HelloGoodbye Template” entry should now look like this:

```
HelloGoodbye Template::
  Events: "Hello Event" "Goodbye Event"
  ConditionName: "HelloGoodbye"
```

Your “Hello Template” entry should now look like this:

```
Hello Template::
  Events: "Hello Event"
  ConditionName: "Hello"
```

If you run the experiment now, it will work exactly as it did before.

Attribute inheritance is a rule that allows entries to use attribute values specified in their owners. For example, an event entry can inherit attributes from the template it is linked to, plus whatever the template is linked to, and so on up to the experiment entry.

Change the font to Geneva for all of the Text events.

- Add a “Font” attribute to the “Hello World” experiment entry:

```
Font: Geneva
```

Your “Hello World” experiment entry should now look like this:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Cycles: 5
  DataFields: RESPONSE_LABEL
  Templates: "HelloGoodbye Template"
             "Hello Template"
             AccessType:> Random
  TrialActions: Conditions[Key[SPACE]] =>
               Actions[Beep["Bloop"]]
  Font: Geneva
```

Now, when you run the experiment, “Hello World” and “Goodbye World” are always printed in the Geneva font, instead of the system font (Chicago).

Attribute inheritance is primarily used to set default values for stimulus attributes (such as the “Font” attribute, above). However, inheritance can also be used to vary the value of an attribute depending on which path of links is followed to build the trial.

Change the “Hello World” event so that it is blue when it is going to be followed by “Goodbye World” in the trial. For trials which do not have a “Goodbye World”, the stimulus will still be black.

- Add a “Color” attribute to the “HelloGoodbye Template” entry:

```
Color: Blue
```

Your “Hello Template” entry should now look like this:

```
HelloGoodbye Template::
  Events: "Hello Event" "Goodbye Event"
  ConditionName: "HelloGoodbye"
  Color: Blue
```

Now, when you run the experiment, you will know to expect a “Goodbye World” whenever you see a blue “Hello World” instead of black.

Inheritance works for most — but not all — attributes. Attributes which define the *structure* of the experiment — such as the “Templates” or “Events” attribute — cannot be inherited. Also, the “Stimulus” event attribute cannot be inherited. These technicalities are explained in detail in “Part 4: Scripting Reference, 13.3.11.1 Structural vs. Non-structural Attributes”, p402.

10.1.5.3 TrialAttrib()

When you want to vary an event attribute with respect to the template (i.e., vary the attribute based on which template is used for the trial), attribute inheritance may not always

provide sufficient functionality to do what you want. This can happen if you want to vary the “Stimulus” attribute (which cannot be inherited), or if multiple events require different values for the same attribute.

These problems can be solved by using a more explicit reference to the template: the `TrialAttrib()` function call.

Change the “Hello World” event to read “Hello World...” (with ellipses) when it is going to be followed by “Goodbye World” in the trial. For trials which do not have a “Goodbye World”, the stimulus will remain “Hello World”.

1. Add a “HelloStimulus” attribute to the “Hello Template” entry:

```
HelloStimulus: "Hello World"
```

Your “Hello Template” entry should now look like this:

```
Hello Template::  
  Events: "Hello Event"  
  ConditionName: "Hello"  
  HelloStimulus: "Hello World"
```

2. Also, add a “HelloStimulus” attribute to the “HelloGoodbye Template” entry, but with a slightly different value:

```
HelloStimulus: "Hello World..."
```

Your “Hello Template” entry should now look like this:

```
HelloGoodbye Template::  
  Events: "Hello Event" "Goodbye Event"  
  ConditionName: "HelloGoodbye"  
  Color: Blue  
  HelloStimulus: "Hello World..."
```

3. Change the value of “Stimulus” attribute in the “Hello Event” entry:

```
Stimulus: TrialAttrib>HelloStimulus)
```

Your “Hello Event” entry should now look like this:

```
Hello Event::  
  EventType: Text  
  Stimulus: TrialAttrib>HelloStimulus)  
  Duration: Mouse[Click]  
  StartRef: 500 after end of START  
  EventActions:  
    Conditions[Key[Any]] =>  
      Actions[Beep[Instances: -1]  
        RT["Incorrect" Instances: -1]]  
    Conditions[End[]] =>  
      Actions[RT["Correct"]]
```

Now, when you run the experiment, you will know to expect a “Goodbye World” whenever you see “Hello World...” instead of “Hello World”.

When attribute values are read while building a trial, `TrialAttrib()` function calls are replaced by values read from the template entry (for the template that is being used to build

the current trial). The single parameter to `TrialAttrib()` specifies which attribute in the template entry to read the value from.

In the above example, `TrialAttrib>HelloStimulus` is replaced with "Hello World" or "Hello World...", depending on whether the "HelloStimulus" attribute is read from the "Hello Template" entry or the "HelloGoodbye Template" entry.

See also "Part 4: Scripting Reference, 13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes", p385.

10.1.6 Scripting Experiment Attributes

There are a large number of standard attributes which can be specified in the experiment entry. These attributes control features of the experiment as a whole, such as the background color of the screen, or which input devices are currently active.

An exhaustive list of the available experiment attributes is provided in "Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes", p360. You should browse through this list to get an idea of all of the features available in PsyScope. (Some of these attributes can not be set through the graphic environment.)

10.2 Scripting Factors

There is a great deal that can be said about the structure and use of factors in PsyScope; however, such information is contained elsewhere in this manual (see "Part 2: Graphic Environment Reference, 5.7 Factors and Lists", p129). This section assumes that the reader possesses a basic understanding of factors and their role in a PsyScope experiment design.

We will once again be implementing the "Acuity" experiment, which was used as the example experiment in the graphic environment tutorial ("Chapter 3. Designing an Experiment", p9). However, once we have scripted this experiment, it will look somewhat different from the script built by the graphic environment. This is because the graphic environment uses a complex structure — the *factor table* — to define experiments in a manner that is much like form-filling. Scripters tend instead to use simpler structures — *free factors*, also known as *lists* in the graphic environment — which are more amenable to programmatic manipulation.

10.2.1 Scripting the Acuity Experiment

Before we begin scripting factors, we must first set up the higher-level structure of the Acuity experiment.

In Acuity, there are three events:

- A fixation point is shown in the center of the screen; it stays there until the subject hits the “2” key, indicating that s/he is ready for the stimulus. This event is the same for every trial.
- A stimulus is shown somewhere on the screen for a brief duration. The size, position, and “wordness” of this stimulus will vary from trial to trial. (These variations will be implemented with factors.)
- A response is collected — either the “1” key or “2” key, indicating that the stimulus was a word or non-word, respectively. This event is the same for every trial.

(The graphic environment tutorial also used an ITI event; we will instead use the “ITI” trial attribute to implement an inter-trial interval.)

Thus, we will need an experiment entry, one template entry, and three event entries.

Create the skeleton of the Acuity experiment:

1. Create a new script by using the **New Experiment** item in the **Design** menu. Name the experiment “Acuity” and save it to its own script file. Once your Acuity script is loaded, open the interactive editor (see “10.1 Scripting a New Experiment”, p281). You will want to make sure that the Design window is closed, so that it does not interrupt your work by trying to read incomplete fragments of the script.
2. In the #> EventDefinitions section of the script, create three new events:

First, create the “Fixation” event:

```
Fixation::
  Stimulus: "+"
  Position: Center Screen Center
           Center Screen Center
  Duration: Key[2]
```

The “Position” attribute is described in “Part 4: Scripting Reference, 14.2.1.1 Text and Screen Attributes”, p424. Here, we have specified that the “+” should be centered both horizontally and vertically with respect to the screen. The duration `Key[2]` means that the fixation symbol will stay on the screen until the “2” key is pressed.

Create the “Stimulus” event:

```
Stimulus::
  Stimulus: "Hello"
  Position: Center Screen Center
           Center Screen Center
  Size: 12
  Duration: 50
```

For now, the “Stimulus”, “Position” and “Size” attributes are set to constant values; these will change when we are ready to link the event to factors. The duration is set to a fixed value of 50 milliseconds; this can be changed to any value you prefer.

Finally, create the “Response” event:

```
Response::
  EventType: NULL
  Duration: Key[1 3]
  EventActions: [Key[1]] => [RT[correct]]
                [Key[3]] => [RT[incorrect]]
```

The “Response” event does not show any stimulus, so its event type is NULL. For now, we have assumed that “1” is a correct response and “3” is an incorrect response, although this will change when we are ready to link the response to a factor. The duration attribute specifies that the event lasts until some response is given.

3. In the #> TemplateDefinitions section of the script, create a new template entry:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
```

The template is linked to the three events. The value of the “ITI” attribute specifies that 500 milliseconds should separate the end of one trial and the start of the next.

A template entry is not really necessary in this implementation of the Acuity experiment. However, it is generally a good idea to use a template entry, because it is then easier to change the script in the future.

4. Link the template to the experiment entry, set the trial count to 5, and add RESPONSE_LABEL to the data fields. Your “Acuity” experiment entry should now look like this:

```
Acuity::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Acuity Data"
  ScaleBlocks: 1
  DataFields: RESPONSE_LABEL
  Cycles: 5
  Templates: "Acuity Template"
```

If you run the experiment now, you will see a fixation point. Hit the “2” key, and the fixation point will disappear, followed immediately by the stimulus. When the stimulus has disappeared, you must hit the “1” or “2” key as a response (nothing will be on the screen); this will end the trial, and then the next trial will start after half a second.

10.2.2 Scripting Free Factors

When you set up a factor design in the graphic environment, you typically use a factor table object. However, a factor table has a very awkward implementation in PsyScript. (You might want to look at a table-based script to see what it looks like.) For scripted experiments, you will usually want to use *free factors*, instead.

When you use free factors, each factor is described by a separate entry in the script. Factors are connected to the experiment hierarchy with a “Factors” attribute; a “Factors” attribute can appear anywhere in the entry hierarchy from the experiment entry to the template entry. The way in which factors are crossed together and accessed (i.e., *factor sets* and their *crossing types* and *access types*) is controlled through sub-attributes of the “Factors” linking attribute.

Once a factor is defined and linked to the experiment hierarchy, attributes can access field values using a `FactorAttrib()` function call.

Create one of the factors needed for the Acuity experiment — the “Position” factor:

1. In the `#> FactorDefinitions` section of the script, create a new factor entry:

```
Position::
  Levels: LeftFar LeftNear Center RightNear
         RightFar
```

2. In the `#> LevelDefinitions` section of the script, create entries for the five levels of the “Position” factor:

```
LeftFar::
  Pos: 10% Screen Center Center Screen Center

LeftNear::
  Pos: 30% Screen Center Center Screen Center

Center::
  Pos: 50% Screen Center Center Screen Center

RightNear::
  Pos: 70% Screen Center Center Screen Center

RightFar::
  Pos: 90% Screen Center Center Screen Center
```

3. Link the factor entry to the “Acuity Template” entry by adding a “Factors” attribute:

```
Factors: Position
AccessTypes:> Random
```

Your “Acuity Template” entry should now look like this:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
  Factors: Position
  AccessTypes:> Random
```

4. Make the position of the stimulus depend on the “Position” factor by changing the value of the “Position” attribute of the “Stimulus” entry:

```
Position: FactorAttrib(Position Pos)
```

Your “Stimulus” entry should now look like this:

```
Stimulus::
  Stimulus: "Hello"
  Position: FactorAttrib(Position Pos)
  Size: 12
  Duration: 50
```

Now, when you run the experiment, the stimulus will show up in a different place on the screen in each trial.

The content of a “Factors” attribute lists references to other factor entries. Each factor entry, in turn, contains a “Levels” attribute with a list of references to level entries. Within each level entry, one or more attributes specify field values for that level of the factor.

In our first factor, there is only one field — “Pos” — which contains a position specification. The “Position” attribute of the “Stimulus” entry refers to this field with the function call `FactorAttrib(Position Pos)`. When the trial is compiled, this function call is evaluated by finding the entry for the current level of the specified factor, and copying the value of the specified field attribute. For example, in trials where “FarNear” is the current level of the “Position” factor, `FactorAttrib(Position Pos)` is replaced with `30% Screen Center Center Screen Center`.

10.2.2.1 Compact Factors

The field values of the “Position” factor are somewhat complicated, since it takes six values to fully specify a position. When the values are simpler, there is no need to create separate entries for each level of the factor.

Create another of the factors needed for the Acuity experiment — the Position factor:

1. In the `#> FactorDefinitions` section of the script, create a new factor entry:

```
Size::
  Levels: Big Medium Small
  IsList: True
  PointSize: 18 12 9
```

2. Link the “Size” factor entry to the “Acuity Template” entry by modifying the “Factors” attribute:

```
Factors: Position Size
AccessTypes:> Random
```

Your “Acuity Template” entry should now look like this:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
  Factors: Position Size
  AccessTypes:> Random
```

3. Make the size of the stimulus depend on the “Size” factor by changing the value of the “Size” attribute of the “Stimulus” entry:

```
Size: FactorAttrib(Size PointSize)
```

Your “Stimulus” entry should now look like this:

```
Stimulus::  
  Stimulus: "Hello"  
  Position: FactorAttrib(Position Pos)  
  Size: FactorAttrib(Size PointSize)  
  Duration: 50
```

Now, when you run the experiment, both the position and size of the stimulus will vary from trial to trial.

Setting the “IsList” attribute of a factor entry to `True` designates the factor as a *compact factor*. In a compact factor, the field values for levels are listed in attributes within the factor entry, in parallel with the “Levels” attribute. The values in the “Levels” attribute are simply read as level names, instead of references to level entries.

10.2.2.2 Factor Interactions

There are still two things which must vary in our Acuity experiment: the word or non-word nature of the stimulus, and the actual text that is displayed. Unlike the other two factors (position and size), these two factors are related: the possible values for the text that is displayed during the trial depends on whether it is a word or non-word trial.

When such interactions arise in a table-based script, they are implemented by setting values for individual cells rather than full rows or columns. When free factors are used, there is no physical representation for a cell, so that other means have to be used to implement the interaction.

Add the two remaining factors for the Acuity experiment — Wordness and TextStim:

1. In the `#> FactorDefinitions` section of the script, create two new factor entries:

```
TextStim::  
  Levels: 1 2 3  
  IsList: True  
  Words: dog cat man  
  Nonwords: biv gaf dut  
  
Wordness::  
  Levels: Word Nonword  
  IsList: True  
  Stimulus: FactorAttrib(TextStim Words)  
            FactorAttrib(TextStim Nonwords)  
  RightKey: 1 3  
  WrongKey: 3 1
```

2. Link the “TextStim” and “Wordness” factor entries to the “Acuity Template” entry by modifying the “Factors” attribute:

```
Factors: Position Size TextStim Wordness
AccessTypes:> Random
```

Your “Acuity Template” entry should now look like this:

```
Acuity Template::
Events: Fixation Stimulus Response
ITI: 500
Factors: Position Size TextStim Wordness
AccessTypes:> Random
```

3. Make the value of the stimulus depend on the “TextStim” and “Wordness” factors by changing the value of the “Stimulus” attribute of the “Stimulus” entry:

```
Stimulus: FactorAttrib(Wordness Stimulus)
```

Your “Stimulus” entry should now look like this:

```
Stimulus::
Stimulus: FactorAttrib(Wordness Stimulus)
Position: FactorAttrib(Position Pos)
Size: FactorAttrib(Size PointSize)
Duration: 50
```

4. Make the response recording sensitive to the “Wordness” factor by changing the “EventActions” attribute of the “Response” entry:

```
EventActions:
[Key[FactorAttrib(Wordness RightKey)]]
=> [RT[correct]]
[Key[FactorAttrib(Wordness WrongKey)]]
=> [RT[incorrect]]
```

Your “Response” entry should now look like this:

```
Response::
EventType: NULL
Duration: Key[1 3]
EventActions:
[Key[FactorAttrib(Wordness RightKey)]]
=> [RT[correct]]
[Key[FactorAttrib(Wordness WrongKey)]]
=> [RT[incorrect]]
```

Now you have a complete Acuity experiment. The position, size, wordness, and stimulus should vary from trial to trial, and a “1” or “3” key press should be recorded as “correct” or “incorrect”.

In the above implementation of “TextStim” and “Wordness”, “TextStim” is never directly accessed from any of the event entries. Instead, “TextStim” references are within “Wordness” field values, thus modelling the interaction between “Wordness” and “TextStim”.

These factors are our first with multiple fields; “TextStim” has “Words” and “Nonwords” fields, while “Wordness” has “Stimulus”, “RightKey”, and “WrongKey”.

Notice that the `FactorAttrib()` function calls are embedded within a more complex action description in the “EventActions” attribute of “Response”. `FactorAttrib()` can be placed anywhere in the content of an attribute, except within string literals which are surrounded by quotes or curly braces (see “Part 4: Scripting Reference, 12.6.1 Literals”, p326).

10.2.2.3 Factor Sets

When you run the Acuity experiment at this point, it is very likely that you will see some word or non-word multiple times before you see all of the words and non-words once. This is because “Wordness” and “StimText” are fully crossed with “Position” and “Size”; it is guaranteed that you will see every position-size-word/non-word combination once before any repeats, but the same word could be seen two different positions or sizes before some other word is seen once.

One way to solve this problem is to put “Wordness” and “StimText” into a separate *factor set* (or *table*).

Partition the factors of “Acuity Template” into two factor sets:

- Add a “Sets” sub-attribute to the “Factors” attribute in “Acuity Template”:

```
Sets:> 2 2
```

- Add another Random to the “AccessTypes” sub-attribute of “Factors” in “Acuity Template”:

```
AccessTypes:> Random Random
```

Your “Acuity Template” entry should now look like this:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
  Factors: Position Size TextStim Wordness
           AccessTypes:> Random Random
           Sets:> 2 2
```

Now, when you run the experiment, you should see every word and non-word once before any word or non-word is used twice.

When you run the script in real-time, it can be very difficult to keep track of which words and non-words you have seen. Try using the Trial Monitor with the **Check Events** and **List Events** options (see “Part 2: Graphic Environment Reference, 6.3 The Trial Monitor”, p238).

The “Sets” sub-attribute of the “Factors” attribute should contain a list of numbers. Each number represents a single factor set, indicating how many factors are in that set. Factors from the content of the “Factors” attribute are placed into sets by picking the first n items for the first set, the next m items for the second set, and so on, where n , m , etc. are the values in the “Sets” sub-attribute.

The “Factors” attribute can have other sub-attributes which determine properties of factor sets; e.g., the “AccessTypes” sub-attribute. Values are listed in these attributes in parallel

with the “Sets” sub-attribute, i.e., there is one value for each number in the “Sets” sub-tribute.

Putting “StimText” and “Wordness” into a separate factor set is not necessarily an acceptable solution. We have lost a useful constraint — that every possible stimulus value is seen in every possible position and size.

Another solution to our problem is to keep all four factors in the same set, but change the *access type* of the set.

Put “StimText” and “Wordness” back into the first factor set, and change the access type of this set:

- Change the “Sets” sub-attribute of “Factor” in “Acuity Template”:

```
Sets:> 4
```

- Change the “AccessTypes” sub-attribute of “Factors” in “Acuity Template”:

```
AccessTypes:> LRandom
```

- Reorder the factors in the “Factors” attribute in “Acuity Template”, so that “TextStim” and “Wordness” have precedence:

```
Factors: Wordness TextStim Position Size
```

Your “Acuity Template” entry should now look like this:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
  Factors: TextStim Wordness Position Size
           AccessTypes:> LRandom
           Sets:> 4
```

Now, when you run the experiment, you should still see every word and non-word once before any word or non-word is used twice, but every word and non-word will also be used in every size and position (if you run enough trials) before the same word or non-word appears in the same size and same position.

The `LRandom` access type is the same as **Least-Used Random** in the graphic environment (see “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140). The complete list of access types are in “Part 4: Scripting Reference, 13.3.7.3 Scripting Access Types”, p391.

There are many other properties of a factor set which you can manipulate. All of these are described in “Part 4: Scripting Reference, 13.3.7 Scripting Factors”, p387.

10.2.2.4 Nested Factors

We have so far implemented the stimulus-selecting factor as an interaction between two top-level factors: “Wordness” and “StimText”. A more natural approach, however, may be to first choose a level of the “Wordness”, and then use a *nested factor* of the selected level:

a “WordStimulus” factor in the “Word” level, and a “NonwordStimulus” factor in the “Nonword” level.

In order to use nested factors, the levels must be implemented as separate entries (i.e., the “owning” factor cannot be a compact factor).

Change the Acuity experiment to use nested factors for the stimulus value selection:

1. Delete the “StimText” factor link in “Acuity Template”, setting the “Sets” sub-attribute of “Factors” to 3 and “AccessTypes” sub-attribute to CRandom. Your “Acuity Template” entry should now look like this:

```
Acuity Template::
  Events: Fixation Stimulus Response
  ITI: 500
  Factors: Wordness Position Size
           AccessTypes:> CRandom
           Sets:> 3
```

2. Create two level attributes for “Wordness” in the #> LevelDefinitions section of the script:

```
Word::
  Stimulus: FactorAttrib(WordStimulus)
  RightKey: 1
  WrongKey: 3
  Factors: WordStimulus

Nonword::
  Stimulus: FactorAttrib(NonwordStimulus)
  RightKey: 3
  WrongKey: 1
  Factors: NonwordStimulus
```

(Here, we are exploiting the fact that the field parameter of the `FactorAttrib()` function defaults to the name of the attribute that owns the `FactorAttrib()` call. Thus, in “Word”, the function call `FactorAttrib(WordStimulus)` is equivalent to `FactorAttrib(WordStimulus Stimulus)`.)

3. Delete all of the attributes of “Wordness” except “Levels”. Your “Wordness” entry should now look like this

```
Wordness::
  Levels: Word Nonword
```

4. Create the two nested factors in the #> FactorDefinitions section of the script:

```
WordStimulus::
  Levels: 1 2 3
           AccessType:> Random
  IsList: True
  Stimulus: dog cat man

NonwordStimulus::
  Levels: 1 2 3
           AccessType:> Random
  IsList: True
  Stimulus: biv gaf dut
```


5. Delete the “StimText” factor from the script.

Now, when you run the experiment, it should behave mostly as before. The interesting properties should be the same (e.g., using every word or non-word once before using any word or non-word twice).

Nested factors are always associated to levels — they cannot be linked to factors. (This would not make sense.)

The “Factors” attribute in a level entry has the same format as when it is in a template, block, group, or experiment entry; thus, the access type on the nested factor can be controlled through the “AccessTypes” sub-attribute of the “Factors” attribute, as before. Here, however, we have chosen to use the “AccessType” (singular) sub-attribute in the “Levels” attributes of the “WordStimulus” and “NonwordStimulus”. This produces the same results as using `Random` in an “AccessTypes” sub-attribute within “Word” and “Nonword”; see “Part 4: Scripting Reference, Scripting the Factor-based Access Type”, p392 for details.

See “Part 4: Scripting Reference, 13.3.7.5 Scripting Nested Factors”, p393 for more information on scripting nested factors.

10.2.3 Scripting Factor Tables

Even though you would not want to script a factor table from scratch, you may find yourself modifying a script that was created in the graphic environment using factor tables. The format and use of factor table entries is beyond the scope of this tutorial, but they are fully described in “Part 3: Scripting User Manual, 10.2.3 Scripting Factor Tables”, p309.

10.3 Scripting Blocks and Groups

10.3.1 Scripting Blocks and BlockAttrib()

Let’s add blocks to the Acuity experiment — as in “Chapter 3. Designing an Experiment”, p9, the graphic environment tutorial — to test the role of attention in a subject’s performance in the experiment. We will need to have four blocks of trials: a single trial to present the initial instructions, a block of trials using these instructions, a single trial to present the second instructions, and then a block of trials using these new instructions.

Add the blocks and create the instructions template:

1. Create a new event in the #> `EventDefinitions` section of the script:

```
Instruction Event::
  EventType: Paragraph
  Duration: Key[Any]
  Stimulus: BlockAttrib(Instructions)
```

2. Create a new template in the #> TemplateDefinitions section of the script:

```
Instructions Template::  
    Events: "Instruction Event"
```

3. In the #> BlockDefinitions section of the script, create three new entries:

```
FirstInstructions::  
    FixedCycles: 1  
    Templates: "Instructions Template"  
    Instructions: "Always look at the center of  
the screen."
```

```
SecondInstructions::  
    FixedCycles: 1  
    Templates: "Instructions Template"  
    Instructions: "Look at the stimulus when it  
appears."
```

```
RegularTrials::  
    Templates: "Acuity Template"
```

4. Change the “Acuity” experiment entry: first, delete the “Templates” and “Cycles” attributes; then, change the “ScaleBlocks” attribute value to 5; finally, add a “Blocks” attribute:

```
Blocks: FirstInstructions RegularTrials  
        SecondInstructions RegularTrials
```

Your “Acuity” experiment entry should now look like this:

```
Acuity::  
    Format: Factor  
    InputDevices: MOUSE KEY  
    Timer: Macintosh  
    Flags: NO_SAVE_SCREEN  
    DataFile: "Acuity Data"  
    ScaleBlocks: 5  
    DataFields: RESPONSE_LABEL  
    Blocks: FirstInstructions RegularTrials  
            SecondInstructions RegularTrials
```

Now, when you run the experiment, you should see the “Always look at the center of the screen” instructions as the first trial, followed by five normal Acuity trials; then, you should see the “Look at the stimulus when it appears” instructions as the next trial, followed by another five normal Acuity trials.

The “FirstInstructions”, “SecondInstructions”, and “RegularTrials” blocks are linked to the experiment through the “Blocks” attribute. “RegularTrials” is listed in this attribute twice; this means that the block should be run *twice* for each pass through the list of blocks. The blocks are executed in the default order, which is sequential.

“FirstInstructions” and “SecondInstructions” are both linked to the same template. The instructions presented by the trial are varies across the blocks by using the BlockAttrib() function. BlockAttrib() works much like TrialAttrib() (described in “10.1.5.3 TrialAttrib()”, p297), except that the referenced attribute is in the block entry instead of the

template entry. See also “Part 4: Scripting Reference, 13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.

Both “FirstInstructions” and “SecondInstructions” use the “FixedCycles” attribute, with a value of 1. This means that only one trial should be run within the block. The “RegularTrials” block does not specify a trial count; by default, the trial count is 1, but this count is *scalable*. The “ScaleBlocks” attribute in “Acuity” has a value of 5, which causes the number of trials within a “RegularTrials” block to be scaled to 5.

Block entries are described in more detail in “Part 4: Scripting Reference, 13.3.4 Scripting Blocks”, p380.

10.3.2 Scripting Groups

Groups can be scripted into the experiment hierarchy similar to how blocks were added. In the simplest case, you can set up groups in your experiment, and then manually pick a group for each run of the experiment. Usually, however, you will want to automatically pick a group based on information that you enter about the current subject.

Subject information tracking is best implemented by using the Subject Info facilities of the graphic environment. There is no structured subject information tracking system built into either PsyScope or factor format; instead, PsyScope’s general-purpose environment configuration is combined with a few “external” functions (built-in DCODs) to attain subject info tracking independent of the format of the script.

Here, we will demonstrate the use of manually-selected groups, in order to show how groups fit into the basic experiment hierarchy.

Create two groups for the Acuity experiment: one group will see only one block of trials (fixating on the center of the screen at all times), while the other group will get the full experiment (with both center-fixation and direct-look blocks):

1. In the #> GroupDefinitions section of the script, create two new entries:

```
SimpleGroup::
  Blocks: FirstInstructions RegularTrials
  ScaleBlocks: 5

FullRunGroup::
  Blocks: FirstInstructions RegularTrials
          SecondInstructions RegularTrials
  ScaleBlocks: 5
```

2. Change the “Acuity” experiment entry: first, delete the “Blocks” and “ScaleBlocks” attributes; then, add a “Groups” attribute:

```
Groups: SimpleGroup FullRunGroup
Current:> 1
```

Your “Acuity” experiment entry should now look like this:

```
Acuity::  
  Format: Factor  
  InputDevices: MOUSE KEY  
  Timer: Macintosh  
  Flags: NO_SAVE_SCREEN  
  DataFile: "Acuity Data"  
  DataFields: RESPONSE_LABEL  
  Groups: SimpleGroup FullRunGroup  
  Current:> 1
```

Now, when you run the experiment, you should see only the “Always look at the center of the screen” block. If you change the value of the “Current” sub-attribute of the “Groups” attribute to 2, then you will see the full experiment again.

Notice that the experiment entry now contains no trial-counting attributes; when a group is used, all of the trial counting attributes are read from the group entry instead of the experiment entry.

Group entries are described in more detail in “Part 4: Scripting Reference, 13.3.3 Scripting Groups”, p379.

10.3.2.1 GroupAttrib()

The `GroupAttrib()` function is directly analogous to the `BlockAttrib()` and `TrialAttrib()` functions. See “Part 4: Scripting Reference, 13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385 for details.

10.4 Advanced Topics

10.4.1 Linking to the PsyScope Environment

As we implemented the Acuity experiment, the keys “2”, “1”, and “3” were “hardwired” as the “go”, “word”, and “non-word” keys, respectively; if you wanted to change this key mapping, you would have to go change every entry that depends on the key assignments.

We instead can store the key assignments in a small set of entries and then reference these entries whenever key assignments are needed. We can then link this single entry to the menu system, so that the key assignments can be modified by someone who does not know PsyScript.

Create a central entry for the key mappings and link this to the menu system:

1. In the `#> MenuDefinitions` section of the script, create a new entry:

```
Go Keys:: 2  
  Dialog: KeyState  
  
Word Keys:: 1
```

```
Dialog: KeyState
```

```
Nonword Keys:: 3
  Dialog: KeyState
```

These entries store the keyboard mapping. They also the PsyScope environment what dialog should be used to modify the values: the `KeyState` dialog.

2. Add "-", "Go Keys", "Word Keys", and "Nonword Keys" to the entry named "Experiment" (not to be confused with the "Experiments" entry or an experiment entry!).

The "Experiment" entry defines the **Experiment** menu in the PsyScope environment. When you put a dash in its content (" - "), a separator line is put in the menu.

Your "Experiment" entry should now look like this:

```
Experiment::
  @StandardPsyScopeMenuItems
  "- "
  "Go Keys"
  "Word Keys"
  "Nonword Keys"
```

3. Select **Reinitialize Script** from the **Utilities** menu. This resets the PsyScope environment menus, reading in your changes to the "Experiment" entry.

Now, when you click look in the **Experiment** menu in the menu bar, you should see **Go Keys**, **Word Keys**, and **Nonword Keys** at the bottom of the menu. However, these values are not yet linked to the experiment.

4. Change the "Duration" attribute of the "Fixation" event entry:

```
Duration: Key["@Go Keys"]
```

Your "Fixation" entry should now look like this:

```
Fixation::
  Stimulus: "+"
  Position: Center Screen Center
            Center Screen Center
  Duration: Key["@Go Keys"]
```

5. Change the "Duration" attribute of the "Response" event entry:

```
Duration: Key["@Word Keys" @"Nonword Keys"]
```

Your "Response" entry should now look like this:

```
Response::
  EventType: NULL
  Duration: Key["@Word Keys" @"Nonword Keys"]
  EventActions:
    [Key[FactorAttrib(Wordness RightKey)]]
```

```
=> [RT[correct]]
[Key[FactorAttrib(Wordness WrongKey)]]
=> [RT[incorrect]]
```

6. Change the “RightKey” and “WrongKey” attributes of the “Word” level entry:

```
RightKey: @"Word Keys"
WrongKey: @"Nonword Keys"
```

Your “Word” entry should now look like this:

```
Word::
  Stimulus: FactorAttrib(WordStimulus)
  RightKey: @"Word Keys"
  WrongKey: @"Nonword Keys"
  Factors: WordStimulus
```

7. Change the “RightKey” and “WrongKey” attributes of the “Nonword” level entry:

```
RightKey: @"Nonword Keys"
WrongKey: @"Word Keys"
```

Your “Nonword” entry should now look like this:

```
Nonword::
  Stimulus: FactorAttrib(NonwordStimulus)
  RightKey: @"Nonword Keys"
  WrongKey: @"Word Keys"
  Factors: NonwordStimulus
```

Now, everything is linked together. Try changing the key mapping — using the menu items in the **Experiment** menu — and run the experiment.

Using PsyScope’s environment-configuration features, you can customize the menu bar, the Console window, and set up entries for automatic execution. See “Chapter 16. Configuring the User Environment”, p449.

Part 4:

Scripting Reference

Chapter 11. Introduction 317

Chapter 12. PsyScript Reference 319

Chapter 13. Experiment Scripting Reference 357

Chapter 14. Actions and Devices Reference 419

Chapter 15. Trial Manager Technical Reference 441

Chapter 16. Configuring the User Environment 449

Chapter 17. Dialog and Function Extensions 467



Chapter 11. Introduction

This part of the manual provides an exhaustive reference for users of PsyScript:

“Chapter 12. PsyScript Reference” is a definition of the “pure” language PsyScript, independent of the use of PsyScript to define experiments.

“Chapter 13. Experiment Scripting Reference” details PsyScope’s interpretation of a script to obtain an executable experiment. Both Factor and non-Factor formats are discussed.

“Chapter 14. Actions and Devices Reference” is a reference for all attributes, actions, and devices that are available for describing an experiment.

“Chapter 15. Trial Manager Technical Reference” contains technical details about PsyScope’s real-time Trial Manager.

“Chapter 16. Configuring the User Environment” describes how PsyScope’s user environment can be customized through the script.

“Chapter 17. Dialog and Function Extensions” is a reference for all of the dialogs and functions, which are mainly used for configuring the interface and subject tracking.



Chapter 12. PsyScript Reference

12.1 Components of a Script

A *script* is a text file used to define a PsyScope experiment. The syntax of the script is called *PsyScript*. A script is made up of:

- **Entries** – Entries are the basic blocks of information in a script, similar to the records of a database. For example, each object in the graphic environment is implemented as a separate entry in the script. Entries make up most of the script; each entry is defined by its name followed with double colons (e.g., `My Entry: :`). The full definition of an entry is in “12.2 Entries”, p320.
- **Comments** – A comment is a line of text that is ignored by the script interpreter. All comments start with “#” and run until the end of the line. Comments can be embedded within an entry. See also “12.3 Comments”, p324.
- **Modifiers** – Modifiers are high-level instructions to the script interpreter. For instance, the `#include` modifier is used to refer to another script file. Modifiers look like comments, but a modifier is distinguished from a comment by a keyword immediately following the “#” (e.g., `include`). There are only a few modifiers; they are all defined in “12.4 Modifiers”, p324.
- **Section markers** – Section markers also look like comments; they serve only to organize a script into different sections. For instance, when you create a new template in the graphic environment, it defines the new template entry in the “Template Definitions” section of the script. Section markers always start with “#>”, and *cannot* be embedded within an entry. See also “12.5 Section Markers”, p326.

As an example, here is a micro-script:

```
#PsyScope 1.0
Experiments:: HelloWorld
#> Experiment Definitions
HelloWorld::
    Events: HWEvent
#> Event Definitions
# The only event
```

```
HWEvent::  
  # Here is the stimulus  
  Stimulus: "Hello World"
```

There are three entries here: “Experiments”, “HelloWorld”, and “HWEvent”. #PsyScope is a modifier. #> Experiment Definitions and #> Event Definitions are section markers. # The only event and # Here is the stimulus are comments.

12.2 Entries

Entries are the basic unit of information within a script. Each entry is defined by giving it a name followed by double-colons, and then defining the *content* and *attributes* of the entry.

12.2.1 Entry Content and Expressions

The *content* of an entry is made up of scripting *expressions*. The result of evaluating these expressions is the entry’s *value*.

An expression can be:

- A simple *literal* text string — quoted or unquoted — or a number (e.g., "Hello World", Hi, 6). See also “12.6.1 Literals”, p326.
- A *function call*. There are built-in primitive functions, and more functions can be defined within the scripting language. Strcat>Hello " " World) is an example of calling a built-in function. See also “12.6.2 Function Calls”, p327.
- An *operation sentence*. There are obvious operators — such as + for addition: 2 + 3 — as well as operators that work on scripting constructs — such as @: @TheEntry. Operators can be *unary* (requiring one operand) or *binary* (requiring two operands). All operators are built-in. See also “12.6.3 Operation Sentences”, p328.
- A parenthesized group of other expressions; e.g., (2 + 3 "Hello World" @TheEntry).

When an expression is *evaluated*, it can return one, many, or zero results. The when the expression 2 + 3 is evaluated, it returns the value 5. Values which are obtained by evaluating expressions are also called *tokens*.

The content of an entry is specified immediately after the double-colons in an entry definition. The content of an entry can be empty, or it can be arbitrarily long.

For example, here is an entry with two expressions in its content:

```
TheEntry:: 2 + 3 "Hello World"
```

12.2.2 Attributes

The *attributes* of an entry are sub-entries that the entry owns. Attributes are defined within an entry using names followed by single colon. For example, here is an entry with two attributes, “Mickey Mouse” and “GarfieldTheCat”:

```
CartoonEntry::
  MickeyMouse: "Walt Disney"
  GarfieldTheCat: "Paws, Inc."
```

Attributes are entries in their own right. Each attribute has a content, which contains expressions for the value of the attributes. When an attribute has its own attributes, they are specified using “:>”:

```
CartoonFaceEntry::
  MickeyMouse: "Walt Disney"
    Ears:> Round
    Nose:> Round
  GarfieldTheCat: "Paws, Inc."
    Ears:> Triangular
    Nose:> Round
```

After the “:>” level, further sub-attributes are specified by adding more “>”s:

```
DetailedCartoonEntry::
  MickeyMouse: "Walt Disney"
    Ears:> Round
      Colors:>> Black
    Nose:> Round
  GarfieldTheCat: "Paws, Inc."
    Ears:> Triangular
      Colors:>> Black Orange
      ColorArrangement:>>> Stripes
    Nose:> Round
```

The collection of all attributes belonging to an entry is called the entry’s *attribute block*. When the script interpreter is searching for an attribute in an entry, it looks at attribute names in the order that they are listed in the attribute block.

It is syntactically allowed to have two attributes with the same name within an entry, but only the first instance can be found with the attribute name (in a reference; see “12.2.4 References”, p323). Processes external to the script interpreter can sometimes reach the second instance by index (e.g., a process might ask for the “fourth” attribute of an entry), but there is no way to reach the second instance within PsyScript.

12.2.3 Entry Syntax

The entries which are defined with double-colons are called *global entries*. Attributes, sub-attributes, etc. are all entries, but they are not global entries.

Each global entry in a script must begin on a new line. Trailing and leading tabs and spaces in a global entry name are ignored (unless they are included in quotes in the entry name).

The double-colons — which signal the beginning of the entry’s content — must be on the same line as the name.

A global entry’s name should be unique; if two global entries are defined using the same name, the one that is defined first will always be used (because global entries are referenced by name).

A global entry definition is terminated by one of the following:

- The beginning of another global entry.
- A file inclusion modifier (see “12.4.2 #include and #winclude”, p325).
- A section marker (see “12.5 Section Markers”, p326).
- The end of the script file.

Attributes of a global entry are defined recursively: an attribute begins with the name followed by a single-colon and ends with the start of the next attribute. Unlike global entries, multiple attributes may be specified on a single line. If an attribute has spaces in its name, the name must be quoted. (The single colon is placed *outside* the quotes.)

All expressions between a global entry’s double-colons and its first attribute (or end of the entry if no attributes are specified) are the entry’s content.

Tokens in the content of an entry are separated by blanks. Blanks may occur within an operation sentence or in the parameter area of a function call, but *may not* appear between a function name and the opening parenthesis of the function’s parameter list.

The blank characters are space, tab, return, newline, and comma. Blanks are ignored unless they are within quotes.

When a binary operator (i.e., an operator that takes two operands) is being used, blanks must be used either on both sides of the operator or on neither side. Blanks should not be placed between a unary operator and its operand.

For a literal expression to contain a blank, it must be quoted; e.g, "Hello World". Smart quotes are not recognized; always use straight quotes. (Smart quotes look like this: “ ”, straight quotes look like this: " ".)

Entry, attribute, and function names are case-insensitive. String comparisons (with the == operator, for example) are also case-insensitive.

Comments may be placed anywhere in the script, as long as no other syntax rules are violated when the comments are replaced by blanks. Comments cannot, of course, be placed within quoted strings.

There are three special separator symbols: =>, @> and ;. These symbols are treated as regular literal tokens, but with a special parsing feature: they always have implicit blanks around them. This means that they will always count as separate expressions when they are

not quoted. For example, if `a=>b` (with no spaces) is in the content of an entry, it counts as three separate expressions: `a`, `=>`, and `b`.

12.2.4 References

A *reference* is a “pointer” to an entry; it is a token that is the “name” of an entry in the script. (The referenced entry does not have to be a global entry.)

In the simplest case, a reference is literally the name of an entry, but this only works for global entries. For example, the expression `Experiments` could be used as a reference to the “Experiments” entry in the script.

The most common use of a reference is to get a value stored in the entry; the basic reference operator is `@`, which returns the value of the referenced entry’s content.

For example, given this entry definition in the script:

```
SomeEntry:: 2 + 3
    Greeting: "Hello World"
    Numbers: 1 2 3
    Prime:> 2 3
```

Then, the value of `@SomeEntry` is 5.

To reference an attribute of a global entry, you concatenate the global entry name, “>>”, and the attribute name. For example, `"SomeEntry>>Greeting"` references the “Greeting” attribute of “SomeEntry”. The value of `@SomeEntry>>Greeting` is “Hello World”.

To reference sub-attributes, you keep using “>>” to build a path to the attribute. Thus, `"SomeEntry>>Numbers>>Prime"` is a reference to the “Prime” sub-attribute of the “Numbers” attribute of the global entry “SomeEntry”.

The `@` operator returns *all* of the tokens in the content of an entry. For example, the value of `@SomeEntry>>Numbers` is three tokens: 1, 2, and 3.

When a reference is generated by a sublist or inline operation (see “12.8.7 Sublisting”, p334 and “12.9 Inline Entries”, p335), there is no way to write the reference in string form. If the script interpreter is forced to translate such a reference into a string (e.g., through the Evaluator; see “12.15 The Evaluator”, p356), it can produce an output string for you to read, but this output cannot be passed back to the script interpreter as a reference.

When you use “>>” within quotes, the script interpreter knows how to decompose the string value to make an attribute reference. You can also reference an attribute using the `>>` operator; in the operator form, `>>` is out of the quotes (if any) in the expression, e.g., `SomeEntry>>Greeting`. The `>>` operator can be used on reference tokens that do not have a string equivalent. (See also “12.14.1.1 Reference Operations”, p350.)

The construct `@entry>>attrib` is very common, so it has a short hand which uses the `->` operator: `entry->attrib`. Like `>>`, `->` operator can work with non-string references.

12.2.4.1 `THIS` and `OWNER`

`THIS` is a keyword which evaluates — in the simple case — as a reference to the *global* entry in which it appears; the context of `THIS` may change when it is used in an entry that is incorporated into an inline entry (see “12.9.6.1 `THIS` and Inline Entries”, p341).

`OWNER` is a similar keyword that can only be used in the content of an attribute (or inline entry). It evaluates as a reference to the entry (possibly non-global) that owns the attribute (or inline entry) in which the `OWNER` keyword appears.

12.3 Comments

A comment may be used most anywhere in the script; comments start with a “#” and end with the end of the line.

Comments cannot appear within entry names, within anything that is quoted, or anywhere such that replacing the comment with spaces violates some other syntactic rule (see “12.2.3 Entry Syntax”, p321). Comments cannot be placed within modifiers or section markers.

12.4 Modifiers

Modifiers change the way the script interpreter reads or evaluates the script (much like compiler directives in C). In general, the placement of a modifier is highly constrained, and modifiers placed out of context are ignored.

Modifiers look like comments in the script: they all start with “#”. A modifier is distinguished from a comment because one of a small set of keywords will appear after the “#”, with no intervening spaces. Some modifiers take a parameter after the keyword; if additional text is added after the parameter (and before the end of the line), it is usually ignored.

The available modifiers are listed and described below.

12.4.1 `#PsyScope`

The most important modifier is the `#PsyScope` modifier, which must appear as the very first line of every script (with no spaces or newlines before it). The `#PsyScope` modifier should be followed by the version of the script interpreter to be used with the script. The current version is 1.0.

For example, scripts that run with `PsyScope 1.0` will all have this first line:

```
#PsyScope 1.0
```


Note: It is a good idea to leave a few spaces after the version number in the #PsyScope modifier; this will allow future automatic script-updaters to easily change the version number if it gets longer (e.g., 1.0.1).

12.4.2 #include and #winclude

The #include and #winclude modifiers are used to *include* an external file in the script. These modifiers must appear at the beginning of a new line, and may not occur within an entry definition (see “12.2.3 Entry Syntax”, p321).

#include takes one parameter: the name of a file. The specified file is read as a script and used as if the #include modifier were replaced with the file’s contents. For example, to include a file name “WordList” (that defines entries to be used as word stimulus), you would write:

```
#include "WordList"
```

If the #winclude (“writable include”) modifier is used, then changes made to entries in the included file (e.g., through script functions) will be saved when the top-level script is saved. Otherwise, the included file is “read-only”. (When an included file is “read-only”, the included entries can still be modified in PsyScope’s memory after the file is loaded; however, these changes cannot be written back to the file.)

The #include modifier can also be used to include standard script files stored in an open resource file (including PsyScope’s resource fork). “SubjectInfoLib” is an example of a resource-stored script.

File names specified for the #include modifiers are subject to the path name conventions described in “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215.

12.4.3 #inherit and #noinherit

These modifiers can be used in the attribute block of an entry: attributes following a #noinherit modifier and preceding an #inherit modifier in the attribute block will never be used as inherited attributes (see “12.11 Inherited Attributes”, p344).

If an #inherit modifier precedes any #noinherit modifiers in the attribute block of an entry, an initial #noinherit modifier is assumed.

Note: Strictly speaking, the “inheritance” described here is different from the inheritance of attributes in Factor format. However, these modifiers still affect Factor format’s inheritance, as well.

12.4.4 #NoIncludeStdLib

This modifier prevents PsyScope's standard library of standard scripting tools to be automatically included in the script. This modifier must be placed by itself on a new line. See also "16.7 PsyScopeStdLib", p461.

12.5 Section Markers

Section markers serve to organize the script by denoting the types of entries that are within a section of the script. PsyScope's graphic environment uses markers for adding new entries to an appropriate place in the script; for example, new template definitions are added to the section marked `#> Template Definitions`. Section markers may also be useful to the scripter for organizing a script.

Section markers always start at the beginning of a line, and must start with "#>". Section markers may not be placed within an entry definition (see "12.2.3 Entry Syntax", p321).

Section markers are also recognized by the `::` pop-up menu in PsyScope's text editor. See "Part 2: Graphic Environment Reference, 7.3.3 Action Bar", p261.

12.6 Operators and Functions

As explained in "12.2.1 Entry Content and Expressions", p320, a PsyScript expression is either:

- a literal,
- a function call,
- an operation sentence,
- or a parenthesized group of other expressions.

12.6.1 Literals

Any text string that does not contain blanks, operators, or parenthesis is a *literal*. Also, any quoted string is a literal.

Any string which contains blanks in it — such as "Hello World" — must be quoted; otherwise, the blank-separated strings count as multiple expressions. Besides the normal quote-marks, strings may also be quoted with curly braces (`{}`); this allows literals which contain quote-marks: `{This literal contains a "quoted" word}`.

All literals in PsyScript are text — even numbers. In the expression `2 + 3`, the `+` operator knows how to convert the strings `2` and `3` to the numbers `2` and `3`. Even though period (`.`) is an operator, decimal numbers — such as `0.5` — can be specified without quotes.

All strings have a *Boolean value* equivalent to `True` or `False`. `0` and `False` (with any capitalization) are the only values equivalent to `False`; everything else is equivalent to `True`. Boolean operators always return the values `True` or `False`.

12.6.2 Function Calls

A *function call* is of the form `FunctionName(parameter1 parameter2 ...)`. There cannot be a blank between the function name and the opening parenthesis.

Each function may take a certain number of input parameters; expressions within the parentheses of a function call specify the parameter values. For example, the function `Strcat()` takes a list of strings for its parameters, and all the strings are put together into one string; thus, `Strcat("Hello " " World")` evaluates to `"Hello World"`.

The values passed to the function are not the parameter expressions themselves, but the result of evaluating the expressions; this distinction is important, since a single expression can evaluate to zero or multiple values. All parameter expressions are evaluated before a function is called.

For example, given this entry definition:

```
HelloWorldParts:: Hello " " World
```

then `Strcat(@HelloWorldParts)` also evaluates to `"Hello World"`. Only one parameter expression was given, but the function received three parameters.

The list of all built-in operators is given in “12.14 Script Operators and Functions Summary”, p349. Additional functions can be defined by a scripter; see “12.9.6 Function Definitions”, p340.

Built-in functions which take only one parameter can actually take any number of parameters; the function will be applied to each parameter and all of the results will be returned. This generalization does not apply for script-defined functions, but other distributive effects can be obtained by combining script-defined functions with the distributing operators `**` and `*!` (see “12.6.3.1 Distributivity”, p328 and “12.14.1 Operators and Functions”, p350).

12.6.2.1 Exceptions to the Rules

There are a handful of built-in functions which break the usual rules:

`Div()`, `NthChar()`, `Power()`, `StripFrontChars()`, and `StripEndChars()` are actually binary operators that are implemented function-style. This means that the parameter

expressions are treated as the operand expressions of a binary operator (see “12.6.3 Operation Sentences”, p328 below).

`If ()` does not evaluate all of its parameter expressions automatically. It always evaluates enough expressions to get the value of its first parameter; if this is `True` (see “12.6.1 Literals”, p326), then `If ()` evaluates enough to get (and return) the second parameter; otherwise, it evaluates only what is necessary to get (and return) the third parameter.

12.6.3 Operation Sentences

An *operation sentence* is made by applying an operator to operand expressions. Operators are always one- or two-character symbols, such as `+` and `->`, and the operands are placed before or after the operator.

Some operators are *unary*, i.e., they take only one operand. Most unary operators are placed before their operand; e.g., the `@` operator in `@EntryName`. There should be no blanks between a unary operator and its operand.

Binary operators are always placed between their parameters; e.g., the `+` operator in `2 + 3`. Blanks can be used within binary operation sentences, but they should be used consistently: if blanks are placed between a binary operator and its first parameter, blanks should also be placed between the operator and its second argument. (The number of blanks on each side could be different; all that matters is whether there are any blanks at all.)

The operands of an operator are expressions. When an operand expression evaluates to multiple tokens, the distributivity property of operators is used (see below).

The list of all built-in operators is given in “12.14 Script Operators and Functions Summary”, p349.

12.6.3.1 Distributivity

A property of all operators is *distributivity*. Distributivity means this: if multiple values are given in the place of an operand, the operation will be performed once for each of the given values. Multiple values may appear as operands because the operands are specified as expressions, and a single expression can evaluate to more than one value.

A parenthesized list of values counts as a single expression; `(2 3)` is a single expression that evaluates to two values: 2 and 3. Thus, in the expression `(2 3) + 3`, two values are given for the first operand of the `+` operator; the values of this expression will be 5 and 6, in that order.

If multiple values are given for both the first and the second operand of a binary operator, the expression is evaluated by taking one value for the first operand and evaluating for each of the possible second operand values, then using the next first operand, and so on. Thus `(2 3) + (10 30)` evaluates to 12, 13, 32, and 33.

Distributivity also applies when an operand evaluates to zero expressions; `() + 3` is a valid expression which evaluates to nothing (i.e., an empty list of values).

12.7 Attribute Block Reference

Just as the `@` operator can be used to refer to the content line of another entry, “`@`” can also be used as an attribute name to refer to the entire attribute block of another entry; this is called an *attribute block reference*. The content of an “`@`” attribute should be a reference to another entry in the script. (Multiple references can be specified.)

When the interpreter is searching for an attribute (see “12.2.2 Attributes”, p321) and it encounters an attribute named “`@`”, it looks in the referenced entry for the attribute. In this way, the referenced entry’s attributes are “inserted” where the “`@`” attribute occurs.

For example, given these entry definitions:

```
EntryOne::
  A: 1
  B: 2
  C: 3

EntryTwo::
  B: 4
  @: EntryOne
  C: 5
```

Then, the value of `EntryTwo->A` is 1. The value of `EntryTwo->B` is still 3, since the “`B`” attribute is specified in “`EntryTwo`” before the “`@`” attribute. However, `EntryTwo->C` is 3, since the “`C`” attribute of “`EntryTwo`” follows the “`@`” attribute.

The “`@`” attribute may be used any number of times within an entry’s attribute block.

There is a technical constraint that was added to speed up “`@`” parsing: multiple references can be specified in the content of a single “`@`” attribute, but each reference must be in a separate expression.

12.8 Lists

Note: Lists as described here are not the same as “lists” in the graphic environment. These lists are more primitive and are purely scripting concepts. Graphic environment lists are described in “Part 2: Graphic Environment Reference, 5.7 Factors and Lists”, p129.

A *list* is an entry that is being used in the context of special scripting functions. There are a few special attributes that a list can have that are used by these functions, but those at-

tributes are optional. Any entry (with at least one token in its content) may be used as a list, and anything that is defined as a list is automatically an entry.

The list functions treat each token of an entry as a separate *item* in the list. The *size* of the list is the number of items it has.

List items are indexed starting with 1; this index can be used with certain operators to obtain a particular item in a list (e.g. the `.` operator). Indices are always used modulo the size of the list, so that the list “wraps around”. Thus, the 5th item in a 12-item list can be indexed by 5, 17, or -7.

It is important to note again the distinction between a *token* and an *expression*. The tokens of an entry are the result of evaluating all of the expressions in the entry’s content. The items of a list are the tokens of the entry, not its expressions.

Here is a list with five items:

```
TheList:: apple 1 Strcat(Hello " " World) (1 2) + 10
```

The `.` (“dot”) operator gets an indexed item from a list. Thus, the value of `TheList.3` is “Hello World”. The dot operator recognizes modulo indexing, so `TheList.8` is the same as `TheList.3`.

12.8.1 Accessing a List

Lists are generally used to contain a set of possible values for some other purpose. *Accessing* a list refers to the built-in process of picking a single item from a list, and marking the item as used.

There is a *checklist* associated with each list, with one place on the checklist for each item in the list. When the list is accessed, one item is marked off on the checklist and made to be the *current item* for the list.

A list is accessed by using the `Access()` function, which takes as its parameter a reference to a list. The return value of `Access()` is the new current item, i.e., the token which was selected by the access. To get the current item of a list without re-accessing it, use the `GetCurrent()` function.

When a new current item is selected, it will always be one that is not yet marked off in the checklist. When all of the items have been marked off, the checklist is cleared and the marking-off starts over.

A checklist can be cleared by *resetting* the list. The `Reset()` function takes as its parameter a reference to a list to reset; its return value is `NULL`. The `ResetAll()` function takes no parameters and resets all of the lists in the script.

For example, using this entry:

```
AnotherList:: a b c d e
      AccessType: Random
```

`Access(AnotherList)` might return `b` (the item `Access()` picks depends on the list attributes, which are discussed below). Immediately evaluating `GetCurrent(AnotherList)` would return `b` again, but `Access(AnotherList)` will return a different value, possibly `d`.

12.8.2 Access Type

The *access type* of a list determines the way in which a new current item is selected when a list is accessed. An access type is specified in an “AccessType” attribute of the list’s entry; it can be one of three types: *sequential*, *random*, or *incremental*. Sequential accessing is used when no access type is specified

If a list’s access type is *sequential*, a new current item is chosen to be the one following the previous current item. The first item in the list is used as the first current item. If, in this sequential process, an item is encountered that is already marked-off (by accessing the list in a non-sequential manner), it is skipped and the next one is used (if it is not already marked-off, too). Sequential accessing is specified by setting the value of the “AccessType” attribute to `Sequential` (or simply `Seq`).

For example, given this entry definition:

```
TheList:: a b c d e
      AccessType: Sequential
```

The first evaluation (after each time the script is loaded) of `Access(TheList)` will return `a`. The second evaluation will return `b`, and so on, until `e` is returned. Then, another evaluation will return `a` again.

In *random* accessing, a new current item is selected randomly from the items that have not been marked-off (i.e., sampling without replacement). Random accessing is specified by setting the value of the “AccessType” attribute to `Random` (or `Rand`).

If “TheList” in the previous example is given the random access type, then the first evaluation of `Access(TheList)` could return anything. However, within the first five evaluations, all five values — `a`, `b`, `c`, `d`, and `e` — will be returned exactly once. The same will be true of the second five evaluations, and so on, but each set of five evaluations can have the items in a different order.

In *incremental* accessing, the value of the “AccessType” attribute is an integer n . The new current item is selected to be the n th item after the previous current item; the first current item is always the first item. Sequential accessing is thus a special case of incremental accessing.

If “TheList” in the previous example is given 2 as the value of “AccessType”, the first evaluation of `Access(TheList)` will return `a`. The next evaluation would return `c`, then `e`, `b`, and `d`.

A special case is introduced in incremental accessing: when a item is encountered in the incremental process that is already marked-off, the next n th item is used. If it keeps trying each n th item and gets back to trying the previous current item, the checklist is “reset”, even though items in-between the n ths may not have been marked-off. This is because list resetting is done on the basis of whether there are any items unmarked that can be selected based on the access type; no current item may be selected by making jumps of n , so the list is “reset” — but the whole list is not reset, only the items in the n -jump set. This is closely related to the concept of sublistings; see also “12.8.7 Sublisting”, p334.

12.8.3 Linking

Two lists of the same size can be made to use the same checklist by *linking* the lists. If a list has a “Link” attribute whose value is a direct reference to some other list, instead of creating its own checklist it will use the referenced list’s checklist. Thus, accessing one list will have the affect of accessing the other list, and they will both have the same *current item index* (although the *current item* will depend on what is in each list at that index).

Two lists which are linked can still be accessed with different access types. Be sure to only link in one direction: if “List A” is linked to “List B”, “then “List A” is using “List B”’s checklist, so “List B” cannot be linked to “List A”. The direction in which the link is specified is never important.

12.8.4 Weights, Multiple, Grip

There are three attributes used by lists which change the way the list is accessed:

The “Weights” attribute assigns to the i th item in the list an integer weight n_i , such that the item must be accessed (i.e. marked off in the checklist) n_i times before the list is reset. The value of the “Weights” attribute should be a list of integers, one for each item in the list. If too few or too many weights are supplied, a warning will be reported.

For example, accessing this entry:

```
WeightedList:: a b c
Weights: 1 2 1
AccessType: Sequential
```

will return a, b, b, and c.

The “Multiple” attribute is a shorthand for assigning the same weight to each item in the list. Its value should be a single integer.

The “Grip” attribute acts much like the “Multiple” attribute, except that when an item is accessed, the next $n - 1$ accesses of the list (where n is the grip value) will return the same item, regardless of the access type of the list. Note that, in sequential accessing, “Grip” and “Multiple” specifications are indistinguishable.

All three of these attributes change the effective size of the list as far as accessing is concerned, but the actual list size is not changed. The n th item in the unweighted list will still be the n th item when the list is weighted; weights and grips only effect accessing and the way items are marked off the checklist. The attributes may also be combined in any way.

Consider these three lists:

```
ListOne:: a b
        Multiple: 2
        AccessType: Random

ListTwo:: a b
        Grip: 2
        AccessType: Random

ListThree:: a b
        AccessType: Random
```

These lists are exactly the same, except that “ListOne” uses a “Multiple” attribute while “ListTwo” uses a “Grip” attribute and “ListThree” uses no special attributes.

Within four access of “ListOne”, a will be returned twice and b returned twice. The pattern could be a a b b, b b a a, a b a b, b a b a, a b b a, or b a a b.

Four accesses of “ListTwo” will also return two a’s and two b’s, but the pattern could only be a a b b or b b a a, because the second access has to return the same value as the first access.

Four accesses of “ListThree” returns either a b a b, b a b a, a b b a, or b a a b; because there are no weights or grips, “ListThree” must be exhausted within each pair of accesses.

12.8.5 Offsets

By default, the indexing of items in a list starts with the first item — which gets the index 1 — and continues to the end of the list. The starting position for indexing may be changed with the “Offset” attribute. If the value of this attribute is an integer n , indexing will start with the n th item in the list and wrap around. The modulo nature of list indexing will remain intact; only the starting position for indexing has been changed.

For example, in this list:

```
OffsetList:: a b c
            Offset: 2
```

the value of `OffsetList.1` is b.

12.8.6 SaveCurrents

By default, a list “remembers” just one current item — the last item marked off in the checklist — for use with `GetCurrent()`. If a list has a “SaveCurrents” attribute with an integer value n , the list will also remember the $n-1$ previous current items.

Just as `GetCurrent(List)` returns the current item for “List”, `GetPrevCurrent(List 2)` returns the item of “List” that was current before the present current item. However, a “SaveCurrents” value of at least 2 must have been set for “List”, or an error will be reported. (`GetPrevCurrent(List 1)` is equivalent to `GetCurrent(List)`.)

If the value of “SaveCurrents” is larger than 4, a warning may be given in regard to version tracking.

12.8.7 Sublisting

A *sublist* is a list made up of items that belong to another list in the script. Sublists are “virtual” entries; there will be no global entry which contains just the items for the sublist; the items are merely referenced from a structure that is hidden within the script interpreter.

The list to which the items actually belong is called the *parent list*. When you access a sublist, the items are marked off from the parent’s checklist and the current item is set in the parent list. Sublists also use the list attributes of the parent list (e.g., “AccessType”, “Weight”). If a sublist is created from another sublist, the new sublist’s parent list is the same as the original sublist’s parent.

The most general way to create a sublist is by using the `Sublist()` function. The input parameters of this function specify a list and a range of items, and the return value is a reference to the sublist.

The first parameter of the `Sublist()` function is a reference to the parent list, the second parameter is the index of the item to start with in the parent list, and the third parameter is the number of items the sublist should have.

For example, given this list definition:

```
BigList:: 1 2 3 4 5 6 7 8 9 10 11 12
```

`Sublist(BigList 3 4)` creates a sublist equivalent to

```
Sublist:: 3 4 5 6
```

Thus, `Access(Sublist(BigList 3 4))` will return 3 the first time, 4 the second, and so on. However, the sublist is never actually made into a global entry, like “Sublist” above.

Sublist references can be used anywhere that a list reference can be used, including in `Access()`, `GetCurrent()` and `Sublist()`; sublist references can also be used as generic entry references. For example, `@Sublist(BigList 3 4)` evaluates to 3, 4, 5, and 6.

Accessing a sublist causes the checklist and current item of the parent list to change. When all of the items of a sublist have been exhausted in accessing the list, only the check marks for the items in the sublist will be reset.

An optional fourth parameter can be given to `Sublist()` which specifies a step size for selecting items from the parent list. Using the previous definition of “BigList”, `Sublist(BigList 3 3 5)` creates a sublist equivalent to

```
Sublist:: 3 8 1
```

There are three specialized sublisting functions: `Row()`, `Column()`, and `Map()`. The `Row()` function takes as its second and third parameters the number of rows in the parent list and which of those rows should be used as a sublist. The `Column()` function does the same for columns.

For example, `Row(BigList 3 2)` creates a list equivalent to

```
Sublist:: 5 6 7 8
```

and `Column(BigList 3 2)` creates a list equivalent to

```
Sublist:: 2 5 8 11
```

The `Map()` function is a variation of the `Row()` function. It takes as its second parameter the name of another list, called the *mapping list*. The number of rows in the parent list is taken as the list size of the mapping list; the row to be used is taken as the current item index of the mapping list. For example, suppose that this list is in the script:

```
Mapper:: a b c
```

and that `GetCurrent(Mapper)` at the moment returns `c`. Then the evaluation of `Map(BigList Mapper)` creates a list equivalent to

```
Sublist:: 9 10 11 12
```

Although `Map()` may seem a bit arcane, it is a powerful method of mapping single items from one list to groups of items in another.

12.9 Inline Entries

An *inline entry* is a scripting construct in which a temporary, non-global entry is created within the content of another entry. Typographically, inline entries are created with square brackets or parentheses. A list of tokens and attribute definitions within these brackets make up the content and attribute block of the inline entry. When square brackets are used, a reference to the inline entry is produced. (Parentheses are slightly more complicated and will be addressed further below.)

Each time an expression describing an inline entry is evaluated, a new entry is created. The content of the new entry is generated by immediately evaluating the content expressions in the inline entry’s definition, and using the resulting tokens as the content value. This is dif-

ferent from global entries, which remember their content expressions and re-evaluate them to get content values.

Here is an entry that defines an inline entry named “Mouse”:

```
EntryWithInline:: Mouse[Click X: 100 Y: 100]
```

The content value of the “Mouse” inline entry is `Click`. “Mouse” has two attributes: “X” and “Y”. Evaluating `@EntryWithInline` will return a reference to (an instance of) “Mouse”.

Since an inline entry is a virtual entry, it does not have to be named. This entry defines an equivalent inline entry, but the inline entry’s name is “”:

```
EntryWithInline:: [Click X: 100 Y: 100]
```

If a name is provided, it must appear before the opening square-bracket without intervening blanks. An inline entry name does not have to be a literal; it can be any expression, as long as there are no blanks between the expression and the opening square-bracket. The naming expression should have only one value; multiple values are ignored.

For example:

```
InlineName:: Mouse
EntryWithInline:: (@InlineName)[Click X: 100 Y:100]
```

This defines an inline entry in “EntryWithInline” that is equivalent to the earlier definition with the name “Mouse”.

12.9.1 Inline Entries vs. Regular Entries

The effects of the difference between regular and inline entries are not obvious. For example, applying `@` to an instance of the “Mouse” inline entry (from the previous example) acts as expected: the value of `@@EntryWithInline` is `Click`.

Here is an example that will demonstrate a difference between regular and inline entries:

```
List:: a b c
RegularEntry:: Access(List)
EntryWithInline:: AccessInline[Access(List)]
```

“RegularEntry” and “AccessInline” have the same essential definition: both contain a single content expression which accesses “List”. However, since evaluating `Access()` has a side-effect, there can be a difference between using a reference to “RegularEntry” or to “AccessInline”.

Evaluating `@RegularEntry` will cause `Access(List)` to be evaluated, returning `a`. Evaluating `@@EntryWithInline` will do the same thing: `@EntryWithInline` causes an “Access-

Inline” entry to be created, and `Access(List)` is evaluated, setting the content of the “AccessInline” instance to `a`; then, `@` is applied to the inline reference returned by `@EntryWithInline`, so that `a` is returned as the final value. There was no difference between using `RegularEntry` or using `@EntryWithInline` in this case.

However, consider evaluating `@(3 ~ RegularEntry)` versus `@(3 ~ @EntryWithInline)`. The `~` operator takes a count for its first operand, and returns its second operand that many times. Thus, `3 ~ RegularEntry` evaluates to `RegularEntry RegularEntry RegularEntry`; the `@` operator distributes over the three references —evaluating the content of “RegularEntry” three times — and returns `a b c`. `3 ~ @EntryWithInline` returns three references to the same instance of “AccessInline” — the one created by evaluating `@EntryWithInline`; `@` distributes over these references, and returns `a a a!`

In this last case, the difference came from the fact that `Access(List)` was evaluated only *once* when the inline instance was created, and then this *value* was remembered for the content of the instance. Then, the content of the instance was read three times; this content did not contain the expression `Access(List)`, only the value `a`.

The reason that inline entries were made to behave this way has to do with the interaction between inline entries and script-defined functions, which are described in “12.9.6 Function Definitions”, p340.

12.9.2 Attributes of Inline Entries

Attributes are not treated the same as content values in an inline entry. While the content of an inline entry is a temporary set of values created just for the instance, all instances of an inline entry use the same set of attributes, and these attributes are permanent.

For example, consider this entry definition:

```
TestEntry:: Switches[Light: Off TV: Off]
TurnOnLight:: (@TestEntry)->Light = On
BadSwitch:: Light[Off]
BadTurnOnLight:: @@BadSwitch = On
```

When `@TurnOnLight` is evaluated, `@TestEntry` returns a reference to an instance of “Switches”. This inline entry has a “Light” attribute, which is assigned a new value by `(@TestEntry)->List = On`. In the script, the definition of “TestEntry” becomes:

```
TestEntry:: Switches[Light: On TV: Off]
```

However, evaluating `@BadTurnOnLight` will cause an error. When `@BadSwitch` instantiates “Light”, the expression `Off` is evaluated into a token value; even though the original `Off` in `Light[]` appears to be an assignable literal, the content of the instantiated “Light” is a value that cannot be changed through assignment.

12.9.3 Incorporating a Global Entry

A special property of inline entries is invoked when an inline entry is given a name that is also the name of a global entry. When this happens, a reference — referencing the global entry’s content line — is added to the end of the content line of the inline entry, and an attribute block reference — referencing the global entry’s attribute block — is added to end of the attribute block of the inline entry. This has the effect of incorporating the global entry’s definition into the inline entry.

While this rule may seem strange, it is actually the most powerful feature of inline entries and leads to a entry-definable functions (see “12.9.6 Function Definitions”, p340). This feature is also useful for changing list parameters for sublistings (see “12.9.4 Inline Entries and Lists”, p338), or for assign attribute values in inheritance (see “12.11.1 Inheritance and Token Reference Inline Entries”, p346).

Here is an example of an inline entry incorporating a global entry:

```
SomeEntry:: b c
    X: 100
    Y: 50

EntryWithInline:: SomeEntry[a]
```

Because the inline entry is given the name “SomeEntry”, it is as if it were defined like this (without the special incorporation rule):

```
EntryWithInline:: SomeEntry[a @SomeEntry @:SomeEntry]
```

Thus, `@EntryWithInline` will return `a b c`, and `(@EntryWithInline)->X` returns 100.

Giving an inline entry the same name as a global entry is a special case; in fact, you can use any reference for the “name” of an inline entry, and the referenced entry will be incorporated into the inline entry. See “12.10 Using a File as an Entry”, p343 for an example use of this feature.

12.9.4 Inline Entries and Lists

Inline references are often useful for list manipulations. One simple application is to use an inline entry for a list instead of creating a separate entry for a small list.

For example:

```
Item:: Access(List)

List:: a b c d
    AccessType: Random
```

can be formed equivalently by

```
Item:: Access([a b c d AccessType: Random])
```

This shorthand can often be more clear than the separate-entry method. However, when an inline entry is used in this way, there is a question as to what entry is owning the checklist, because new instance of the inline entry is created each time the inline entry definition is evaluated.

The problem is solved by a convention of automatically list-linking (see “12.8.3 Linking”, p332) inline instances:

- If an inline entry incorporates a global entry and does not define any content in addition to the global entry’s content, the inline instance is linked to the global entry.
- Otherwise, a special permanent instance of the inline entry is kept within the scripting language; all instances of the inline are linked to this special instance.

These conventions let us use the shorthand for defining lists as demonstrated in the example. They also give us a way of overriding list attributes for sublisting.

When a sublist is created (using, for example, the `Sublist()` function), the list attributes of the sublist are the same as the attributes of the parent list. Through the automatic linking of inline entries to global entries, you can set the list attributes of a sublist to be different from the parent list by: creating an inline entry which incorporates the parent list, and then specifying list attributes within the inline entry.

For example:

```
List:: a b c d
      AccessType: Random

PartOfList:: Row(List[AccessType: Sequential] 2 2)
```

The `Row()` function in “PartOfList” creates a sublist of the original “List”, but overrides the “AccessType” attribute. Evaluating `Access(@PartOfList)` will perform a sequential access on the second half of “List”.

12.9.5 Token Reference Inline Entries

The square brackets of an inline entry definition can be replaced by parentheses; this changes the way in which the defined inline entry is used for evaluation.

In the simple case, replacing square brackets with parentheses is the same as apply the `@` operator to the reference returned by evaluating the square-bracket definition.

For example, consider this entry definition:

```
EntryWithInline:: [a b c]
```

Evaluating `@EntryWithInline` will return a reference to an instance of an inline. Changing the square brackets to parentheses:

```
EntryWithInline:: (a b c)
```

has the same effect as:

```
EntryWithInline:: @[a b c]
```

In either of these two cases, evaluating `@EntryWithInline` will return `a b c`.

If you consider what this means in an expression like $(2 + 3) * 4$, the results are exactly what you would expect: the expression `2 + 3`, owned by the inline entry, is evaluated first and returned; the returned value is then multiplied by 4 to get 20. In fact, this is not the way most parentheses are implemented, but it is conceptually consistent to assume that this is how computations are performed.

The complete meaning of parenthesis-form inline entries is more complex. Evaluation of a parenthesis-form inline always starts by creating an inline instance — just as with square brackets — but the value that is extracted from the reference follows these rules:

- If the inline entry has a “Result” attribute, return the value of the attribute.
- If the inline entry has an “ExtResult” attribute, invoke the specified ‘XRES’.
- Otherwise, return the content values of the inline entry.

The first rule is the one which allows script-defined functions (see “12.9.6 Function Definitions”, p340). The second rule allows externally-defined extensions to the scripting language. The last rule makes the standard usage of parentheses consistent with inline definitions (as discussed above); it also has applications for attribute inheritance (see “12.11.1 Inheritance and Token Reference Inline Entries”, p346).

12.9.6 Function Definitions

Consider the following example script definitions:

```
StandardSize::  
    Result: 12  
  
TextSize:: StandardSize()
```

The expression `StandardSize()` in “TextSize” is a token reference inline definition which incorporates the “StandardSize” global entry. According to the rules in “12.9.5 Token Reference Inline Entries”, p339, the value of `StandardSize()` will be taken from the “Result” attribute of “StandardSize”, so that `@TextSize` evaluates to 12.

Of course, it is no accident that `StandardSize()` looks like a function call. Functions are defined in the scripting language by exploiting the properties of inline entries and the `THIS` keyword.

12.9.6.1 `THIS` and Inline Entries

The `THIS` keyword was introduced in “12.2.4.1 `THIS` and `OWNER`”, p324. `THIS` has a special property related to inline entries: When an inline entry incorporates a global entry, `THIS` keywords incorporated from the global entry will now refer to the inline entry.

For example:

```
EntryWithThis:: b c
  X: @THIS

EntryWithInline:: EntryWithThis[a]
```

Evaluating `EntryWithThis->X` will return `b c`. Evaluating `(@EntryWithInline)->X` uses the same “`X`” attribute definition from “`EntryWithThis`”, but `THIS` has been re-scoped, so that the result is `a b c`.

Note, however, that `THIS` statements in the inline entry declaration still refer to the global entry in which the inline entry is defined. For example:

```
EntryWithThis:: b c
  X: @THIS
  Y: d

EntryWithInline:: EntryWithThis[THIS->Y]
  Y: a
```

Evaluating `(@EntryWithInline)->X` will still return `a b c`, because `THIS` in the inline definition still refers to “`EntryWithInline`”.

12.9.6.2 Using `THIS` to Define a Function

Putting together token reference inline entries and the special scoping of `THIS`, we can now write a function that returns the “head” (first element) of an input list:

```
ListHead::
  Result: (@THIS).1

ExampleList:: a b c d e
```

When `ListHead(ExampleList)` is evaluated, an inline instance of “`ListHead`” is built, with a reference to “`ExampleList`” as its content and incorporating the attributes of the “`ListHead`” global entry. According to the rules of token reference inline entry evaluation (see “12.9.5 Token Reference Inline Entries”, p339), the “`Result`” attribute is read. The value of `@THIS` will be the reference to “`ExampleList`”, and then applying the `.1` will get the first item of the list. Thus, evaluating `ListHead(ExampleList)` will return `a`.

Here is a less trivial example that takes a list reference and creates a sublist that is the reverse of the given list (see also “12.8.7 Sublisting”, p334):

```
ReverseList::
  Result: Sublist(@THIS 2~ListSize(@THIS) -1)
```

Functions may be called recursively, though the effective stack for PsyScript is only about 50 calls deep (depending on the size of the heap given to PsyScope). The factorial function can be implemented by the following:

```
Factorial::
  Result: if(@THIS == 0 1 @THIS * Factorial(@THIS - 1))
```

This function will work up to about $35! = 10^{40}$; larger numbers cannot be represented in PsyScript. (The special handling of `if()` is important to the operation of this example; see “12.6.2.1 Exceptions to the Rules”, p327.)

12.9.6.3 Parameter Tags

Writing functions using only the `THIS` keyword can be difficult, and the resulting definition may be difficult to understand. One last facility is provided for function-writing: *parameter tags*.

A parameter tag is an expression of the form ``name`` (``` is the backquote; `name` cannot contain blanks or be quoted). Evaluating a tag directly returns no value. The way in which tags are intersperse with other expressions is not important. All that matters is the relative ordering of tags among themselves.

For example:

```
AbsDiff:: `a `b
  Result: if(a - b > 0 a - b b - a)
```

In this entry definition, there are two tags: “a” and “b”.

When, in the course of evaluating expressions, the script interpreter encounters an unquoted literal, it does not immediately return the literal value as the result. First, it checks the content of the entry that the `OWNER` keyword would refer to; if a tag is found in the content which is named the same as the literal expression, then the result value for the literal will be taken from the content. If no tag is found, the search continues up the `OWNER` hierarchy until a global entry (not just an incorporation) is reached.

The value that is taken from the content line depends on the tag’s position in the list of tags; if it is the first tag, then the first value of the content is returned, and so on. If there are too few values in the content, and error is given to the user.

Using the previous example of “AbsDiff”, consider evaluating `AbsDiff(5 10)`. This will create an instance of an “AbsDiff” inline entry, incorporating the “AbsDiff” global entry. When the “Result” attribute is being evaluated and the expression `a - b` is reached, the “a” and “b” tags will be found. `a` will be replaced with the first value of the content — 5 — and `b` will be replaced by the second value — 10. The result of the function call will be 5.

Parameter tags provide a surprisingly powerful scoping system. Using tags, it is possible to write functions that return functions as their value.

For example:

```
MakeAdder:: `n
      Result: [ `m Result: m + n]
```

The result of evaluating `MakeAdder(2)` is a function that will add 2 to a single input. Thus, `(MakeAdder(2))(4)` will return 6.

“MakeAdder” could not have been written so easily without tags. If we had tried use `THIS`, how could the “n” and “m” parameters be distinguished?:

```
MakeAdderTry::
      Result: [Result: @THIS + @THIS]
```

This “MakeAdderTry” is really a “MakeDoubler” which ignores its input values. In fact, “MakeAdder” could still be written:

```
MakeAdder::
      Result: [@THIS Result: THIS.1 + THIS.2]
```

It is left as an exercise for the reader to figure out why this works. In any case, this definition is clearly more difficult to understand than the “MakeAdder” which uses tags.

12.10 Using a File as an Entry

Often it is useful to keep lists of stimuli in separate text files. However, to get these stimuli into the experiment, the list must be accessible from the scripting language. The contents of a file can be transformed into the content of an entry by using a *file reference*. A file reference is a type of reference (see “12.2.4 References”, p323), so it can be used with the `@` operator (and many others) to obtain the values from the file.

File references are created by using the `FileRef()` function. When a `FileRef()` expression is evaluated, a virtual entry is created to contain the expressions from the file; this virtual entry is called a *file entry*.

The referenced text file can have one of two forms: *literals-only* or *full-expressions*. **Literals-only** files can be read in more quickly and efficiently than files with non-literals, but they are also less expressive in that they can contain only literal strings. **Full-expression** files are distinguished from literals-only files by starting with the identifier `#NonLiteral`; they can contain expressions of any type.

The referenced text file (literals-only or full-expression) should *not* have the syntax of a script entry; it should just be a list of expressions for the content of the file entry. In a literals-only file, all blank-delimited strings are treated as literals; otherwise, expressions are read from the file just as from the content of an entry. Quotes and curly braces (`{ }`) can be used to delimit literals in either type of file.

Suppose we have a text file named “Fruit” (in the same directory as the current script) that looks like this:

```
apple  
banana  
coconut
```

The expression `FileRef("Fruit")` creates a file entry with three tokens — `apple`, `banana`, and `coconut` — in its content. Evaluating `@FileRef("Fruit")` returns `apple banana coconut`. `Access(FileRef("Fruit"))` returns `apple` the first time it is evaluated, `banana` the second time, and so on.

If the same file is referenced in more than one place in the script, the same file entry will be used. If the referenced file is changed, the file entry will be automatically updated.

File entries will never have attributes since only the content of a file entry can be specified in the file. However, a file reference can be incorporated into an inline entry (see “12.9 Inline Entries”, p335), and then attributes can be defined within the inline entry. Thus, `Access(FileRef("Fruit")[AccessType: Random])` will access the items of the file randomly.

Another way to access the file items randomly would be to create a new entry like this:

```
Fruit:: @FileRef("Fruit")  
        AccessType: Random
```

Then, `Access(Fruit)` will access the items of the file randomly. This is a little less efficient for the script interpreter, but a possibly little more clear.

12.11 Inherited Attributes

Note: Attribute inheritance is a feature that is built into PsyScript; this is not exactly the same concept as the attribute inheritance used in a factor format experiment (see “Part 2: Graphic Environment Reference, 5.8.1.1 Attribute Inheritance”, p150).

Attributes have thus far been defined as objects owned by entries. In an experiment definition, however, it is useful to be able to assign attributes to individual *tokens* — which are used as stimuli. The way in which a token can be assigned attributes is referred to as *inheritance*.

The rules of attribute inheritance give intuitive results in most cases. A complete definition, however, is complicated and needs to be mastered only by a scripter writing very complex scripts in one of the old scripting formats (i.e., `EventList` or `StimList`).

A token inherits attributes based on the location of its source expression(s). There are three basic rules of inheritance:

Rule 1.

A token that is the result of a particular express inherits all of the attributes of the expression's entry.

For example, in this entry definition:

```
TheEntry:: "Hello World" 2 + 3
          Color: Red
          Size: 14
```

Both tokens of “TheEntry” — “Hello World” and 5 — inherit the “Color” and “Size” attributes.

Rule 2.

If the expression is a *token reference*, the token inherits all of the inherited attributes of the referenced token.

The operators and functions that form token references are: @, ->, ., ?, Access(), GetCurrent(), GetPrevCurrent(), GetToks(), and Do(). Evaluation of a script-defined function is also a token reference.

In this example:

```
TheEntry:: "Hello World" 2 + 3
          Color: Red
          Size: 14
          DefaultSize:> 12

AnotherEntry:: @TheEntry TheEntry->Size
```

The first two tokens of “AnotherEntry” — “Hello World” and 5 — inherit the “Color” and “Size” attributes of “TheEntry”. The third token — 14 — inherits the “DefaultSize” sub-attribute of “Size” in “TheEntry”.

Rule 3.

Attributes which are inherited by Rule 2 take precedence over those inherited by Rule 1.

Expanding the example a bit more:

```
TheEntry:: "Hello World" 2 + 3
          Color: Red
          Size: 14
          DefaultSize:> 12

AnotherEntry:: @TheEntry TheEntry->Size
              Color: Blue
```

The first two tokens of “AnotherEntry” will inherit the “Color” attribute of “TheEntry”, in accordance with Rule 2. The third token will also inherit a “Color” attribute, but the one from “AnotherEntry”, in accordance with Rule 1.

Inheritance in this way establishes a kind of hierarchy of attributes. The pattern of token referencing is always direct; the last entry referenced — the one in which a literal or non-referencing expression was finally located — takes highest precedence because it is the most “specific” entry for the token. The top-level entry is the most general.

Attributes inherited by Rule 2 are taken only from the *referenced token*, i.e., the token which is the result of evaluating the expression. Attributes inherited by parameters of the expression are not used.

For example:

```
AddingNumber:: 3
               Minimum: 2

Addition:: @AddingNumber + 7
```

The first token of “Addition” does *not* inherit the “Minimum” attribute of “AddingNumber”. While the result of @AddingNumber does inherit this attribute, the inheritance does not pass through the + operator.

12.11.1 Inheritance and Token Reference Inline Entries

Calls to script-defined functions (see “12.9.6 Function Definitions”, p340) are considered token references, but inheritance in this case deserves some discussion. Consider this example:

```
Double:: `n
        Result: n * 2
           Even:> True
           Odd: False

DoubledValue:: Double(9)
```

The token of “DoubledValue” — 10 — will inherit the “Even” sub-attribute of “Result” in “Double”; it will *not* inherit the “Odd” attribute. This is because the result of a function call is taken from the “Result” attribute.

On the other hand, consider this example:

```
StandardText::
  Size: 9
  Font: Monaco

TextValue:: StandardText("Hello World")
```

Here, the “function” entry “StandardText” does not have a “Result” attribute. By the rules of token reference inline entries, the value of StandardText("Hello World") is taken from the content, which gives "Hello World". Because “StandardText” was incorporated into the inline entry, this token does inherit the “Size” and “Font” attributes.

The previous example suggests a way to use inline entries so that tokens are assigned attributes with a very simple syntax. For example, suppose an experiment had a list of word

stimuli, each of which was to be displayed in a different but pre-assigned color. The list could be implemented like this:

```
Stimuli:: man
          hat
          (cat Color: blue)
          cow
          (dog Color: red)
          (tree flower Color: green)
          truck
          Color: black
```

The default color for each stimulus is `black`, from the “Color” attribute of “Stimuli”. But the `cat` token has been assigned `blue`, `dog` has been assigned `red`, and `tree` and `flower` have been assigned `green`.

12.12 Crossing Lists

PsyScript provides a primitive facility for crossing lists together as factors. The crossing facilities of Factor format are built on top of this.

When a number of lists are *crossed*, a current item is selected for each list so that all of the unique *combinations* of items from each list are obtained. For example, there are six such combinations when crossing of a list with two items and a list with three items. Setting the current items in the lists which make up a combination constitutes *selecting* the combination.

The crossing function is `Cross()`, which takes as its arguments references to lists. It sets a current item for each list, and returns nothing.

A checklist is created by the `Cross()` function to keep track of which combinations have been selected (see below, “12.12.2 Checklist Storage”, p349). Each time the `Cross()` function is evaluated, the combination is chosen by this algorithm:

- Decide which items from the first list can make a new combination.
- Modify the checklist of the first list, so that those items which do not make new combinations are marked off.
- Access the first list.
- Decide which items from the second list can make new combinations, given the item selected from the first list.
- Modify the checklist of the second list...

and so on, until an item has been selected in all of the lists.

For example, consider the crossing of these two lists:

```
List1:: a b
List2:: 1 2

CrossEm:: Cross(List1 List2) GetCurrent(List1 List2)
```

The first evaluation of `@CrossEm` will return `a 1`. The next evaluation will return `b 2`, then `a 2`, then `b 1`.

The crossing sequence can be modified by assigning “Access” attributes to the lists, since the crossing algorithm calls `Access()`. However, some special features of crossing can be obtained through a “Crossing” sub-attribute of the “AccessType” attribute. The possible values for “Crossing” — and how they modify the basic crossing algorithm — are:

`Force` – This is the usual crossing mode, as described above.

`Static` – If the item that was selected for this list in the last `Cross()` can still be used, re-use that item.

`Independent` – Try to access the list independently of the `Cross()` algorithm, but still follow obey the constraint of generating unique combinations. Algorithmically, this means that the checklist is not modified if all of the unchecked items can be used in new combinations.

`Least` – Select from the set of list items that have so far been used the least in combinations.

For example, if we change “List2” above to

```
List2:: 1 2
      AccessType: Seq
      Crossing:> Static
```

then evaluations of `@CrossEm` will return `a 1`, then `b 1`, then `a 2`, then `b 2`.

12.12.1 Mapped Crossings

A variation of the `Cross()` function is the `MappedCross()` function. `MappedCross()` also takes references to lists, but the first list should be a special “mapping list” of *combination specifications*.

Each specification is a reference to a list of numbers, one number for each list to be crossed, designating the item from that list to be used the combination. Combinations may be repeated in the mapping list, and the order in which the combinations are selected is based on the access type of the mapping list. (Here, “mapping list” is used differently than it was used in “12.8.7 Sublisting”, p334.)

Mapped crossing are usually best implemented with inline entries. In this example of the mapping function


```
List1:: a b
List2:: 1 2
MapCrossEm:: MappedCross([[1 2] [2 1]] List1 List2)
              GetCurrent(List1 List2)
```

evaluations of `@MapCrossEm` will return a 2, then b 1.

12.12.2 Checklist Storage

For lists, the checklist is stored with the list itself, so that multiple references can be made to the list, and all of the references will use the same list.

For `Cross()` and `MappedCross()`, the checklist is stored with the *expression*. (This is somewhat like inline entries, where the checklist is stored with some special instance of the entry; see “12.9.4 Inline Entries and Lists”, p338.) Thus, two different expressions will keep separate checklists, even if they cross the same factors.

However, when `Cross()` or `MappedCross()` is evaluated, the checklist and current item of the crossed lists are modified. This is independent of the expression itself.

12.13 Optimizations

Here are a few tips to make your script more easily and quickly read by the interpreter:

Use the `#noinherit` and `#inherit` modifiers in your experiment entries to block out attributes not used by the events. See “12.4.3 `#inherit` and `#noinherit`”, p325.

When you have a complex expression that evaluates to only one literal, use single token referencing functions and operators (such as the item reference operator: `.`) instead of multiple-token referencing (such as `@` and `->`). For example, instead of writing `(SomeEntry->X/@Multiplier)*4`, use `(SomeEntry>>X.1/Multiplier.1)*4`.

Try to access lists directly, rather than through another entry that refers to the entire list through a token reference. This is especially faster when the list is a file entry. For example, instead of declaring `MyList::@(FileRef("filename"))` and using `Access(MyList)`, just use `Access(FileRef("filename"))`. See also “12.10 Using a File as an Entry”, p343.

12.14 Script Operators and Functions Summary

In the below summaries,

ref, *list* may be any reference
n, *x*, *y* may be any numerical value
attrib may be any name of an attribute in the relevant entry
string may be any string
bool may be any boolean value (see “12.6.1 Literals”, p326)
func may be any *ref* that is a script-defined function
token may be any value
NULL is the empty list of values, counting as 0 tokens

Functions which are marked with [PsyScopeTools] are not built-in functions; these are script-defined functions that are available to you through PsyScopeStdLib. Since they are script-defined functions, they do not automatically distribute over arguments, but the ** and *! operators can be used with them.

12.14.1 Operators and Functions

12.14.1.1 Reference Operations

^string - creates a reference to the entry named by *string*. When a reference is used with an operator in a function that takes a reference as a parameter, the ^ is understood and may be left out. Thus *@entry* is the same as *@^entry*. (In practice, ^ is never used; it could be useful if you needed to *force* a string to be a reference.)

@ref - returns all of the tokens in the content of the references entry. See also “12.2.4 References”, p323.

ref.n - returns the *n*th list item in the referenced entry.

ref->attrib - returns all of the tokens of the attribute named *attrib* in the referenced entry. If *attrib* is not an attribute of *ref*, NULL is returned.

THIS may be used as a reference to the *global* entry in which it is used in the script. If the global entry is incorporated in to an inline entry, the scoping of THIS will be changed to refer to the inline entry. (See “12.2.4.1 THIS and OWNER”, p324.)

OWNER may be used as a reference to the entry (not necessarily global) owning the entry in which the keyword is used. (See “12.2.4.1 THIS and OWNER”, p324.)

FileRef(string) creates and returns a reference to a virtual entry containing the tokens in the file named by *string*. See “12.10 Using a File as an Entry”, p343.

f^string is synonymous with *FileRef(string)*.

GetToks(ref) is synonymous with *@ref*.

12.14.1.2 Math Operations

`+`, `-`, `*`, `/`, and `%` are supported as the standard operators ($n \% m$ is n modulo m). `+=`, `-=`, etc. combine the math operators with the assignment operator (see below) like the standard C operations.

`x//y` returns the integer part of x/y .

`Div(x, y)` returns the integer part of x/y . See also “12.6.2.1 Exceptions to the Rules”, p327.

`Power(x, y)` returns x raised to the power y . See also “12.6.2.1 Exceptions to the Rules”, p327.

`Exp(x)` returns e (Euler’s number) raised to the power x .

`Log(x)` returns the natural logarithm of x .

`Sin(x)`, `Cos(x)`, `Tan(x)`, and `Arctan(x)` perform the standard trigonometric operations on x .

`Sum(x1, x2, ... xn)` returns the sum $x1 + x2 + ... xn$.

`Product(x1, x2, ... xn)` returns the product $x1x2 ... xn$.

`GCD(x1, x2, ... xn)` returns the greatest common divisor of the x ’s.

`LCM(x1, x2, ... xn)` returns the least common multiple of the x ’s.

`<`, `>`, `==`, `!=`, `<=`, `>=` compare two numerical values and return `True` or `False`.

`bool1 && bool2` returns `True` if both `bool1` and `bool2` are equivalent to `True`, or `False` otherwise.

`bool1 || bool2` returns `True` if either `bool1` or `bool2` is equivalent to `True`, or `False` otherwise.

`!bool` returns `True` if `bool` is equivalent to `False`, or `False` if `bool` is equivalent to `True`.

`Or(x1, x2, ... xn)` returns `True` if at least one of `x1`, `x2`, ... `xn` is equivalent to `True` (see “12.6.1 Literals”, p326), or `False` otherwise. (`Or(NULL)` returns `False`.)

`And(x1, x2, ... xn)` returns `True` if none of `x1`, `x2`, ... `xn` are equivalent to `False` (see “12.6.1 Literals”, p326). (`And(NULL)` returns `True`.)

`TruthVal(x1)` [PsyScriptTools] returns 1 if `x1` is equivalent to `True`, 0 otherwise.

`Max(x1, x2, ... xn)` [PsyScriptTools] returns the maximum value among `x1`, `x2`, ... `xn`.

12.14.1.3 String Operations

`Strlen(string)` returns the number of characters in *string*.

`$string` is synonymous with `Strlen(string)`.

`Strcat(string1, string2, ..)` returns a new string which is *string1* followed by *string2* etc. No extra spaces are added between the concatenated strings.

`string1$+string2` is synonymous with `Strcat(string1, string2)`.

`StripFrontChars(string, n)` returns *string* without the first *n* characters. If a character is passed instead of a number in *n*, characters will be stripped from *string* up to and including the first occurrence of the character in *string*. See also “12.6.2.1 Exceptions to the Rules”, p327.

`StripEndChars(string, n)` returns *string* without the last *n* characters. If a character is passed instead of a number in *n*, characters will be stripped from *string* after and including the last occurrence of the character in *string*. See also “12.6.2.1 Exceptions to the Rules”, p327.

`string$-n` and `string-n` are synonymous with `StripFrontChars(string, n)` and `StripEndChars(string, n)`, respectively.

`NthChar(string, n)` returns the *n*th character in *string*. See also “12.6.2.1 Exceptions to the Rules”, p327.

`<`, `>`, `==`, `!=`, `<=`, `>=` compare two strings (using dictionary ordering) and return `True` or `False`. The string comparisons are case-insensitive.

12.14.1.4 List Operations

`ListSize(list)` returns the number of items in *list*.

`ListWeight(list)` returns the total weight of the of items in *list*.

`Access(list)` chooses and returns a new current item based on the *list*'s attributes. See also “12.8.1 Accessing a List”, p330.

`AccessAll(list)` [PsyScopeTools] calls `Access()` on *list* `ListWeight(list)` times, thus exhausting the list once.

`Next(list)` chooses a new current item for *list*, but returns *list*'s entry name instead of the new current item.

`list?` is synonymous with `Next(list)`.

`GetCurrent(list)` returns the current item for *list*.

`?list` is synonymous with `GetCurrent(list)`.

`?ref?` is synonymous with `Access(list)`.

`GetPrevCurrent(list, n)` returns the n th previous current item for `list`, with $n = 1$ being the current item. The definition of `list` must include a “SaveCurrents” attribute. See also “12.8.6 SaveCurrents”, p334.

`SetCurrent(list, n)` accesses the list choosing the n th item, thus making it the current item.

`CurrentIndex(list)` returns the index number for the current item in `list`.

`PrevCurrentIndex(list, n)` returns the index number for the n th previous current item in `list`, with $n = 1$ being the current item. The definition of `list` must include a “SaveCurrents” attribute. See also “12.8.6 SaveCurrents”, p334.

`Column(list, n1, n2)` creates an accessible sublist of `list` by taking the $n2$ th of $n1$ columns. See also “12.8.7 Sublisting”, p334.

`Row(list, n1, n2)` creates an accessible sublist of `list` by taking the $n2$ th of $n1$ rows. See also “12.8.7 Sublisting”, p334.

`Sublist(list, n1, n2, n3)` creates an accessible sublist of `list` starting at the $n1$ th item and $n2$ items long. The $n3$ parameter is optional; if it is present, the $n2$ items are not chosen sequentially, but $n3$ items apart. See also “12.8.7 Sublisting”, p334.

`Map(list1, list2)` creates an accessible sublist of `list` by mapping `list2` onto `list1`. See “12.8.7 Sublisting”, p334.

`ref1|ref2` is synonymous with `Map(ref1, ref2)`

`Diag(list)` returns the same thing as `Sublist(list, 2, root, root + 1)`, where `root` is the square-root of the length of `list`. Then, if `list` has `root` rows and `root` columns, the sublist is the items along the diagonal.

`AllExcept(list, n1, n2)` [PsyScopeTools] returns a sublist of `list`, omitting $n2$ items, starting with the $n1$ th item. $n2$ defaults to 1.

`AccessSome(list, n1, n2, n3)` [PsyScopeTools] accesses all of the items from a sublist of `list`. The parameters are the same as for `Sublist()`.

`GetSome(list, n1, n2, n3)` [PsyScopeTools] returns all of the items from a sublist of `list`. The parameters are the same as for `Sublist()`.

`ListSum(list)` [PsyScopeTools] returns the weighted sum of all elements in the given list. This is like `Sum(list)`, but taking into account the weights on the items of `list`. `ListSum()` calls `AccessAll()`.

`ResetList(list)` resets `list` to its pre-accessed state. Multiple references may be passed to the function. The return value is `NULL`.

`ResetAll()` resets all lists in the script that have been accessed. The return value is `NULL`.

`Cross(list1, list2, ...)` sets the current item in each of the lists to produce a unique combination. See “12.12 Crossing Lists”, p347.

`MappedCross(mapping_list, list1, list2, ...)` sets the current item in each of the lists based on the specifications in `mapping_list`. See “12.12 Crossing Lists”, p347.

`Head(list)` [PsyScopeTools] returns the first item in `list`.

`Tail(list)` [PsyScopeTools] returns a sublist containing all items except the first item in `list`.

12.14.1.5 Other Operations

`n~value` returns `n` copies of `value`.

`tok_ref = string` sets the token referred to by `tok_ref` to the value `string`. (The `@` operator is the most common token reference operator).

`start .. end` returns all of the integers from `start` to `end` (inclusive). `start` may be less than or greater than `end`.

`func**(value1, value2, ...valuen)` evaluates `func(value1)`, `func(value2)`, ... `func(valuen)`. Built-in function names cannot be used for `func`, only script-defined functions.

`func*!(ref1, ref2, ...refn)` evaluates `func(@ref1)`, `func(@ref2)`, ... `func(@refn)`. Built-in function names cannot be used for `func`, only script-defined functions.

`Inherited(string or ref, attrib)` returns a reference to the attribute named `attrib` inherited by `string` or `ref`. If the attribute is not found, `NULL` is returned. See also “12.11 Inherited Attributes”, p344.

`EntryName(ref)` returns the full name of the entry referred to by `ref`.

`Evaluate(string)` evaluates `string` as an expression.

`If(bool token1 token2)` returns `token1` if `bool` is equivalent to `True`, or `token2` if it is equivalent to `False`. `token2` may be `NULL`. See also “12.6.1 Literals”, p326 and “12.6.2.1 Exceptions to the Rules”, p327.

`Null(token1, ...)` ignores the `tokens` and returns `NULL`.

`Time(n)` returns a string for the time at which the call is made. The parameter `n` specifies the style of the string:

- 0: “1:00 PM”
- 1: “13:00”
- 2: “13:00:00”

-1: <integer representing the date and time>

Date(*n*) returns a string for the date on which the call is made. The parameter *n* specifies the style of the string:

0: "01/01/90"
 1: "January 1, 1990"
 2: "01JAN90"
 -1: <integer representing the date and time>

Random(*n*) generates a random integer from 0 to *n*-1.

Run(*ref*, *attrib*) performs user input for the entry referenced by *ref* and using the dialog or function specified in the *attrib* attribute of the referenced entry. Run() returns the tokens of the referenced entry after the dialog/function has been executed. If *attrib* is not specified, a default dialog attribute ("Dialog", "Function", or "DCOD") will be used. DoDialog, DoFunction, and Do are synonyms for Run. See also "Chapter 17. Dialog and Function Extensions", p467.

Iterate(*start_ref*, *test_ref*, *loop_ref*) is a function for performing iterative operations, always returning NULL. Iterate begins by evaluating all of the tokens of *start_ref*. Then, while evaluating all of the tokens of *test_ref* give true boolean values, all of the tokens of *test_ref* are evaluated. The values of *test_ref* are checked once before *loop_ref* is ever evaluated. Do not use an inline entry for the test condition (see "12.9.1 Inline Entries vs. Regular Entries", p336).

AppendTok(*ref*, *token1*, ...) appends the *tokens* to the content line of the referenced entry. The number of tokens may be arbitrary.

DeleteTok(*ref*, *n*) deletes the *n*th token from the content of the referenced entry.

DeleteAllToks(*ref*) deletes the all tokens from the content of the referenced entry.

AddAttrib(*ref*, *attrib*) adds an attribute named *attrib* to the attribute block of the referenced entry.

Match(*func*, *list*) [PsyScopeTools] calls *func* on each item in *list*; Match() returns all items for which *func* returned a True value.

IsItemInList(*list*, *string*) [PsyScopeTools] calls Match() on *list* with a function that returns True when the list item matches *string*. Thus, the result will be *string* if it is found in *list*, or NULL otherwise.

PosIfItemInList(*list*, *string*) [PsyScopeTools] returns the position of the first item in *list* which matches *string*.

RemoveMatching(*list*, *string*) [PsyScopeTools] removes the first item in *list* which matches *string*.

RemoveDups(*list*) [PsyScopeTools] removes items from *list* so that no item value is duplicated.

`CopyContent(list1, list2)` [PsyScopeTools] copies all of the tokens from the content of `list2` to the content of `list1`. The number of expressions in the content of `list1` may be changed, if necessary.

`StartWatchCursor()` and `EndWatchCursor()` [PsyScopeTools] changes the user's cursor to a spinning watch or changes it back to the arrow. Each call to `StartWatchCursor()` should be balanced by a call to `EndWatchCursor()`. The calls can be nested arbitrarily.

12.14.2 Operator Precedence

From highest precedence to lowest:

```
{ }  
( ) and [ ] inline entry (and function) constructs  
^  
@, f@  
..  
>>, ->  
**, *!  
|  
!  
$  
*, /, //, %  
+, -, $+, $-, -$  
>, <, >=, <=, ==, !=, ||, &&  
=  
ref?  
?ref
```

The highest precedence operators will be evaluated first in a expression.

12.15 The Evaluator

The Evaluator in PsyScope allows you to type valid PsyScript expressions and evaluate them immediately. The evaluator does this by temporarily creating a temporary entry, making the entered expression(s) the token(s) of the entry, and then evaluating all of the tokens of the entry. The “TempEntry” entry is destroyed after each evaluation and recreated. An expression is re-parsed and compiled each time it is re-evaluated.

Expressions to be evaluated are typed into the evaluator window; when **Evaluate** is selected from the menu (or the **Evaluate** button is hit). If a range of text is selected, the range is used as the expression to be evaluated; otherwise, the expression to be evaluated is taken as all lines preceding the caret position and following the last commented line.

See also “Part 2: Graphic Environment Reference, 7.4 The Evaluator”, p263.



Chapter 13. Experiment Scripting Reference

13.1 Experiment Scripting Basics

13.1.1 Introduction

By “scripting”, we refer to the task of using PsyScript to describe an experiment for PsyScope to execute. All experiments are ultimately defined in PsyScript; even experiments built in the graphic environment are scripted — the graphic environment merely builds the script for you. When you become the scripter, you gain more control over the implementation of your experiment.

13.1.2 Script Interpretation

An experiment is executed in the following way:

- First, PsyScope reads the script and finds all experiment-wide information (e.g., the input devices to be monitored). All of the timing and scheduling structures are set up to run the experiment. An initial calculation is made for the *trial count*, i.e., number of trials to be executed.
- Then, for each trial:
 - PsyScope gets all of the information needed to run the trial; this is the *compiling* phase.
 - After the trial is compiled, it is sent to the Trial Manager for *execution*, i.e. running the trial in real time.
 - Once the trial ends, data for the trial is recorded into the data file.
 - The trial count is re-calculated. The trial count can change if block durations are used or if the script has been changed during an earlier compilation (see below), but usually the trial count is fixed.
- When the number of trials that have been executed is the same or greater than the trial count, then the experiment is over.

13.1.2.1 Self-Modifying Scripts

While compiling and executing an experiment, the script usually remains a static description of the experiment. However, the script can change in a few ways as trials are compiled and executed:

- In the compiling phase, *side-effect script operations* can alter the information in the script, affecting the way the rest of the trial or future trials are compiled.
- In the execution phase, polling of the `ScriptWhen[]` condition and execution of the `ScriptEval[]` action can modify the script by causing side-effect script operations to be evaluated.
- At the end of executing a trial, trial variable values can be written back to the script.

The *side-effect script operations* are the assignment operators (`=`, `+=`, `-=`, etc.), `AppendTok()`, `DeleteTok()`, `DeleteAllToks()`, and `AddAttrib()`.

Any of these script-modifying features may be used in complex scripts to execute elaborate calculations. Usually, however, a static description of an experiment is possible and desirable.

13.1.3 Script Formats

The exact way in which an experiment is described in PsyScript depends on which script format is used. There are currently three experiment formats: *StimList*, *EventList*, and *Factor*.

All formats define an experiment using the same PsyScript constructs: entries, attributes, and values. They use the same specific attribute names to specify features of stimuli or the experiment as a whole. However, the formats differ in how the high-level structure of an experiment is defined (i.e., groups, blocks, templates, and factors).

PsyScope's graphic environment produces scripts using Factor format, which is the preferred format. In Factor format, structural information — groups, blocks, templates, and factors — are defined explicitly using a hierarchy of entries; this hierarchy corresponds directly to the object hierarchy of the graphic environment (see “Part 2: Graphic Environment Reference, 5.2.1 Objects and the Experiment Hierarchy”, p107).

StimList and EventList are primitive formats that are no longer in common use; they rely more heavily on the scripter to perform all structural manipulations. For example, Factor format implements groups, blocks, etc. for you based on your scripted description, but you must implement any such concepts directly for StimList and EventList formats by using the functional aspects of PsyScript.

All formats overlap in that:

- The experiment description begins with an *experiment entry* that is listed in the script's "*Experiments*" entry (see below).
- There are standard experiment, trial, and event attributes; however, different formats require these attributes to be defined at different places (i.e., within different entries) in the script.

13.1.4 The 'Experiments' Entry

All scripts must contain an "Experiments" entry, which contains the names of all experiments defined in the script.

The "Experiments" entry has one attribute (which PsyScope will add for you if it is not present) called "Current"; "Current" specifies which of the various experiments in the script is currently active, i.e., which one will be executed when the script is run.

For example, here is an "Experiments" entry in a script which contains two experiments: "Acuity" and "2-D Acuity"; "Acuity" is the current experiment:

```
Experiments:: "Acuity" "2-D Acuity"  
Current: 1
```

In practice, you will usually only want one experiment per script, so that your scripts will be smaller and easier to manipulate. However, the multiple-experiment facility is useful for experiments that are small, extremely similar, or that use shared data resources or routines.

13.1.5 Experiment Entries

For each experiment name listed in the "Experiments" entry, there should be an entry in the script with that name; each of these entries is an *experiment entry*. (Note that each *experiment entry* is different from — and not to be confused with — the "*Experiments*" entry.)

The definition of an experiment always begins at an experiment entry; how the description proceeds from this entry depends on the experiment's format.

Different experiments within the same script are not required to use the same experiment definition format. The format for an experiment is specified in the experiment entry, in a "Format" attribute; the possible values for this attribute are `Factor`, `StimList`, and `EventList`.

13.1.6 Standard Attributes

13.1.6.1 Standard Experiment Attributes

There are a standard set of attributes which define properties of the experiment as a whole, such as the format of the script, the name of the data file, the background color to be used for screen stimuli, etc. These attributes are specified in the experiment entry, regardless of the experiment's format.

The complete list of experiment built-in attributes is given here. There are other experiment attributes which are specific to various output devices; see “14.2 Stimulus Types Reference”, p424 and the documentation for any PsychoScope Extensions which you are using.

Format: Factor/StimList/EventList
Default: Factor
Graphic: *always Factor*

This is the format of the experiment description; see “13.1.3 Script Formats”, p358.

Title: *string*
Default: *Name of experiment entry*
Graphic: *not available*

This is the title of the experiment that will appear in the log entry, and in the data file. It can be different from the name of the experiment entry in the script, but defaults to the experiment entry name.

InputDevices: *device-name-list*
Default: NULL

This attribute determines which input devices should be active during the experiment; every input device whose name is in the list will be active. (An input device name is the same as the name used for action conditions.)

Note: Conditions for actions are noticed only if the corresponding device has been activated.

Timer: *timer-name*
Default: Macintosh

This attribute determines which timer device should be used to run the experiment. *timer-name* is the name of a timing device (a ‘TIMR’ PsychoScope Extension).

Flags: NO_CLEAR_BY_DEFAULT *and/or* STORE_AT_END
and/or NO_SHOW_INSTRUCTIONS *and/or* NO_SAVE_SCREEN
and/or NO_DRAW_STIM_PORTS *and/or* ONSET_REF_START_BY_DEFAULT
and/or ODEV_DEBUG_MODE_ON *and/or* EVENT_DRAW_STIM_PORTS
and/or PRELOAD_ALL_STIMULI *and/or* BEEP_AT_REST_BREAKS
Default: NULL
Graphic: **Special**

This attribute contains a number of flags that modify how the experiment is executed:

NO_CLEAR_BY_DEFAULT: this flag changes the default “ClearType” event attribute value from FORCE_CLEAR to NO_CLEAR.

STORE_AT_END: Experiment data is usually stored after the completion of each trial. This flag causes all data to be stored instead at the end of the experiment.

DONT_HIDE_CURSOR: The cursor is usually turned off during an experiment run. This flag keeps the cursor turned on.

NO_SHOW_INSTRUCTIONS: If this flag is on, the instructions and debriefing message are skipped, even if they are defined.

NO_SAVE_SCREEN: If this flag is *not* on, the current screen is saved before rest periods (or error messages) are displayed. This flag is generally turned on due to memory constraints.

NO_DRAW_STIM_PORTS: This flag turns off the automatic drawing of port borders. this flag overrides EVENT_DRAW_STIM_PORTS.

ONSET_REF_START_BY_DEFAULT: This flag changes the way unspecified startrefs should be interpreted. With the flag off, the default startRef is 0 after the end of the previously specified event; with the flag on, the default startRef is 0 after the *start* of the previous event.

ODEV_DEBUG_MODE_ON: This flag makes the experiment window smaller and turns off some interrupts.

EVENT_DRAW_STIM_PORTS: When NO_DRAW_STIM_PORTS is not on, each event’s stimulus port is drawn at the beginning of the event, rather than at the start of the trial.

PRELOAD_ALL_STIMULI: When the experiment is precompiled, this flag causes stimuli to be loaded at precompile time rather than at run time. This can cause some memory problems; see “Part 2: Graphic Environment Reference, Preloading All Stimuli”, p250.

BEEP_AT_REST_BREAKS: When this flag is on, a beep is sounded when a rest period occurs.

UNUSUAL_SCREEN_PRESENT: Some very unusual screens (usually on the Pow-

erBook) do not generate retrace interrupts; this hangs PsyScope when it is trying to draw flicker-free. This flag is not for general use, but may be used in an attempt to make PsyScope run with a non-standard video device.

BBOX_IN_ONLY: An error in old button box cables caused problems with button box output (turning the lights on and off). This flag disables button box output.

Also, specific input and output devices can define flags; consult the device documentation.

DataFile: *file*
Default: NULL

This attribute contains the name of the file in which the data from the experiment will be stored. It is read at the beginning of the experiment.

DataHeader: *entry-list*
Default: *no entries*
Graphic: *not available*

This attribute should contain a list of references to entries in the script; the content of each referenced entry is then written on a separate line at the beginning of the data file.

Whether or not the information specified is actually written is controlled by the `DATA_FILE_HEADER` keyword in the “DataFields” experiment attribute (see below).

RunLabel: *entry-list*
Default: *no entries*
Graphic: *not available*

This is similar to the “DataHeaders” experiment attribute, except that the entry contents will appear in every record of the data file. The contents of all referenced entries are concatenated together and printed as a field in the data file.

Whether or not the information specified is actually written is controlled by the `RUN_LABEL` keyword in the “DataFields” experiment attribute (see below).

DataFields: `RUN_LABEL and/or STIMULUS and/or EVENT and/or EVENT_TAG and/or PUT_UP_BY and/or REMOVED_BY and/or DURING and/or RELATIVE_TO and/or RESPONSE_LABEL and/or ONSETS and/or TIMING_STATS and/or NO_HEADER`
Default: *no entries*
Graphic: **Data Info**

The following keywords, when included in the “DataFields” attribute of an experiment, designate which information about each response should be saved in the data file:

`RUN_LABEL:` includes the run label specified in the “RunLabel” experiment attribute as a field in the data file.

STIMULUS: reports the name of the stimulus of the event with which the data were stored.

EVENT: reports the event with which the response data were stored by the RT[] action.

EVENT_TAG: reports the event tag that was specified for the event the data were stored with.

PUT_UP_BY: reports the event that owned the RT[] action.

REMOVED_BY: reports the value of the “ActiveUntil” parameter of the RT[] action.

DURING: reports the event during which the RT[] action was executed. However, because several events may be running at one time, **DURING** reports only the event that was most recently begun. If all events have ended, then **DURING** has the value **SCHED_END** (this occurs when **FORCE_ONE** or **FORCE_ALL** response actions are still active).

RELATIVE_TO: reports the event relative to whose start the time of the response was measured.

RESPONSE_LABEL: reports the response label stored with the RT information.

NO_HEADER: inhibits reporting of the information specified in the “Data-Header” attribute of the experiment. This information is listed at the top of the data file, just after the title and time of the experiment.

MOUSE_POS: reports the position of the cursor (even if it was hidden) at the time of the response.

KEY_SEQUENCE: reports the contents of the **KEYSEQUENCE** input device buffer, filled by the most recent **KEYSEQUENCE** event.

ONSETS: reports the onset time of any event listed, measured relative to the start of the trial.

TIMING_STATS: reports summary statistics concerning the timing of individual events, averaged over all of the trials in the experiment.

FULL_TIMING_STATS: reports the onset and duration of every event in every trial, segmented by condition.

NumTrialsPerRest or **NumRestPeriods**: *number*
Default: *no rests*

One or the other of these two attributes should be used to set the number of rest periods to be given during the experiment run. Use “NumTrialsPerRest” to make rest periods always occur after a certain number of trials. Use “NumRestPeriods” to set a fixed number of rests to be given during the entire experiment; these will be divided evenly across the total number of trials in the experiment.

RestPeriod: *rest-msecs*
Default: 0
Graphic: **Rest Duration**

This attribute specifies a minimum duration (in msec) of the rest period. After this period, a “Press any key to continue...” message will be displayed. This value is only used if a rest period count is specified with “NumTrialsPerRest” or “NumRestPeriods”.

Instructions: *file*
Default: *no instructions*

This attribute specifies a file to be displayed before running trials. The file will be displayed in a window before any trials are run, and the words “Hit any key to continue” will appear at the bottom. Then when the subject hits a key, the window will close and the trials will be run.

If trials are precompiled, the instructions will be displayed first, and the trials will be compiled in the background. If the subject finishes reading the instructions and hits a key before the all the trials have been compiled, a timebar will appear indicating the continuing progress of compilation, and trials will be run when the compilation is finished.

The same formatting commands are available for instructions as for paragraph and document stimuli.

Instructions will not be displayed if `NO_SHOW_INSTRUCTIONS` is included in the “Flags” experiment attribute.

Debrief: *file*
Default: *no debriefing message*

This attribute specifies a file to be displayed after running all trials. The file will be displayed in a window before any trials are run, and the words “Hit any key to continue” will appear at the bottom.

The same formatting commands are available for debriefing as for paragraph and document stimuli.

A debriefing message will not be displayed if `NO_SHOW_INSTRUCTIONS` is included in the “Flags” experiment attribute.

DataRecordSeparator: *string*

Default: *newline*

This attribute specifies a string that will be used to separate the records in the data file.

Resources: *file-list*

Default: *no files*

This attribute designates a list of resource files that should be opened before the experiment is executed. This attribute is useful for opening resource files containing 'PICT' and 'snd' resources.

Precompile: *number*

Default: *0*

This attribute specifies the number of trials to always compile before running the experiment; the rest of the trials are compiled on a trial-by-trial basis, unless the precompile option is turned on. The keyword ALL may be used to precompile all trials.

ExpVariables: *variable-list*

Default: *no variables*

This attribute specifies a list of variables to be used in running the experiment. Each variable has an entry which defines the type and initial value of the variable. Built-in variables should not be included in this list.

DataVariables: *variable-list*

Default: *no variables*

This attribute specifies a list of variables that should be recorded with each line in the data file.

RunMode: *Run/Practice/Check*

Default: *not applicable*

The value of this attribute is automatically set when the experiment is run or checked. This can be used in PsyScript expressions to vary how aspects of the experiment are run in the different modes.

The value of this attribute is ignored by the experiment compiler after it is automatically set.

Reset: *references*
Default: `ResetAll()`
Graphic: *not available*

This attribute functions like an execution entry (see “16.6.2 Execution Entries”, p460) for when the experiment is over. When trials are executed with the Trial Monitor (see “Part 2: Graphic Environment Reference, 6.3 The Trial Monitor”, p238) this attribute is used only when the **Reset** button is hit (or when the experiment is automatically reset).

13.1.6.2 Standard Trial Attributes

There are a standard set of attributes which define properties of a trial, such as the condition name or actions which are active for the whole trial. Where these attributes are defined depends on the experiment’s format; see “13.2.1.1 Trial Attributes in StimList Format”, p370 or “13.3.5 Scripting Templates”, p381. The complete list of standard trial attributes is given here.

ConditionName: *format*
Default: *depends on format*
Graphic: **Condition Name**

This attribute specifies the format of the condition name to be recorded in the data file. In a StimList or EventList experiment, *format* is simply used as the condition name; “” is the default. In a Factor experiment, *format* can have the form:

prefix<separator>suffix

in which case the resulting condition name will include the names of the current levels and items of factors and lists relevant to the trial. The condition name will have the form:

prefixLevel1separatorLevel2separator...LevelNsuffix

where *Level1*, *Level2*, ... *LevelN* are the names of the current levels and items.

If no angle brackets appear in *format*, then *format* is simply used as the condition name. The default value is `< >`.

ITI: *msecs*
Default: `0`
Graphic: **Minimum ITI**

This attribute specifies the minimum amount of time that should be allotted in advance to load this trial, including compile time in non-precompiled experiments. The default is 0. See also “Part 2: Graphic Environment Reference, 6.5.2 Loading Stimuli”, p247.

TrialActions: *condition_action_pairs*
 Default: *no actions*
 Graphic: **Actions**

This attribute specifies condition-action pairs for actions that may remain active the duration of the trial; see also “13.4.1 Action Lists”, p409.

13.1.6.3 Standard Event Attributes

There are a standard set of attributes which define properties of an event, such as start time, duration, or actions. Where these attributes are defined depends on the experiment’s format; see “13.2.1 StimList Format”, p368 or “Part 4: Scripting Reference, 13.3.6 Scripting Events”, p383. The complete list of standard event attributes is given here, although most event types recognize a number of additional type-specific event attributes (see “14.2 Stimulus Types Reference”, p424).

EventName: *name*
 Default: *depends on format*
 Graphic: *not available*

This attribute sets the name of the event for the purposes of data recording, start references, and event specification in action parameters. The default depends on the format; see “13.2.4 StimList/EventList Event Names”, p373 or “13.3.6 Scripting Events”, p383. This attribute is usually not used.

EventTag: *tag*
 Default: *""*
 Graphic: **Tag**

This attribute assigns a tag to the stimulus for use with analysis; it is ignored by the Trial Manager, aside from being recorded in the data file.

EventType: *type*
 Default: *Text*
 Graphic: *not set as an attribute*

This attribute sets the type of the event, such as `Text` or `SoundLabel`. The list of all built-in types is in “14.2 Stimulus Types Reference”, p424. More types may be made available by using PsyScope Extensions (see “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216).

Duration: *duration*
 Default: *depends on event type*

This attribute sets the duration of the event; see “13.4.3 Duration”, p411 for details.

StartRef: *start_reference*
Default: *0 milliseconds after the end of the "previous" event*
Graphic: *not set as an attribute*

This attribute specifies the start reference of the event (i.e., when the event starts relative to the start of the trial or another event in the trial). See “13.4.2 Start Reference”, p411 for details.

ClearType: NO_CLEAR | FORCE_CLEAR *and/or* MASK
Default: ""
Graphic: **Tag**

This attribute sets whether and how the event is to be cleared; either NO_CLEAR or FORCE_CLEAR can be specified to override the default clear type set in the “Flags” experiment attribute (see “13.1.6.1 Standard Experiment Attributes”, p360 for information on the NO_CLEAR_BY_DEFAULT value).

If MASK is specified, the stimulus is “cleared” by drawing the mask rather than by erasing the stimulus. This feature applies primarily to Text stimuli.

EventActions: *condition_action_pairs*
Default: *no actions*
Graphic: **Actions**

This attribute specifies condition-action pairs for actions that are to be posted by this event; see also “13.4 Complex Attribute Formats”, p409.

13.2 StimList and EventList Formats

The StimList and EventList formats are the most flexible and terse formats for writing an experiment. The difference between StimList and EventList formats is small; this section will first document StimList format and then indicate how EventList format differs from StimList.

13.2.1 StimList Format

In StimList format, the way in which a trial is generated is conceptually simple: the values in the content of the experiment entry are the stimuli for the trial; each stimulus corresponds to one event, and the attributes for each event are read from the inherited attributes of its stimulus. Writing a non-trivial experiment from this simple rule requires a deep understanding of PsyScript.

Stimlist format can be very terse. Here is the script for an experiment that presents “Hello World” and waits for the user to press a key:

```
#PsyScope 1.0  
Experiments:: "Hello World Experiment"
```

```

Hello World Experiment:: "Hello World"
  Format: StimList
  InputDevices: Mouse
  Timer: Macintosh
  Duration: Mouse[Click]

```

Every trial will have one event with the stimulus "Hello World". The duration Mouse[Click] is picked up through attribute inheritance (see “13.2.3 Attribute Inheritance in StimList/EventList Format”, p372). The event is a Text event because the “EventType” attribute defaults to Text. Only one trial is executed because there is not “TrialCount” attribute.

The first step in creating non-trivial StimList script is to use lists and the Access() function (see “12.8.1 Accessing a List”, p330). In this experiment, three different messages are shown in three trials:

```

#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: Access(Hellos)
  Format: StimList
  InputDevices: Mouse
  Timer: Macintosh
  TrialCount: 3
  Duration: Mouse[Click]

Hellos:: "Hello World" "Hi World" "Howdy World"
  AccessType: Random

```

Here, the “TrialCount” attribute is set to 3, so three trials will be run.

“Templates” can also be implemented using Access() to create trials with completely different structures. In this example, the first trial shows a greeting in blue and is terminated by clicking the mouse; the second trial shows a parting message in red and is terminated by a key press:

```

#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: @Access(["Hello Trial" "Goodbye Trial"])
  Format: StimList
  InputDevices: Mouse Key
  Timer: Macintosh
  TrialCount: 2
  Duration: Mouse[Click]

Hello Trial:: Access(Hellos)
  Color: Blue

Goodbye Trial:: Access(Goodbyes)
  Duration: Key[Any]
  Color: Red

Hellos:: "Hello World" "Hi World" "Howdy World"
  AccessType: Random

```

```
Goodbyes:: "Goodbye World" "Bye World" "Adios World"  
  AccessType: Random
```

13.2.1.1 Trial Attributes in StimList Format

In StimList format, trial attributes are read by searching the inherited attributes of all stimuli in the trial (see “13.2.3 Attribute Inheritance in StimList/EventList Format”, p372). In the case of trial actions, the values are combined from all “TrialActions” attributes which are found.

13.2.1.2 Block Mode in StimList Format

StimList format does provide one level of built-in structural description: blocks. StimLists scripts run in either *Direct mode* (as described above) or *Block mode*. Direct mode is the default; Block Mode is specified by setting the “BlockMode” attribute of an experiment to True.

In Block mode, a list of blocks is given in the experiment entry’s content instead of a list of stimuli; each block is used for a the number of trials indicated in the block.

The list of blocks in the experiment entry are references to *block entries*; block entries are analogous to experiment entries, which are referenced in the content of the “Experiments” entry. Trials within a block are built from a block entry in the same way that trials are built from the experiment entry in Direct mode.

For example, here is a script that runs three greeting trials, and then three parting trials:

```
#PsyScope 1.0  
  
Experiments:: "Hello World Experiment"  
  
Hello World Experiment:: "Hello Trials" "Goodbye Trials"  
  Format: StimList  
  InputDevices: Mouse Key  
  Timer: Macintosh  
  BlockMode: True  
  Duration: Mouse[Click]  
  
Hello Trials:: Access(Hellos)  
  TrialCount: 3  
  
Goodbye Trials:: Access(Goodbyes)  
  TrialCount: 3  
  
Hellos:: "Hello World" "Hi World" "Howdy World"  
  AccessType: Random  
  
Goodbyes:: "Goodbye World" "Bye World" "Adios World"  
  AccessType: Random
```

Sometimes it is useful to scale the number of trials within all blocks by changing a single value in the experiment entry. Do do this, use the “BlockCount” attribute in the blocks, and set the “BlockMult” attribute in the experiment entry.

In this example, only two trials are executed within each block, but this can be changed by varying only the “BlockMult” experiment attribute:

```
#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: "Hello Trials" "Goodbye Trials"
  Format: StimList
  InputDevices: Mouse Key
  Timer: Macintosh
  BlockMode: True
  BlockMult: 2
  Duration: Mouse[Click]

Hello Trials:: Access(Hellos)
  BlockCount: 1

Goodbye Trials:: Access(Goodbyes)
  BlockCount: 1

Hellos:: "Hello World" "Hi World" "Howdy World"
  AccessType: Random

Goodbyes:: "Goodbye World" "Bye World" "Adios World"
  AccessType: Random
```

Note that the “BlockCount” block attribute allows trial count scaling, while the “Trial-Count” attribute does not.

13.2.2 EventList Format

The format name “StimList” is derived from the fact that trials are created by reading the experiment entry (or block entry) content as a list of stimuli; this implies the rule: “one event, one stimulus”.

In “EventList” format, the experiment entry (or block entry) content is read as a list of references to *event entries*; each event entry defines one event in the trial, but each event can have multiple stimuli (if the event’s type allows). In particular, EventList format must be used to create Pasteboard events, since multiple stimuli are what makes a Pasteboard event useful.

In this example, a Pasteboard event is used to show “Hello World” and “Goodbye World” simultaneously:

```
#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: HelloGoodbye
  Format: EventList
  InputDevices: Mouse
  Timer: Macintosh
  Duration: Mouse[Click]

HelloGoodbye:: ("Hello World" Position: 25%)
```

```
        ("Goodbye World" Position: 75%)
EventType: Pasteboard
StimType: Text
```

13.2.3 Attribute Inheritance in StimList/EventList Format

The mechanism by which attributes are collected for trials, events, and stimuli in the StimList and EventList formats is *inheritance*, as described in “12.11 Inherited Attributes”, p344.

Inheritance completely defined is a difficult topic, but in generally it works in the way one would expect. Basically, attributes are inherited top-down: an attribute can be specified starting at the experiment level, or at the block level, and so on down to the entry in which the stimulus actually is specified.

For example, consider the most recent example:

```
#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: HelloGoodbye
    Format: EventList
    InputDevices: Mouse
    Timer: Macintosh
    Duration: Mouse[Click]

HelloGoodbye:: ("Hello World" Position: 25%)
                ("Goodbye World" Position: 75%)
    EventType: Pasteboard
    StimType: Text
```

Here, the Pasteboard event inherits the “Duration” attribute value Mouse[Click] from the experiment entry. Both the “Hello World” and “Goodbye World” sub-stimuli inherit the “StimType” attribute value Text from “HelloGoodbye”.

If an attribute is set at multiple levels, the value that is used is the one defined “closest” to the stimulus. This means that attribute values specified in the stimulus’s entry override values specified in the experiment entry.

Consider this earlier example:

```
#PsyScope 1.0

Experiments:: "Hello World Experiment"

Hello World Experiment:: @Access(["Hello Trial" "Goodbye Trial"])
    Format: StimList
    InputDevices: Mouse Key
    Timer: Macintosh
    TrialCount: 2
    Duration: Mouse[Click]

Hello Trial:: Access(Hellos)
    Color: Blue
```



```
Goodbye Trial:: Access(Goodbyes)
                Duration: Key[Any]
                Color: Red

Hellos:: "Hello World" "Hi World" "Howdy World"
         AccessType: Random

Goodbyes:: "Goodbye World" "Bye World" "Adios World"
          AccessType: Random
```

Here, the greeting events inherit the “Duration” attribute value `Mouse[Click]` from the experiment entry, but the parting events use the “Duration” value `Key[Any]` from “Goodbye Trial”.

The most common misunderstanding about inheritance relates to function calls. The parameters of functions could possibly have inherited attributes, but these attributes will *not* be passed on to the result of the function.

For example, in this fragment:

```
ConcatenatedWord:: Strcat(@ConPart @CatenatePart)

ConPart:: "con"
          Color: Blue

CatenatePart:: "catenate"
              Color: Red
```

the resulting “concatenate” value of “ConcatenatedWord” inherits the color black (by default); the “con” inherits a blue color and the “catenate” part inherits a red color, but this is not passed on to the result of the `Strcat()` function.

This function-calling rule appears in some cases to be violated by script-defined functions; this is because script-defined “functions” are actually shorthand for other referencing constructions and can therefore pass on certain inheritances. See “12.11.1 Inheritance and Token Reference Inline Entries”, p346 for details.

13.2.4 StimList/EventList Event Names

In StimList format, the default event name is taken as the name of the entry where the most specific (i.e., most overriding) attributes can be defined.

In EventList format, the default event name is the name of the event entry.

13.2.5 StimList/EventList Optimization

By default, the StimList and EventList experiment compilers attempt to optimize by skipping script-reads which appear unnecessary. In particular, when the compiler reads a set of attributes for an event or trial, it remembers where the attributes came from; when future

trials are compiled, it checks to see if any of these attributes could have changed before it re-reads them.

If assignment statements are used in your script, optimization is not likely to help. Thus, to keep your script optimizable, you should avoid using the assignment operators (which is a good practice in general).

If the “SaveCurrents” attribute of a list is too high (see “12.8.6 SaveCurrents”, p334), the optimizer may be unable to properly track whether attributes have changed; if this is the case, the script interpreter will warn you when the list is initialized.

If your script turns out to be non-optimizable for either of the above reasons, it is best to turn optimization off with the “Optimization” experiment attribute (see below), since attempted optimization will take up extra memory and processing time.

13.2.6 Summary of Attributes for EventList and StimList Formats

Special experiment entry attributes:

`BlockMode`: Determines the mode in which the experiment is defined. Defaults to “FALSE.”

`TrialCount` or `Cycles`: Number of trials in experiment. The default is the value in Trials console.

`PracticeTrialCount` or `Practice`: Number of practice trials. The default is the value in Trials console.

`BlockMult` and `PracticeBlockMult` or `ScaleBlocks` and `PracticeScaleBlocks`: For use with Block Mode, the number sets to run of each block that has a “BlockCount” attribute. The default is 1, or “TrialCount” if defined.

`OptimizeEvents`, `OptimizeTrials`: Specifies whether the builder should attempt to optimize the building process at the event and trial levels. The default for both is “TRUE”.

`Optimize`: If neither “OptimizeEvents” or “OptimizeTrials” are specified, this attribute may be used to turn optimization on or off at both levels. The default is “TRUE”.

Block entry attributes (for use with Block Mode):

`BlockCount`, `PracticeBlockMult`: The number of trials in a set for this block; multiplied by the value of “BlockMult”, the number of trials of this block to run. The default is 1.

`TrialCount`, `PracticeTrialCount`: Number of trials in this block. (Default uses “BlockCount”).

13.2.7 StimList/EventList Compilation Details

This section addresses in more detail how a script is compiled into an experiment. It is optional reading, and may be skipped with little sacrifice of sagacity. Also, it is not quite right if optimization is on, although the results should be the same. This information is provided because scripters may find it useful to know exactly what the compiler is doing for performing side-effects in the right order or playing other tricks.

- 1) All the standard experiment attributes are read in.
- 2) The “BlockMode” attribute of the experiment is checked; if it is `True`, the list of block references is read and the number of trials in each block calculated.
- 3) For each trials:
 - 3.1) In block mode, the list of block references is used to determine which block should be used.
 - 3.2) The expressions in the experiment/block entry are evaluated from first to last to obtain the stimulus values. As each value is resolved, a list of references to the value’s inherited attributes is maintained.
 - 3.3) The events are taken one at a time to obtain the event attributes:
 - 3.3.1) The event name is read.
 - 3.3.2) The event tag is read.
 - 3.3.3) The start reference is read.
 - 3.3.4) The clear type is read.
 - 3.3.5) The event’s type is read, and all of the attributes associated with the event type are read.
 - 3.3.6) The duration is read.
 - 3.3.7) The event actions are read.
 - 3.5) The condition name and minimum ITI are read by searching the inherited attributes of each event, starting with the inherited attributes of the first event; once a value is found, the search stops.
- 3.4) Trial actions are read by searching the inherited attributes of each event, starting with the inherited attributes of the first event; the search combines the trial actions found from all events.

13.3 Factor Format

Factor format is the highly-structured scripting format used by the graphic environment. In Factor format, you define the structure of your experiment using the concepts of groups, blocks, templates, factors, and events. (These concepts are defined fully in “Part 2: Graphic Environment Reference”.) An understanding and careful use of these structures can greatly facilitate your design process.

13.3.1 Scripting the Experiment Hierarchy

“Part 2: Graphic Environment Reference, 5.2.1 Objects and the Experiment Hierarchy”, p107 describes the basic structure of a Factor format experiment. The *objects* of the graphic environment translate directly into *entries* in PsyScript, and the *attributes* of graphic objects are implemented as PsyScript *attributes* of these entries.

Linking a child object to a parent object translates into using a reference to the child entry in an attribute of the parent entry; for example, group entries are linked to the experiment entry by referencing the groups in the experiment’s “Groups” attribute.

The example below shows a Factor format script that implements all of the objects in Factor format. Later sections of the manual will refer back to this example to illustrate aspects of the experiment hierarchy:

```
#PsyScope 1.0
# Script template, Version 1.0

Experiments:: "Contrived Hello World"
  Current: 1

#> ExperimentDefinitions

Contrived Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  Flags: NO_SAVE_SCREEN
  DataFile: "Contrived Hello World Data"
  ScaleBlocks: 1
  Groups: Group1 Group2
  Current:> 1

#> GroupDefinitions

Group1::
  Blocks: Superblock1

Group2::
  Blocks: Superblock1

#> BlockDefinitions

Superblock1::
  Blocks: Block1 "Pause Block" Block2
  Factors: "Word Features"
```

```
ScaleBlocks: 5

Block1::
  Templates: "Hello-Goodbye Template"
  Cycles: 1

Block2::
  Templates: "Hello-Goodbye Template"
  Cycles: 1

Pause Block::
  Events: "Pause Event"
  FixedCycles: 1

#> FactorDefinitions

Word Features::
  Levels: Level1 Level2 Level3

#> LevelDefinitions

Level1::
  Color1: Red
  Color2: Cyan
  Face: ""

Level2::
  Color1: Green
  Color2: Magenta
  Face: "Bold "

Level3::
  Color1: Blue
  Color2: Yellow
  Face: "Italic "

#> TemplateDefinitions

Hello-Goodbye Template::
  Events: "Hello Event" "Goodbye Event"

#> EventDefinitions

Pause Event::
  EventType: Text
  Duration: Key[ RETURN ]
  Stimulus: "Take a break. Press Return to Continue..."

Hello Event::
  EventType: Text
  Duration: 500
  Stimulus: "Hello World"
  Color: FactorAttrib("Word Features", "Color1")
  Face: FactorAttrib("Word Features", "Face")

Goodbye Event::
  EventType: PasteBoard
  Duration: 500
  Stimuli: "Goodbye Stimulus" "Smiley Stimulus"

#> StimulusDefinitions

Goodbye Stimulus::
  StimType: Text
```

```
Stimulus: "Goodbye World"
Face: FactorAttrib("Word Features", "Face")
Color: FactorAttrib("Word Features", "Color2")

Smiley Stimulus::
  StimType: Text
  Stimulus: ":-)"
```

In Factor format, the content of an entry is always ignored (unless a PsyScript expression specifically refers to some entry's content); attributes are used to define all of the details of an experiment.

13.3.2 Scripting the Factor Format Experiment Entry

An Factor format experiment definition starts with an *experiment entry* (see “13.1.5 Experiment Entries”, p359).

The following attributes are recognized in the experiment entry:

- The standard experiment attributes, as listed in “13.1.6.1 Standard Experiment Attributes”, p360.
- The “Groups” attribute if the experiment has groups. (See also “13.3.3 Scripting Groups”, p379.)
- The “Blocks” attribute if the experiment has blocks, but no groups. (See also “13.3.4 Scripting Blocks”, p380.)
- The “Templates” attribute if the experiment has templates, but no blocks or groups. (See also “13.3.5 Scripting Templates”, p381.)
- The “Events” attribute if the experiment has no templates, blocks, or groups. (See also “13.3.6 Scripting Events”, p383.)
- The “Factors” attribute. (See also “13.3.7 Scripting Factors”, p387.)
- Trial counting attributes if the experiment has no groups. (See “13.3.10 Scripting Factor Format Trial Counts”, p401.)
- The “Optimization” attribute. (See “13.3.11.3 Factor Format Optimization”, p404.)
- Attributes referenced by `RunModeAttrib()`. (See “13.3.6.3 Linking Event Attributes to the Run Mode”, p385.)
- Any inheritable attributes recognized by a template or event entry. (See “13.3.5 Scripting Templates”, p381 and “13.3.6 Scripting Events”, p383.)

An experiment should have one, and only one, of the “Groups”, “Blocks”, “Templates”, or “Events” attributes. (This corresponds to the graphic environment's constraint that only

one type of object can be connected to the experiment object; see “Part 2: Graphic Environment Reference, 5.2.1.1 Linking Objects”, p108.)

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the “Groups” attribute is used in the “Contrived Hello World” experiment entry.

13.3.2.1 Factoring and Linking Experiment Attributes

All of the standard experiment attributes are *structural* (see “13.3.11.1 Structural vs. Non-structural Attributes”, p402), so that experiment attributes are not usually factored or otherwise varied. However, such operations can be performed on experiment attributes which are inherited by a template or event, or which are referred to through a `RunModeAttrib()` function call (see “13.3.6.3 Linking Event Attributes to the Run Mode”, p385). See “13.3.11.2 Attribute Inheritance in Factor Format”, p403 for more information on factoring and linking these attributes.

13.3.3 Scripting Groups

To define a group, you must use a “Groups” attribute in the experiment entry; the value of this attribute should be a list of references to entries representing the groups, called *group entries*.

Only one group will be used in a single experiment run; this is the *current group*. The “Current” sub-attribute of the “Groups” attribute specifies which group will be used when the experiment is executed; the value of the “Current” sub-attribute is a number which is the index of the current group in the list. (See also “Part 2: Graphic Environment Reference, 6.2.4 Automatic Grouping”, p233.) If the value of the “Groups” attribute or the “Current” sub-attribute changes during experiment execution, the original group will still be used.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the “Groups” attribute is used in “Contrived Hello World”, and “Group1” is the current group.

Groups can be nested within a group (a.k.a. *supergroup*), but this structure is not supported by the graphic environment.

The following attributes are recognized in group entries:

- The “Blocks” attribute if the group has blocks. (See also “13.3.4 Scripting Blocks”, p380.)
- The “Templates” attribute if the group has templates, but no blocks. (See also “13.3.5 Scripting Templates”, p381.)
- The “Events” attribute if the group has no templates or blocks. (See also “13.3.6 Scripting Events”, p383.)
- The “Factors” attribute. (See also “13.3.7 Scripting Factors”, p387.)
- Trial counting attributes. (See “13.3.10 Scripting Factor Format Trial Counts”, p401.)

- Attributes referenced by `GroupAttrib()`. (See “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.)
- Any inheritable attribute recognized by a template or event entry. (See “13.3.5 Scripting Templates”, p381 and “13.3.6 Scripting Events”, p383.)

A group should have one, and only one, of the “Blocks”, “Templates”, or “Events” attributes.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, both “Group1” and “Group2” use the “Blocks” attribute to include a superblock.

13.3.3.1 Factoring and Linking Group Attributes

All of the standard attributes of a group are *structural* (see “13.3.11.1 Structural vs. Non-structural Attributes”, p402), so that group attributes are not usually factored or otherwise varied. However, such operations can be performed on group attributes which are inherited by a template or event, or which are referred to through a `GroupAttrib()` function call (see “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385). See “13.3.11.2 Attribute Inheritance in Factor Format”, p403 for more information on factoring and linking these attributes.

13.3.4 Scripting Blocks

To define blocks, you must use a “Blocks” attribute in the experiment, group, or superblock entry (to which the block should be linked); the attribute’s value should be a list of references to entries representing the blocks, called *block entries*.

All of the blocks of an experiment, group, or superblock are executed. The “Blocks” attribute is accessed as a PsyScript list (see “12.8 Lists”, p329), so that the order in which the blocks are executed depends on the “AccessType” and other sub-attributes of the “Blocks” attribute.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the superblock “Superblock1” owns “Block1”, “Pause Block”, and “Block2”. These blocks are executed sequentially. “Superblock1” is itself linked to “Group1” and “Group2”.

In a *pass* through the blocks, the “Blocks” list is accessed once for each item in the list (regardless of list weights) and the block returned by the access is executed. By default, one pass is performed. A “Cycles” attribute can be defined in the experiment/group entry (the one that has the “Blocks” attribute) to specify the number of passes to performed.

Note: For the purpose of determining how many trials should be run for the whole experiment, the Factor format compiler assumes that every block will be run once in each pass. (This may not be the case if the RRandom access type is used.)

When a block is executed, all of the block's trials will be executed together. Each block specifies for itself how many trials are to be run within the block, using the "Cycles" or "FixedCycles" attribute. See "13.3.10 Scripting Factor Format Trial Counts", p401 for more information on trial counting.

The following attributes are recognized in block entries:

- The "Blocks" attribute if the block is a superblock (owning other blocks).
- The "Templates" attribute if the block has templates and no blocks. (See also "13.3.5 Scripting Templates", p381.)
- The "Events" attribute if the block has no templates or blocks. (See also "13.3.6 Scripting Events", p383.)
- The "Factors" attribute. (See also "13.3.7 Scripting Factors", p387.)
- Trial counting attributes. (See "13.3.10 Scripting Factor Format Trial Counts", p401.)
- Attributes referenced by `BlockAttrib()`. (See "13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes", p385.)
- Any inheritable attribute recognized by a template or event entry. (See "13.3.5 Scripting Templates", p381 and "13.3.6 Scripting Events", p383.)

A block should have one, and only one, of the "Blocks", "Templates", or "Events" attributes.

In the example of "13.3.1 Scripting the Experiment Hierarchy", p376, both "Block1" and "Block2" use the "Templates" attribute to include a template, but "Pause Block" is linked directly to an event with the "Events" attribute. "Superblock1" is a superblock because it includes other blocks with the "Blocks" attribute; it also is linked to the factor "Word Features".

13.3.4.1 Factoring and Linking Block Attributes

All of the standard attributes of a block are structural (see "13.3.11.1 Structural vs. Non-structural Attributes", p402), so that block attributes are not usually factored or otherwise varied. However, such operations can be preformed on block attributes which are inherited by a template or event, or which are referred to through a `BlockAttrib()` function call (see "13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes", p385). See "13.3.11.2 Attribute Inheritance in Factor Format", p403 for more information on factoring and linking these attributes.

13.3.5 Scripting Templates

To define a template, you must use a "Templates" attribute in the experiment, group, or block entry (to which the template should be linked); the attribute's value should be a list

of references to entries representing the templates, called *template entries*. See also “13.3.9 Scripting Factor Tables”, p395.

For each trial, one of the templates listed in the “Templates” attribute will be used. The “Templates” attribute is accessed as a PsyScript list (see “12.8 Lists”, p329) and the selected template is executed. The order in which the templates are executed thus depends on the “AccessType” and other sub-attributes of the “Templates” attribute.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the template “Hello-Goodbye Template” is linked to both “Block1” and “Block2”.

The following attributes are recognized in template entries:

- The standard trial attributes (inheritable), as listed in “13.1.6.2 Standard Trial Attributes”, p366.
- The “Events” attribute. (See also “13.3.6 Scripting Events”, p383.)
- The “Factors” attribute. (See also “13.3.7 Scripting Factors”, p387.)
- Attributes referenced by `TrialAttrib()`. (See “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.)
- Any inheritable attribute recognized by an event entry. (See “13.3.6 Scripting Events”, p383.)

A template entry should always have an “Events” attribute with a list of references to event entries (see “13.3.6 Scripting Events”, p383).

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the template “Hello-Goodbye Template” is linked to two events: “Hello Event” and “Goodbye Event”.

13.3.5.1 Factoring and Linking Template Attributes

Most of the standard template attributes are *non-structural* (see “13.3.11.1 Structural vs. Non-structural Attributes”, p402); these attributes can be:

- Factored using the `FactorAttrib()` function call; this corresponds to using **Vary by List** in the graphic environment. `FactorAttrib()` is described in “13.3.6.1 Factoring Event Attributes”, p384.
- Linked to a block attribute with `BlockAttrib()`; this corresponds to using **Vary by Block** in the graphic environment. `BlockAttrib()` is described in “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.
- Linked to a group attribute with `GroupAttrib()`; this corresponds to using **Vary by Group** in the graphic environment. `GroupAttrib()` is described in “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.

- Varied by the run mode using `RunModeAttrib()`; this corresponds to using **Vary by Run Mode** in the graphic environment. `FactorAttrib()` is described in “13.3.6.3 Linking Event Attributes to the Run Mode”, p385.

Also, such operations can be preformed on template attributes which are inherited by an event, or which are referred to through a `TrialAttrib()` function call (see “13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385). See “13.3.11.2 Attribute Inheritance in Factor Format”, p403 for more information on factoring and linking these attributes.

13.3.6 Scripting Events

To define events, you must use a “Events” attribute in the experiment, group, block, or template entry (to which the event should be linked); its value should be a list of references to entries representing the events, called *event entries*. Event entries are required in any experiment design.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, “Hello Event” and “Goodbye Event” are events linked to the “Hello-Goodbye Template”.

The event entry is where a stimulus and its properties are finally specified. The stimulus for the event (or *stimuli*, for certain event types) is taken from the value of the “Stimulus” attribute; the “Stimulus” attribute cannot be inherited (see “13.3.11.2 Attribute Inheritance in Factor Format”, p403). The type of the stimulus is specified in the “EventType” attribute; it defaults to `Text` (and may be inherited). The name of an event defaults to the name of the event entry.

By default, events are executed within the trial in the same order as they are listed in the “Events” attribute; this order can be changed by using “StartRef” attributes with the event entries. (See “13.4.2 Start Reference”, p411.)

Note: Event entries are not accessed from the “Events” attribute as a PsyScript list; the default ordering comes from the way “StartRef” defaults.

For event types which contain sub-stimuli (i.e., `Pasteboard` events), you will need to use the “Stimuli” attribute instead of the “Stimulus” attribute. The “Stimuli” attribute should contain a list of references to *sub-stimulus entries*. These sub-stimuli entries are similar to event entries, except that they use a “StimType” attribute instead of an “EventType” attribute. Also, the standard event attributes are not recognized; only stimulus type-specific attributes are read.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, “Hello Event” is a simple `Text` event. “Goodbye Event” is a `Pasteboard` event linked to `Text` sub-stimuli “Goodbye Stimulus” and “Smiley Stimulus” (the two sub-stimuli are presented simultaneously).

The following attributes are recognized in event entries:

- The standard event attributes; see “13.1.6.3 Standard Event Attributes”, p367. These are inheritable.
- The “Stimulus” attribute or “Stimuli” attribute; “Stimulus” overrides “Stimuli” if both are specified.
- The “Constant” attribute. (See “13.3.6.5 Constant Events in Factor Format”, p387.)
- Stimulus type-specific attributes (inheritable); these are listed by stimulus type in “14.2 Stimulus Types Reference”, p424.
- Any inheritable attribute recognized by a sub-stimulus entry.

The following attributes are recognized in sub-stimulus entries:

- The “Stimulus” attribute.
- Stimulus type-specific attributes (inheritable); these are listed by stimulus type in “14.2 Stimulus Types Reference”, p424.

13.3.6.1 Factoring Event Attributes

Attribute values can be set based on a factor by using the `FactorAttrib()` function in place of a literal value. This is analogous to using **Vary by List** in the graphic environment. (**Vary by List** is implemented in the script by using `FactorAttrib()`).

Factors in PsyScript are described below in “13.3.7 Scripting Factors”, p387. `FactorAttrib()` only works with *free factors*; you cannot use `FactorAttrib()` with factors in a *factor table*.

Note: The the graphic environment reserves the term “factor” for the built-in factors of a factor table. The “free factors” of PsyScript are called “lists” in the graphic environment.

`FactorAttrib()` takes two parameters: the name of a factor and the name of a field within the factor. The factor name must be specified, but the field name defaults to the name of the attribute owning the `FactorAttrib()` call.

In this event entry, for example, the stimulus value is varied with a “Words” factor using its “Noun” field:

```
Noun Event::
  EventType: Text
  Stimulus: FactorAttrib(Words Noun)
  Font: Geneva
  Size: 12
```

The factor-linking requirements for `FactorAttrib()` are the same as for **Vary by List** in the graphic environment; i.e., in order to use `FactorAttrib()`, the specified factor must be

connected to some “ancestor” of the event entry in the experiment hierarchy. This requirement is explained in detail in “Part 2: Graphic Environment Reference, Linking Lists to the Hierarchy”, p135.

13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes

Attribute values in an event entry can be set based on an attribute in the current template, block, or group using the `TrialAttrib()`, `BlockAttrib()`, and `GroupAttrib()` functions, respectively. These correspond to using **Vary by Template**, **Vary by Block**, and **Vary by Group** in the graphic environment.

`TrialAttrib()`, `BlockAttrib()`, and `GroupAttrib()` each take one parameter: the name of an attribute in the template, block, or group entry. Like the field parameter of `FactorAttrib()`, this parameter defaults to the name of the attribute owning the function call. The function call is replaced by the contents of the referenced attribute in the template, block, or group that is being used to compile the current trial.

When superblocks are used in the experiment hierarchy, the lowest-level block is checked first for an attribute referenced by `BlockAttrib()`; if the attribute is not found, the search continues up to the highest-level superblock.

For example, the color of the stimulus in this “Hello Event” is varied by template; it will be red in trials compiled through “Red Template”, and blue in trials compiled with “Blue Template”:

```
Red Template::
  Events: "Hello Event"
  StimColor: Red

Blue Template::
  Events: "Hello Event"
  StimColor: Blue

Hello Event::
  Stimulus: "Hello World"
  Color: TrialAttrib(StimColor)
```

13.3.6.3 Linking Event Attributes to the Run Mode

Attribute values in an event entry can be set based on whether the experiment is being executed in Run or Practice mode. This is done by specifying attributes in the experiment entry with names that start with “Run” and “Practice”, and then using the `RunModeAttrib()` function in the event entry attribute. This corresponds to using **Vary by Run Mode** in the graphic environment.

The `RunModeAttrib()` function takes one parameter: a “base” attribute name. This name is prefixed with “Run” or “Practice” — depending on which mode the experiment is executed in — and then the experiment entry is searched for that attribute; the `RunModeAttrib()` function call is replaced with the attribute’s contents. Like the field parameter of `FactorAttrib()`, `RunModeAttrib()`’s parameter defaults to the name of the attribute owning the function call.

For example, suppose the experiment entry were defined as follows, with “RunColor” and “PracticeColor” attributes:

```
Hello World::
  Format: Factor
  InputDevices: MOUSE KEY
  Timer: Macintosh
  DataFile: "Hello World Data"
  ScaleBlocks: 1
  Events: "Hello Event"
  RunColor: Blue
  PracticeColor: Red

Hello Event::
  EventType: Text
  Stimulus: "Hello World"
  Color: RunModeAttrib(Color)
```

Then, “Hello Event” would print “Hello World” in blue when the experiment is run normally, but it would be in red for practice trials.

13.3.6.4 Factor Format Tags

A tag allows you to record — separate from the script — a value or list of values and associate it to a tag name. Tags are a feature of Factor format which is not available to the graphic interface.

Tags are defined with the `SetTag()` function. The first parameter to `SetTag()` is the tag name; the rest of the parameters are recorded as the value of the tag. `SetTag()` returns the tag value(s) as its result.

The tag value may be used subsequently with the `GetTag()` function call; `GetTag()` takes one parameter — the tag name — and returns the value(s) previously recorded with the tag. It is an error to give `GetTag()` a tag name that has not yet been associated to any values.

For example, consider the compilation of these two events (and assume that “Hello Event” is listed before “Goodbye Event” in the “Events” attribute):

```
Hello Event::
  Stimulus: "Hello World"
  Style: SetTag(EventStyle Geneva 12 "bold italic")

Goodbye Event::
  Stimulus: "Goodbye World"
  Style: GetTag(EventStyle)
```

“Goodbye World” will be printed in the same text style as “Hello World”; the style for “Hello World” is recorded in the tag “EventStyle” and then retrieved for “Goodbye World”.

Tags should be used sparingly since they depend on compilation order, and make the experiment compilation process less optimizable (see “13.3.11.3 Factor Format Optimization”, p404).

13.3.6.5 Constant Events in Factor Format

The Factor format compiler performs special optimizations to avoid unnecessary processing of the script (see “13.3.11.3 Factor Format Optimization”, p404). The “Constant” attribute lets you hand-optimize the compilation by specially marking events which you know are the same from trial to trial.

When an event entry contains a “Constant” attribute with a `True` value, then the event description is only read once from the entry, and all trials which use that event will use the same compilation.

Instead of `True`, you can specify `Trial` or `Block` in the “Constant” attribute; this indicates that the event is constant whenever it is used through the same template or block, although it may be different when it is used different templates or blocks.

Whenever you specify the “Constant” attribute, the Factor format compiler does not check to make sure that the event really would be constant; it simply uses the first instance it compiles for all instances of the event.

13.3.7 Scripting Factors

In the graphic environment, factors are usually represented in a factor table; however, a factor table has a very complex syntax in PsyScript, so scripted experiments generally use *free factors* (a.k.a. *lists* in the graphic environment) instead. Although factor tables *can* be scripted (with significant effort), free factors become much more flexible and powerful when combined with the expressiveness of PsyScript.

To define a free factor, include a “Factors” attribute in the experiment, group, block, or template entry (to which the factor should be linked); its value should be a list of references to entries representing the factors, called *factor entries*.

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the “Word Features” factor is linked to “Superblock1”.

Each factor entry should have a “Levels” attribute with a list of references to entries representing the factor’s levels; these are *level entries*. The level entries will store the actual values for the factor’s fields. (*Compact factors* are different; see “13.3.8 Scripting Compact Factors”, p394.)

Factor entries (for non-compact factors) should contain only a “Levels” attribute (although the “Levels” attribute can have many different sub-attributes related to the order in which levels are selected; see “13.3.7.3 Scripting Access Types”, p391). The fields of a factor are not explicitly defined within a factor entry; they are implicit from values stored in the level entries.

Level entries have one attribute for each field of the factor (the name of each attribute is the same as the field it represents); The value of each attribute determines the field value for the level. (Levels can also have a “Factors”; see “13.3.7.5 Scripting Nested Factors”, p393.)

In the example of “13.3.1 Scripting the Experiment Hierarchy”, p376, the “Word Features” factor contains three levels — “Level1”, “Level2”, and “Level3” — and three fields — “Color1”, “Color2”, and “Face”.

How factors are crossed for each trial depends on a number of sub-attributes of the “Factors” attribute (in the template, block, group, or experiment entry). These sub-attributes control:

- How the factors are grouped into *sets*. This is set by the “Sets” sub-attribute. (See “13.3.7.1 Scripting Factor Sets”, p388.)
- The *access type* of each set. This is set by the “AccessTypes” sub-attribute and other attributes within the factor entries. (See “13.3.7.3 Scripting Access Types”, p391.)
- The *crossing type* of each set. This is set by the “Types” sub-attribute. (See “13.3.7.2 Scripting Crossing Types”, p389.)
- How the cells are weighted. This is controlled by the “BaseCellWeight” sub-attribute and other attributes within the factor or level entries. (See “13.3.7.4 Scripting Cell Weights”, p393.)

13.3.7.1 Scripting Factor Sets

Factor sets determine how factors that linked to the same entry are crossed together. Only factors in the same set are crossed; factors in different sets are independent of each other — i.e. levels are selected from one factor without regard to the current level in the other factor. (Factors linked to different entries will never be crossed together because they cannot be in the same set.)

By default, all factors which are together in the same “Factors” attribute are also in the same set. Explicit factor sets are specified by creating a “Sets” sub-attribute of the “Factors” attribute (in the template, block, group, or experiment entry owning the factors).

The “Sets” sub-attribute should contain a list of numbers — one number for each set of factors. The first number in the list is the number of factors to be crossed together in the first set, the second number is the number of factors for the second set, and so on. (The sum of numbers in the “Sets” sub-attribute should equal the number of factors listed in the “Factors” attribute.) Sets are then made by partitioning the list of factors in the “Factors” attribute: if the first set has n factors, then the first n factors are used for the first set, and so on.

For example, in this template entry, “Factor1” and “Factor2” are in the first set, and “Factor3”, “Factor4”, and “Factor5” are in the second set:

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
  Sets:> 2 3
```


Scripting Factor Set Names

By default, factor sets are named “1”, “2”, etc. Factor sets can be given different names using a “SetNames” sub-attribute of the “Factors” attribute; this attribute should contain a list of names in parallel with the numbers in the “Sets” sub-attribute. Currently, the set name is only used when defining Latin square partitions (see “Scripting Latin Square Partitions”, p390).

For example, we can assign the names “FirstSet” and “SecondSet” to the two sets from the previous example:

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
  Sets:> 2 3
  SetNames:> FirstSet SecondSet
```

13.3.7.2 Scripting Crossing Types

By default, factors in the same set are fully crossed, and a new crossing of levels is selected at the beginning of each trial. These two aspects of a set (which crossings are used and when a new crossing is selected) can be changed by specifying a *crossing type* for the factor set.

The crossing type of a set is configured with a “Types” sub-attribute of the “Factors” attribute (in the template, block, group, or experiment entry); the values of the “Types” sub-attribute are in parallel with the “Sets” sub-attribute (i.e. there is one value in the “Types” list for each number in the “Sets” list).

All of the possible crossing types are described for the graphic environment in “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140; their translations to values in the “Types” sub-attribute are:

- Normal – Same as **Within** in the graphic environment.
- Between – Same as **Between** in the graphic environment.
- Latin – Same as **Latin Square** in the graphic environment.
- List – Same as **List Access** in the graphic environment.
- Use – Same as **Use Access** in the graphic environment.
- UseReset – Same as **Use/Reset Access** in the graphic environment.
- Static – Same as **Fixed** in the graphic environment.

Note: The `Omit` crossing type may show up in scripts built by the graphic environment. This is used to import free factors into a factor table; the graphic environment has to add the imported factor to the standard “Factors” list for bookkeeping purposes. Factor sets with crossing type `Omit` are ignored.

For example, the two sets here are set to crossing types `Normal` and `Latin`:

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
  Sets:> 2 3
  Types:> Normal Latin
```

Scripting Indices

The `Between`, `Static`, and `Latin` crossing types require *index* parameters; these are set through a “Indices” sub-attribute of the “Factors” attribute. The “Indices” sub-attribute should contain a list of number in parallel with the “Sets” (and “Types”) sub-attribute. (The index is ignored for crossing types other than `Between`, `Static`, and `Latin`.)

The keyword `DEFAULT` can be used instead of a number as an index; this value is replaced with the number found in a “SubjectNumber” global entry, or 1 if no such entry is found. This feature is available because the index is commonly linked to the subject number. (See also “Part 2: Graphic Environment Reference, 6.2 Subject Info”, p224.)

If the “Indices” sub-attribute is not found or no value is specified, a value of 1 is always used (contrary to what you might expect, considering the meaning of the `DEFAULT` keyword).

For example, we can use the `DEFAULT` index for the `Latin` set from the previous example:

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
  Sets:> 2 3
  Types:> Normal Latin
  Indices:> 0 DEFAULT
```

Note that we had to include an index for the `Normal` set; this index will be ignored, but we had to include it to stay in parallel with the “Types” sub-attribute.

Scripting Latin Square Partitions

The `Latin` crossing type requires a second level of partitioning within the factor set; this partitioning specifies which factors are fully crossed together, and which are crossed using a Latin square subset of the full crossing. See “Part 2: Graphic Environment Reference, Latin Squares”, p133 for a detailed explanation of these *Latin square partitions*.

Latin square partitions are specified through a “*SetName_LatinSets*” sub-attribute of the “Factors” attribute, where *SetName* is the name of the factor set (which has the `Latin` crossing type).

Unlike the “Sets” sub-attribute, a “*SetName_LatinSets*” sub-attribute contains a list of all of the factors in the set with semi-colons separating factors into partitions.

For example, we can put “Factor3” and “Factor5” together, leaving “Factor4” separate (from the previous example):

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
  Sets:> 2 3
  Types:> Normal Latin
  Indices:> 0 DEFAULT
  2_LatinSets:> Factor3 Factor5 ; Factor4
```

The Latin square partition list also affects the access ordering of the set (see below, “13.3.7.3 Scripting Access Types”, p391); the effect is as if the order of the factors in the “Factors” attribute was changed to the order of the factors in the Latin square partition list (in the “*Set_LatinSets*” sub-attribute).

13.3.7.3 Scripting Access Types

The order in which cells are selected from a factor set depends on the *access type* of the set. The access type is configured through an “AccessTypes” sub-attribute of the “Factors” attribute (in the template, block, group, or experiment entry); the values of the “AccessTypes” sub-attribute are in parallel with the “Sets” sub-attribute (i.e. there is one value in the “AccessTypes” list for each number in the “Sets” list).

All of the possible access types are described for the graphic environment in “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140; their translations to values in the “AccessTypes” sub-attribute are:

`Sequential` – Same as **Sequential** in the interface

`Random` – Same as **Random** in the interface

`RRandom` – Same as **Random with Replacement** in the interface

`CRandom` – Same as **Cycle Random** in the interface

`LRandom` – Same as **Least-Used Random** in the interface

`BRandom` – Same as **Blocked Random** in the interface

`Factor` – Same as **By Factor** in the interface

For example, using the previous example and setting the first set’s access type `CRandom` and second set’s access type to `Random`:

```
Template1::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2 Factor3 Factor4 Factor5
```

```
Sets:> 2 3
Types:> Normal Latin
AccessTypes:> CRandom Random
```

Scripting the Factor-based Access Type

When the “AccessTypes” value for a set is `Factor`, then the access type is determined by attributes in the factor entries — through the “AccessType” (singular) sub-attribute of the “Levels” attribute in each factor entry.

You can learn about the `Factor` access type in two ways: by understanding the `Cross()` PsyScript function (discussed in “12.12 Crossing Lists”, p347), or by translating the graphic environment’s *level access types* (discussed in “Part 2: Graphic Environment Reference, 5.7.2.2 Table Info Dialog”, p140) to PsyScript values.

The `Cross()` PsyScript function is essentially used to order cell-selections; references to the “Levels” attributes of the factor entries are passed to `Cross()`. `Cross()` is discussed in “12.12 Crossing Lists”, p347.

The graphic environment’s level access types translate to PsyScript by mapping level access types to: 1) values of the “AccessType” sub-attribute of a “Levels” attribute, and 2) values of the “Crossing” sub-sub-attribute of “AccessType” sub-attribute:

Sequential – “AccessType” is `Sequential`, “Crossing” is `Always` (default)

Random – “AccessType” is `Random`, “Crossing” is `Always` (default)

Random with Replacement – “AccessType” is `RRandom`, “Crossing” is `Always` (default)

Blocked Sequential – “AccessType” is `Sequential`, “Crossing” is `Force`

Blocked Random – “AccessType” is `Random`, “Crossing” is `Force`

Cycle Random – “AccessType” is `Random`, “Crossing” is `Independent`

Least-Used Random – “AccessType” is `Random`, “Crossing” is `Least`

For example, consider the following script fragment:

```
Templatel::
  Events: Event1 Event2 Event3
  Factors: Factor1 Factor2
  Sets:> 2
  AccessTypes:> Factor

Factor1::
  Levels: Red Green Blue
  AccessType:> Random
  Crossing:>> Force

Factor2::
  Levels: Large Medium Small
  AccessType:> Sequential
```

“Template1” owns a factor set made up of “Factor1” and “Factor2”, where “Factor1” is blocked random and “Factor2” is sequential. Because “Factor1” is listed first, a random level of “Factor1” will be selected (“Red”, “Blue”, or “Green”), and then all of the levels of “Factor2” will be used in order (“Large” then “Medium” then “Small”).

13.3.7.4 Scripting Cell Weights

The effect of cell weights on cell-selection is described in “Part 2: Graphic Environment Reference, Cell Weights”, p133. Weights can be assigned to cells in a factor set by:

- Using the “BaseCellWeights” sub-attribute of the “Factors” attribute (in the template, block, group, or experiment).
- Using the “Weights”, “Mult”, and “Grip” list sub-attributes on the “Levels” attribute in a factor entry.

Values are specified in the “BaseCellWeights” sub-attribute in parallel with the “Sets” sub-attribute (i.e. there is one value in the “BaseCellWeights” list for each number in the “Sets” list). A “BaseCellWeight” value multiplicatively increases the weight on every cell in the factor set. The default “BaseCellWeight” value for a set is 1.

The “Weights”, “Mult”, and “Grip” attributes assign weights to *levels* in a factor; when the factor is crossed with other factors, each resulting cell is weighted according to the product of the weights on the cell’s levels. The way which “Weights”, “Mult”, and “Grip” weight levels is explained in “12.8.4 Weights, Multiple, Grip”, p332.

The interaction of cell weights with the set’s access type and crossing type is more fully explained in “Part 2: Graphic Environment Reference, Cell Weights”, p133.

13.3.7.5 Scripting Nested Factors

Level entries can contain a “Factors” attribute, which has the same syntax as the “Factors” attribute in a template, block, group, or experiment entry. Factors in a set owned by a level are called *nested factors*.

Nested factors behave somewhat differently than regular factors:

- A new crossing is selected in a nested factor set only when the owning level is the current level of its factor.
- Field values of a nested factor are available only to attributes within the owning level; i.e., a `FactorAttrib()` function call can only address a nested factor from within the nested factor’s owner.

Factors can be nested arbitrarily deeply; i.e, levels of nested factors can own other nested factors, and so on.

13.3.8 Scripting Compact Factors

There is an alternate format for the factor entry which does use level entries; this kind of factor is called a *compact factor*.

Note: The standard list object in the graphic environment is implemented as a compact factor.

The structure of a compact factor differs from a regular factor in that:

- The values in the “Levels” attribute are simply read as strings (for the level name), instead of references to level entries.
- There is an “IsList” attribute in the factor entry, which must have the value `True`.
- There is one attribute of the factor entry for each field of the factor. Each of these attributes contains values in parallel with the “Levels” attribute (i.e., there is one value in the “Levels” list for each number in the “Levels” list).

For example, the “Word Features” factor from the example in “13.3.1 Scripting the Experiment Hierarchy”, p376 could be implemented as a compact factor:

```
Word Features::  
  IsList: True  
  Levels: Level1 Level2 Level3  
  Color1: Red Green Blue  
  Color2: Cyan Magenta Yellow  
  Face: "" "Bold" "Italic"
```

If a single field value can have more than one token, a special format must be used for that field’s attribute:

- The field attribute must have a “Multiple” sub-attribute with the value `True`.
- Each value in the content of the attribute should be a reference; the value of the field for each level will be the content of the referenced entry. (Usually, this is implemented by placing square brackets ([]) around each level’s values, thus using inline entries; see also “12.9 Inline Entries”, p335.)

For example, suppose “Word Features” had a “Style” field instead of a “Face” field:

```
Word Features::  
  IsList: True  
  Levels: Level1 Level2 Level3  
  Color1: Red Green Blue  
  Color2: Cyan Magenta Yellow  
  Style: [Geneva 12 ""] [Geneva 12 "Bold"] [Geneva 12 "Italic"]  
  Multiple: True
```

Nested factors cannot be used with the levels of a compact factor.

13.3.9 Scripting Factor Tables

Factor tables are a common way of implementing factors in the graphic environment. In PsyScript, however, their implementation becomes very complex; usually, free factors are used for scripting, instead (see “13.3.7 Scripting Factors”, p387 and “13.3.8 Scripting Compact Factors”, p394).

A *factor table set entry* has the same format as a template entry, with the addition of “FactorTable” and “CrossingValues” attributes. The “FactorTable” attribute defines the structure of the included tables, and the “CrossingValue” attribute stores all of the attribute values dependent on the tables.

Note that a factor table set entry does not look at all like a *factor entry*. In fact, a factor table set entry can include free factors using the “Factors” attribute, just like a template entry. In the experiment hierarchy, factor tables entries are used in the place of template entries. (For example, they are connected to an owner object though the “Templates” attribute.)

13.3.9.1 Scripting Factor Table Structures

The value of the “FactorTable” attribute has the same form as a “Factors” attribute, with the exception that the factors which are listed are not usually references to separate factor entries; instead, most factors in a table are *built-in factors*; each built-in factor is defined through a sub-attribute of the “FactorTable” attribute.

Built-in factors are distinguished from free factors through a “FactTypes” sub-attribute of the “FactorTable” attribute. The “FactType” sub-attribute has values in parallel with the content of the “FactorTable” attribute (i.e., there is one value in the “FactType” list for each number in the “FactorTable” list). Each value in “FactTypes” should be one of:

External – the corresponding item in the “FactorTable” list is a free factor imported into the table (i.e., the factor name in the “FactorTable” attribute is a reference to a factor entry).

Normal – the corresponding item in the “FactorTable” list is a built-in table factor (i.e., the factor is described in a sub-attribute of the “FactorTable” attribute). This is the default.

For each built-in factor, there should be a sub-attribute of “FactorTable” (the factor’s name is used for the sub-attribute’s name). The value of each of these sub-attributes should be a list the factor’s levels. (These level names never refer to separate level entries.)

For example, here is a factor table entry with two factors, “Color” and “Face”:

```
Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face
             FactTypes:> Normal Normal
             Color:> Red Green Blue
             Face:> Plain Bold Italic
  CrossingValues:
  ...
```

There are no fields or values associated to built-in factors; the way in which other values depend on the factor is encoded in the “CrossingValues” attribute of the factor table entry (see below, “13.3.9.2 Scripting Factor Table Crossing Values”, p396).

Like the “Factors” attribute, the “FactorTable” attribute can specify a number of distinct factor sets. All of the sub-attributes which apply to the “Factors” attribute — for defining factor sets and setting crossing types, access types, etc. — also apply to the “Factor Table” attribute. (See “13.3.7 Scripting Factors”, p387.)

Scripting Nested Factors in a Factor Table

Although the levels of a built-in factor are never represented by separate level entries, factor tables *can* be nested within built-in levels.

To do this, the sub-attribute for the built-in factor should have a sub-sub-attribute with the same name as the nesting level; this sub-sub-attribute has the same format as the “FactorTable” attribute; in this way, tables can be nested recursively to any depth.

For example, we could nest a “Shade” factor within the “Red” level of the “Color” factor from the previous example:

```
Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face
    FactTypes:> Normal Normal
    Color:> Red Green Blue
      Red:>> Shade
        FactTypes:>>> Normal
        Shade:>>> Standard Pink
      Face:> Plain Bold Italic
  CrossingValues:
  ...
```

A new crossing for a nested factor is only selected when the owning level is current (as with nested factors of free factor sets).

13.3.9.2 Scripting Factor Table Crossing Values

When free factor sets are used, the normal attributes in the template, event, and sub-stimuli entry reference the factors by using `FactorAttrib()` (see “13.3.6.1 Factoring Event Attributes”, p384). When a factor table is used, however, a different method is used for linking an attribute to its table-dependent value.

All of the attribute values which are determined by the table are stored in the “CrossingValue” attribute of the factor table entry. There is no explicit link to a factor within the template, event, or sub-stimulus entry to which the attribute belongs.

For each entry linked to the factor table set entry, there is a sub-attribute of the “CrossingValue” attribute. This sub-attribute has the same name as the entry it represents (i.e., the table set entry itself as a template or an event or sub-stimulus entry owned by the template). Within each of these sub-attributes, the attribute structure of the “actual” entry is mimicked by sub-sub-attributes. Finally, the values for the attribute in a particular cell are listed as

sub-sub-sub-attributes; these sub-sub-sub-attributes are the *value attributes*, and the cell combination that owns the value is encoded in this value attribute's name.

For example, here is a factor table set entry which owns a single event entry. The color of the event stimulus depends on the cell which is current in the factor table:

```

Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face
    Color:>
      Levels:>> Red Green Blue
    Face:>
      Levels:>> Plain Bold Italic
  CrossingValues:
    "Hello Table":>
    "Hello Event":>
      Stimulus:>>
      Color:>>
        1_Red____:>>> Red
        1_Green____:>>> Green
        1_Blue____:>>> Blue

Hello Event::
  Stimulus: "Hello World"

```

The value attributes “1_Red____”, “1_Green____”, and “1_Blue____” contain final values for the “Color” attribute of the “Hello Event” entry; these values apply when different cells are currently selected from the “Color” x “Face” table (as described in the next section).

Value Attribute Names

A single value attribute can correspond to one cell or to many. However, for each factor, it must correspond to either all of the levels of the factor, or just one. (This means that the cells represented by a value attribute must form a “hyper-column” in the factor table.)

A value attribute's name starts with a table name (factor set name; see “13.3.7.1 Scripting Factor Sets”, p388). The rest of the name is built by combining the names of the levels which determine the value's cell(s). When the value applies to all of the levels in a factor, an underscore (“_”) is used as a placeholder for that factor in the value attribute name. All of the level names and placeholders are concatenated together using the underscore as a separator. A trailing underscore is always present

Thus, the value attribute name “1_Red____” is derived from using the “Red” level from the “Color” factor, and the placeholder “_” for the “Face” factor. Extra underscores are interspersed as separators and the set name “1” is used to give “1_Red____” (three trailing underscores). This names the two cells in the “Red” column of the factor table.

An attribute value that only applied when both “Red” and “Bold” are current (the cell in “Red”'s column and “Bold”'s row) would have the name “1_Red_Bold_”.

The order in which the level names are concatenated must match the order in which their factors are listed in the “FactorTable” attribute. This is why “Red”, “Green”, and “Blue” always appear before “Bold” or “Italic” in the current set of examples.

If an attribute value does not depend on any factor — i.e., it is the default for all cells — it is usually stored in the regular attribute location (within the event entry, for example, instead of in the hierarchy below “CrossingValues”); however, it can also be specified with the other table values by using the cell name “_”. A set name followed by underscores for all of the factors is *not* allowed.

Multiple Factor Tables

Each value attribute only applies to a single table within a factor table set. Consider this example:

```
Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face Size Font
    Sets:> 2 2
    Color:> Red Green Blue
    Face:> Plain Bold Italic
    Size:> 12 24
    Font:> Pretty Ugly
  CrossingValues:
    "Hello Table":>
    "Hello Event":>
      Stimulus:>>
      Color:>>
        1_Red__:>>> Red
        1_Green__:>>> Green
        1_Blue__:>>> Blue
      Font:>>
        2__Pretty_:>>> Geneva
        2__Ugly_:>>> Chicago

Hello Event::
  Stimulus: "Hello World"
```

This factor table set entry contains two factor tables: “Color” x “Face” and “Size” x “Font”. These tables are specified the same way that separate factor sets are specified: by using the “Sets” sub-attribute (see “13.3.7.1 Scripting Factor Sets”, p388).

In the “CrossingValues” attribute, there are value attributes with names like “1_Red__” and “2__Pretty_”. No value attribute could have a name like “?_Red_Pretty_”, because the “Color” and “Font” attributes are not crossed with each other (i.e., they are not in the same factor table).

Factor Table Value-Finding Algorithm

Using a factor table, it is possible that more than one value could be specified for the same cell. For example, consider this factor table:

```
Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face
    Color:> Red Green Blue
    Face:> Plain Bold Italic
  CrossingValues:
    "Hello Table":>
    "Hello Event":>
      Stimulus:>>
```

```

Color:>>
  1_Red____:>>> Red
  1_Blue____:>>> Blue
  1_Blue_Bold_:>>> Purple
  1_____Italic:>>> Yellow

```

```

Hello Event::
  Stimulus: "Hello World"
  Color: Black

```

When we are in the “Blue”-“Bold” cell, there are three possible values that could apply for the color of the stimulus: Blue from the “1_Blue____” value attribute, Purple from the “1_Blue_Bold_” value attribute, and “Black” from the “Color” attribute in “Hello Event”.

In such cases, the *most specific* attribute value is used. The *most specific* value is the one that depends on the level-selection of the greatest number of factors (as opposed to defaulting across all levels in factors). Thus, Purple value will be used because “1_Blue_Bold_” is constrained by levels in two factors, while “1_Blue____” depends on level-selection in only one factor and “Color” in “Hello Event” depends on zero factors.

This means that you can supply defaults for large collections of cells, and then refine the value for smaller collections. In most cases, a default value is put in the template, event, or sub-stimulus entry and then it is overridden in certain cells by value attributes.

In some cases, there may be no most specific attribute. For example, there is no most specific color in the previous example for the “Blue” x “Italic” cell; both “1_Blue____” and “1_____Italic” depend on one factor. In this case, the result is undefined (one of the values will be picked “randomly”). Such a situation generally implies an incorrect or incomplete design.

13.3.9.3 Nested Factor Values in a Factor Table

When a value attribute name contains a level that owns nested tables, then an additional string must be inserted into the value attribute name. This additional string specifies a cell in the *nested* table in the normal way; it is inserted in the value attribute name right after the owning level’s name.

For example, consider this factor table:

```

Hello Table::
  Events: "Hello Event"
  FactorTable: Color Face
    FactTypes:> Normal Normal
    Color:> Red Green Blue
    Red:>> Shade
      FactTypes:>>> Normal
      Shade:>>> Standard Pink
    Face:> Plain Bold Italic
  CrossingValues:
    "Hello Table":>
    "Hello Event":>
      Stimulus:>>
      Color:>>
        1_Red_____:>>> Red

```

```
1_Red_Pink____:>>> Magenta
1_Green____:>>> Green
1_Blue____:>>> Blue
```

The “1_Red_____” (five trailing underscores) value attribute applies when “Red” is the current level of “Color” regardless of the cell selected in “Shade”. “1_Red_Pink____” (three trailing underscores) applies only when “Red” is the current level of “Color” *and* “Pink” is the current level of “Shade”.

13.3.9.4 Scripting the Factor Set Scope

When a factor table set is linked to multiple owning entries, its tables have an independent access history for each owner. Further, when there is more than one “path” of links back to the experiment entry, each path produces an independent table access history. This property — called the *scope* of a factor set — is explained more fully in “Part 2: Graphic Environment Reference, Factor Set Scope”, p136.

The scope of a factor set *in a table* can be controlled by using the “SetScopes” sub-attribute of the “Factors” attribute (in the template, block, group, or experiment entry); values are specified in the “SetScopes” sub-attribute in parallel with the “Sets” sub-attribute (i.e. there is one value in the “SetScopes” list for each number in the “Sets” list). The “SetScopes” attribute does *not* work for sets of free factors.

The possible values for the “SetScopes” attribute are:

Template – This is the default value. Every set from a different entry or different path keeps an independent access history.

Block – The access history is common for every set (containing the same factors) with a path that goes through this set’s block (the lowest-level block, when multiple levels of blocks are used).

Group – The access history is common for every set (containing the same factors) with a path that goes through this set’s group.

For example, in this script fragment, “Factor1” is crossed with “Factor2” in trials for both “Block1” and “Block2”, but the access histories are different:

```
Experiment::
  ...
  Blocks: Block1 Block2
  ...

Block1::
  Templates: Table1
  ...

Block2::
  Templates: Table1
  ...

Table1::
  ...
```

```

FactorTable: Factor1 Factor2
Sets:> 2
...

```

The access histories can be merged by changing “Table1”:

```

Table1::
...
FactorTable: Factor1 Factor2
Sets:> 2
SetScopes:> Group
...

```

13.3.10 Scripting Factor Format Trial Counts

The intricacies of trial counting in Factor format are explained in “Part 2: Graphic Environment Reference, 5.12.2 Trial Counting”, p211. Here, we specify which attributes are used to hold the numbers described there.

- If an experiment has groups, no trial counting attributes appear in the experiment entry.
- When an experiment, group, or superblock entry owns blocks, it can have the following attributes:
 - “Cycles” – This attribute specifies how many times to pass through the entire list of blocks. The default is 1.
 - “PracticeCycles” – Like “Cycles”, but this attribute will override “Cycles” when the experiment is executed in Practice mode.
 - “ScaleBlocks” – Specifies the scaling factor to be applied to scalable counts in the owned blocks.
 - “PracticeScaleBlocks” – Like “ScaleBlocks”, but this attribute will override “ScaleBlocks” when the experiment is executed in Practice mode.
- Superblock entries can have a “FixedCycles” (and “PracticeFixedCycles”) attribute instead of a “Cycles” attribute, to make the number of passes through its blocks unscalable. (I.e., a “ScaleBlocks” attribute in an entry owning the superblock will have no effect.)
- When an experiment, group, or block entry owns trials or events, it can have the following attributes:
 - “Cycles” – This attribute specifies how many trials to execute in the experiment or block. The default is 1. In a block entry, this value can be scaled by a “ScaleBlocks” attribute in the block’s owner.
 - “PracticeCycles” – Like “Cycles”, but this attribute will override “Cycles” when the experiment is executed in Practice mode.

- “BlockDuration” – This attribute can be used instead of “Cycles”; it specifies how long trials should be executed (in seconds). When a trial ends after this many seconds, no more trials will be executed within the experiment or block. The “Cycles” attribute overrides this attribute. Block durations are never scaled.
- “PracticeBlockDuration” – Like “BlockDuration”, but this attribute will override “BlockDuration” when the experiment is executed in Practice mode. “PracticeCycles” overrides this attribute.
- Block entries which own templates or event can have a “FixedCycles” (and “Practice-FixedCycles”) attribute instead of a “Cycles” attribute, to make the number of trials to be executed in the block unscalable. (I.e., a “ScaleBlocks” attribute in an entry owning the block will have no effect.)

13.3.11 Technical Details of Factor Format Scripting

13.3.11.1 Structural vs. Non-structural Attributes

In discussing the use of special attribute tools — such as `FactorAttrib()`, `BlockAttrib()`, and inheritance — it is sometimes necessary to distinguish between *structural attributes* and *non-structural attributes*. The effective difference is that these tools cannot be used with structural attributes, only with non-structural attributes.

Structural attributes define the structure of the experiment as a whole; this structure must be read in from the script *before* the special attribute tools can be used for other attributes. The following attributes are structural:

- All standard experiment, block, and group attributes, including the “Groups”, “Blocks”, “Templates”, and “Factors” attributes.
- The “Factors” attributes of a template entry.
- The “FactorTable” attribute of a factor table set entry, and any attribute below it.
- Any factor attributes, except field-value attributes in a compact factor.
- The “Factors” attribute of a level entry (for nested factors).
- Any list-access specifications, such as “AccessType”.
- Any factor set attributes, including “Sets”, “Types”, and “AccessTypes”, etc.
- Any trial counting attributes.

All other attributes in an experiment definition are ***non-structural attributes***. This includes the standard trial attributes, all event and stimulus attributes, and factor field values.

The “Events”, “Stimulus”, and “Stimuli” attributes are special in that they can be factored or linked to template, block, or group attributes, but they *cannot* be inherited.

13.3.11.2 Attribute Inheritance in Factor Format

Attribute inheritance for Factor format is explained in “Part 2: Graphic Environment Reference, 5.8.1.1 Attribute Inheritance”, p150. For the most part, there is nothing special that needs to be said about inheritance in the translation to PsyScript. However, with respect to the interaction of `FactorAttrib()` et al. and inheritance, Factor format actually allows more flexibility than the graphic environment.

In the graphic environment, you are never allowed to link to a list with **Vary by List** when the list is not connected to the object or not in all of the object’s paths back to the experiment object (see “Part 2: Graphic Environment Reference, Linking Lists to the Hierarchy”, p135). This insures that no run-time errors will occur due to an out-of-context reference to a factor.

However, it is possible for such a reference to be made which does not cause any run-time errors; consider this example script fragment:

```
Block1::
  Templates: "Hello Template" "Goodbye Template"
  Color: FactorAttrib(WordColors Color)

Hello Template::
  Factors: WordColors
  Events: "Hello Event"

Hello Event::
  Stimulus: "Hello World"

Goodbye Template::
  Events: "Goodbye Event"

Goodbye Event::
  Stimulus: "Goodbye World"
  Color: Black
```

This script fragment could not have been generated by the graphic environment, because “Block1” references the “WordColors” factor, but the “WordColors” factor is *below* “Block1” in the hierarchy. The “WordColors” factor is not always available during “Block1” (specifically, when “Goodbye Template” is used for the trial).

However, this is a perfectly legal script. The reason is that “Color” is evaluated only when it is inherited by “Hello Event”, in which case the “WordColors” factor is available through “Hello Template”.

Similar cases can happen with other interactions among inheritance, `TemplateAttrib()`, `BlockAttrib()`, `GroupAttrib()`, and `FactorAttrib()`. Since an attribute will only be evaluated when it is used, an expression is valid when it valid in all states for which it is relevant.

When the `TemplateAttrib()`, `BlockAttrib()`, or `GroupAttrib()` function call is used, the attribute that is referenced must be defined directly in the template, block, or group; it cannot be inherited.

13.3.11.3 Factor Format Optimization

The Factor format compiler performs special optimizations to avoid unnecessary processing of the script. The degree to which the compiler tries to optimize can be controlled through the “Optimization” experiment attribute; it can have any of the following values:

`False` – No optimizations are performed. (This is the same as **No optimization** in the graphic environment.)

`Constant` – This is the default. Only events which are marked as constant will be optimized (see “13.3.6.5 Constant Events in Factor Format”, p387). (This is the same as **Optimize constant-declared events only** in the graphic environment.)

`Standard` – Full optimization. (This is the same as **Standard optimization** in the graphic environment.) A compiled event can be reused when:

- The script has not been modified (during compilation) in a way which can affect any value read for the event, and
- The factors which are used by the event’s attributes (not counting the “Stimulus” attribute) have the same current levels, and
- The same hierarchical path was used to reach the event (i.e., the same blocks and template are used), and
- No `Access()`, etc. list-based function calls are made when compiling the event, and
- No tag values are set or used, through `SetTag()` or `GetTag()` function calls.

Optimized events can be re-used even if the stimulus changes (it is the non-“Stimulus” attributes which are optimized).

13.3.11.4 Factor Format Compilation Order

The elements of a Factor format script are compiled in the following order:

- The experiment structure is read, starting with the experiment entry and working down to the templates. Factor sets are read at each hierarchical level along the way. The hierarchy is followed depth-first in the order which links are listed in the linking attributes (“Groups”, “Blocks”, “Factors”, etc.). Groups which are not used are not compiled.
- The experiment attributes are read.

- For each trial:
 - The block and template are selected, and then the crossing is selected for all of the relevant factor sets.
 - The trial attributes are read.
 - All of the event's stimuli are read.
 - For each event, the stimulus attributes are read, and then the event attributes.
 - The trial is executed and trial variables may be written back to the script.

13.3.12 Summary of Factor Format

The following is a brief summary of the structure of a Factor format experiment. First, the different entry types are listed, sometimes referring to groups of object attributes (e.g., *ExperimentAttribs*) or ownership attributes (e.g., *OwnedGroupsDescription*); afterwards, the object and ownership attribute groups are detailed.

13.3.12.1 Factor Format Entry Types Summary

Experiment Entry Definition

```

Experiment1::
  ExperimentAttribs
  TrialAttribs
  EventAttribs

  OwnedGroupsDescription
  or
  OwnedBlocksDescription
  or
  OwnedTemplatesDescription
  or
  CycledOwnedEventsDescription

  OwnedFactorsDescription

```

Group Entry Definition

```

Group1::
  ExperimentAttribs
  TrialAttribs
  EventAttribs

  OwnedGroupsDescription
  or
  OwnedBlocksDescription
  or
  OwnedTemplatesDescription
  or

```

OwnedCycledEventsDescription

OwnedFactorsDescription

Block Entry Definition

```
Block1::
  TrialAttribs
  EventAttribs

  OwnedBlocksDescription
  or
  OwnedTemplatesDescription
  or
  OwnedCycledEventsDescription

  OwnedFactorsDescription
```

Template Entry Definition

```
Templatel::
  TrialAttribs
  EventAttribs

  OwnedEventsDescription

  OwnedFactorsDescription
```

Factor Table Entry Definition

```
Table1::
  FactorTableDescription

  CrossingValues:
    owned entry name>>
    attribute of owned entry:>>
    name of cell of attribute dependence:>>>
    ...
  ...

  TrialAttribs
  EventAttribs

  OwnedEventsDescription

  OwnedFactorsDescription
```

Event Entry Definition

```
Event1::
  Stimulus: ...
  or
  Stimuli: list of stimulus entries

  Constant: True, Block, Trial, or False (defaults to False)

  OwnedEventsAttribs
```

Factor Entry Definition

Factor1::
 Levels: *list of level entries or level names (in compact form)*
 AccessType:> *access type*
 Crossing:>> *always, force, indep, or least*

IsList: True or False

other attribs for fields (in compact form)

Level Entry Definition

Level1::
field attributes

OwnedFactorsDescription (for nested factors)

13.3.12.2 Factor Format Summary Object Attribute Groups***ExperimentAttribs***

Title: ...
 Format: ...
 InputDevices: ...
 Timer: ...
 Flags: ...
 DataFile: ...
 Instructions: ...
 DataHeader: ...
 DataFields: ...
 RunLabel: ...
 DataRecordSeparator: ...
 Resources: ...
 NumTrialsPerRest or NumRestPeriods: ...
 RestPeriod: ...
 Precompile: ...
 ExpVariables: ...
 DataVariables: ...
 Instructions: ...
 Debrief: ...
 Reset: ...
 RunMode: ...
 Optimization: ...
other device type-specific attributes

TrialAttribs

ConditionName: <condition name template>
 MinLoadTime: ...
 TrialActions: ...

EventAttribs

EventName: ...
 EventType: ...
 EventTag: ...
 Duration: ...
 StartRef: ...
 EventActions: ...
other stimulus type-specific attributes

13.3.12.3 Factor Format Summary Description Attribute Groups

OwnedGroupsDescription

Groups: *current group entry name*
Current:> *index of current group*

OwnedBlocksDescription

Blocks: *list of block entries*
AccessType:> *access type*
Cycles or FixedCycles: *number of repetitions of the block list*
ScaleBlocks: *number of repetitions of the block*

OwnedFactorsDescription

Factors: *list of factor entries*
FactorSetSubattributes

FactorSetSubattributes

Sets:> *set counts*
SetNames:> *names of sets*
Types:> *crossing type for each set*
Indices:> *indices used by some crossing types*
SetName_LatinSets:> *for each set with Latin crossing type*
AccessTypes:> *access type for each set*
SetScopes:> *Template, Block, or Group for each set*
BaseCellWeights:> *multiplying weight for all cells in set*

FactorTableDescription

FactorTable:
FactorSetSubattributes
FactTypes:> *list of Normal or External for each factor*
built-in factor name:>
Levels:>> *levels for built-in*
built-in level name:
FactorTableDescription (for nested)
...
...

OwnedTemplatesDescription

Cycles: *exact number of trials for block*
or
FixedCycles: *scalable number of trials for block*
or
BlockDuration: *number of seconds to run trials*

Templates: *list of template and factor table entries*
AccessType:> *access type*

OwnedEventsDescription

ConditionName: *condition name template*
Events: *list of event entries*

CycledOwnedEventsDescription

Cycles: *exact number of trials for block*
or
FixedCycles: *scalable number of trials for block*

EventDescription

13.4 Complex Attribute Formats

13.4.1 Action Lists

Actions are explained in “Part 2: Graphic Environment Reference, 5.9 Conditions and Actions”, p192. Although this description is geared towards Factor format and the graphic environment, the ideas are the same regardless of format.

Trial actions are specified in an “TrialActions” attribute; the placement of the “TrialActions” attribute depends on the script format that is used; see “13.2.1.1 Trial Attributes in StimList Format”, p370 and “13.3.5 Scripting Templates”, p381.

Event actions are specified in an “EventActions” attribute; the placement of the “EventActions” attribute depends on the script format that is used; see “13.2.1 StimList Format”, p368, “13.2.2 EventList Format”, p371, and “13.3.6 Scripting Events”, p383.

13.4.1.1 Specifying Action Lists

The value of an “EventActions” or “TrialActions” attribute consists of a list of conditions-actions pairs. Each pair consists of a list of conditions — such as input device states — and a list of actions that will be triggered when/if any of these conditions becomes true.

A *condition* is described by an inline entry named the same as the condition type — Start, End, When, ScriptWhen, or an input device name — and with the condition parameters as its content. (Such an inline entry has the following form: a condition type followed — without spaces — by square brackets containing the condition parameters. See also “12.9 Inline Entries”, p335.) The format of the parameters depends on the condition type; see “14.3 Conditions and Inputs”, p437 for parameters of each type.

An *action* is described by an inline entry named after the action name and with the action parameters as its content. (Such an inline entry has the following form: an action name followed — without spaces — by square brackets containing the action parameters. See also “12.9 Inline Entries”, p335.) The list of all possible actions and the parameters that each takes are listed in “14.1 Actions Reference”, p419.

A *condition-action pair* is made up of a reference for the list of conditions, an equals-greater-than arrow (“=>”), and a reference for the list of actions. Usually, inline entries are used to contain the conditions and actions list, so that the condition-action pair looks like this:

```
[Condition1[params...] Condition2[params...] ...] =>
[Action1[params...] Action2[params...] ... ]
```

Within

For example, this event entry specifies that “correct beep” should be sounded when the mouse button is clicked, and “incorrect beep” should be sounded when a key is pressed:

```
Hello Event::  
  Stimulus: "Hello World"  
  EventActions: [Mouse[Clicked]] => [Beep["correct beep"]]  
               [Key[Any]] => [Beep["incorrect beep"]]
```

When the graphic environment creates condition-action pairs, it names the conditions reference “Conditions” and actions reference “Actions”, like this:

```
Hello Event::  
  Stimulus: "Hello World"  
  EventActions: Coonditions[Mouse[Clicked]] => Actions[Beep[]]
```

These names are ignored, however; they simply provide extra guidance for someone looking at the script.

13.4.1.2 Instances and ActiveUntil

The *instances* and *active until* settings of an action are described in “Part 2: Graphic Environment Reference, 5.9 Conditions and Actions”, p192. The instances setting is defined by using an “Instances” attribute in the action-describing inline entry. Similarly, the active until setting is defined by using an “ActiveUntil” attribute in the action-describing inline entry.

The value of the “Instances” attribute is a number. The value of “ActiveUntil” should be one of:

- Blank; this implies the default, which is to remain active until the end of the posting event. (This is the same as **End of This Event** in the graphic environment)
- The name of any event; the action will remain active until the end of that event. (This is the same as **End of [Event]** in the graphic environment)
- The `TRIAL_END` keyword; the action will remain active until the end of the trial. (This is the same as **End of Trial** in the graphic environment)
- the `FORCE_ONE` keyword; the action will remain active until at least one instance of the action are triggered. (This is the same as **At Least One Instance** in the graphic environment).
- the `FORCE_ALL` keyword; the action will remain active until all of the allowed instances of the action are triggered. (This is the same as **All Instances** in the graphic environment)

For example, we can force the subject to click the mouse five times for five beeps in “Hello Event”:

```
Hello Event::  
  Stimulus: "Hello World"  
  EventActions: [Mouse[Clicked]] =>  
               [Beep["correct beep"]  
               Instances: 5 ActiveUntil: FORCE_ALL]]
```

Because they are attributes in the inline entry, both “Instances” and “ActiveUntil” must come after the regular parameters, but their relative ordering does not matter.

13.4.2 Start Reference

13.4.2.1 Start Reference Format

A *start reference* defines a point of time during the execution of a trial. Usually, a start reference is used in the “StartRef” event attribute to specify the starting time of an event; however, start references are also used by various built-in actions.

A start reference will have on of the following forms:

- "*msecs* after start of event"
- "*msecs* after end of event"
- "*msecs* after start"

where *msecs* is a number representing a time in milliseconds and *event* is the name of an event in the trial (see “13.2.4 StimList/EventList Event Names”, p373 or “13.3.6 Scripting Events”, p383). A start reference should be quoted in the script, since it always contains spaces. (A “StartRef” attribute can be read even if the start references is not quoted, but it will be incompatible with the graphic environment.)

An event is allowed to have a “StartRef” value relative to itself, but it will never be executed unless it is started with a RunEvent[] action (see “RunEvent[Event]”, p419).

13.4.3 Duration

Event durations are described in “Part 2: Graphic Environment Reference, Duration”, p179. In the script, an event duration is assigned through the “Duration” attribute; it’s value should have one of the following forms:

- *msecs*, where *msecs* is a number representing a fixed duration in milliseconds
- Self_Terminate
- Trial_End
- A conditions list — specified the same as a list of action conditions (see “13.4.1.1 Specifying Action Lists”, p409) — plus an optional Time[*msecs*] specification for a timeout.

For example, this “Hello Event” lasts until the mouse button is clicked, or until five seconds pass:

```
Hello Event::  
  Stimulus: "Hello World"  
  Duration: Mouse[Click] Time[5000]
```

13.5 Trial Manager Variables

Trial Manager Variables (or *trial variables*) are named variables defined by entries in the script. Trial variable values can be used and changed by the Trial Manager while it is executing an experiment (independently of the compiling process). Trial variables allow data collected during the run of an experiment to be written to the script and used in subsequent trials or later runs of the experiment. See also “Part 2: Graphic Environment Reference, 5.10 Trial Manager Variables”, p205.

13.5.1 Declaring Variables

Before the Trial Manager can use a variable, it must be declared in the script. The declaration tells the Trial Manager the *type*, *initial value*, and *update type* of the variable.

13.5.1.1 Variable Types

Each variable has a *type*, which the Trial Manager uses to determine how to interpret the value of the variable. There are three built-in *primitive types*:

Integer: an integer in the range -2^{15} to $2^{15} - 1$

Long_integer: an integer in the range -2^{31} to $2^{31} - 1$

Float: a rational number

These types can be assigned to a variable, or they can be used to define any more complicated *composite types*, as described in “13.5.1.3 Composite Types”, p413. There are other predefined types which are composites; these are described in “Built-in Composite Types”, p416.

13.5.1.2 Variable Declaration Entries

A variable is defined through an entry of the form

```
VariableName:: value  
  Type: type  
  Init: initial-value  
  Update: update-between
```

Where:

VariableName is the name of the variable.

value is the current value(s) of the variable. When the variable definition is read in, this value is used for the variable's initial value, or it can be overridden with the "Init" attribute. If the value of "Update" is `True`, the current value of the variable is written here after each trial is executed. The variable value is always updated at the end of an experiment run.

type is the type of the variable — one of the built-in types, `Array` for an array (see "13.5.1.3 Composite Types", p413), `Record` for a record (see "13.5.1.3 Composite Types", p413), or a type defined in the script (see "13.5.1.4 Type Declarations", p415).

initial-value is a value(s) to which the variable is set at the beginning of each experiment run. If the "Init" attribute is omitted, *value* is used instead. See also "Initializing Composite Types", p414.

update-between is `True` if the variable value should be written back to the script after each trial is executed.

VariableName and *type* are required; the rest of the form is optional.

For each variable, a reference to its defining entry must be placed in the "ExpVariables" experiment attribute. Variable definitions which are not referenced in the "ExpVariables" attribute will be ignored.

13.5.1.3 Composite Types

By combining primitive types, more complicated variable types can be defined in the script; these are called *composite types*. The two kinds of composite types are *arrays* and *records*.

An *array* is a numbered list of variables having the same data type; any item of the list — or *element* — can be retrieved using its position — or *index* — in the list.

A *record* is a collection of variables of several different types. Each part of a record — called a *field* — is identified by a name.

Array variables are defined by using the `Array` keyword in the "Type" attribute of the variable's definition. The "Type" attribute should then have sub-attributes "Type" and "Count", which specify the type of the elements of the array and the number of elements of the array, respectively.

The following example specifies a variable called "MyArray" which is an array of 10 integers.

```
MyArray::
  Type: Array
      Type:> Integer
      Count:> 10
```

Record variables are defined by using the `Record` keyword in the "Type" attribute of the variable's definition. The "Type" attribute should then have one sub-attribute for each field

in the record; the name of the sub-attribute should be the same as the name of the field, and the value of the attribute should be the field's type.

The following example defines a record variable called "MyRecord", with fields "I", "N", and "X" whose types are `Integer`, `Long_Integer`, and `Float`, respectively.

```
MyRecord::
  Type: Record
      I:> Integer
      N:> Long_Integer
      X:> Float
```

It is possible to compose variables of arbitrary complexity in this manner. Array elements can be other arrays or records, and record fields can be arrays or other records. To create multi-dimensional arrays (matrices) simply create arrays of arrays, as many levels deep as necessary.

In this example, "ComplicatedVar" is a 3x4 matrix of records, where each record contains an array and another record.

```
ComplicatedVar::
  Type: Array
      Count:> 3
      Type:> Array
          Count:>> 4
          Type:>> Record
              AField:>>> Array
                  Count:>>>> 10
                  Type:>>>> Integer
              RField:>>> Record
                  X:>>>> Float
                  Y:>>>> Long_Integer
```

Initializing Composite Types

To initialize a scalar value, you place the value in the content of the variable declaration entry or in the "Init" attribute. With composite types, however, there is more than one value which has to be assigned, so the syntax is more complex.

Arrays of scalars are initialized by simply specifying a list of values instead of a single value.

Records are initialized by giving attributes to the variable declaration entry or sub-attributes to the "Init" attribute, where each field is represented by an attribute with its name and the value for the field is in the attribute's content.

Arrays of non-scalars (records or other arrays) are specified by a list of references; each reference should contain initialization values in the format appropriate to the item type.

For example, "Coordinates" is a record with X and Y positions, initialized to (0,1):

```
Coordinate::
  Type: Record
      X: Integer
      Y: Integer
  X: 0
  Y: 1
```

In this example, “CoordinateArray” is an array of X-Y coordinates; it is initialized to the list (0,1), (2,3), (3,4) using the “Init” attribute:

```
CoordinateArray::
  Type: Array
    Count:> 3
    Type:> Record
      X:>> Integer
      Y:>> Integer
  Init: [X:0 Y:1] [X:2 Y:3] [X:3 Y:4]
```

In this example, scripting tricks are used to initialize the 10x10x10 array “BigArray” to all 1’s:

```
BigArray::
  Type: Array
    Count:> 10
    Type:> Array
      Count:>> 10
      Type:>> Array
        Count:>>> 10
        Type: Integer
  Init: 10 ~ [10 ~ [10 ~ 1]]
```

13.5.1.4 Type Declarations

It is possible to define your own composite types, so that variable definitions can be given a simple type name, rather than given a complete composite definition. This is especially useful when you have multiple variables with the same composite type.

The format of a type declaration is the same as the format of a variable declaration, without the value, update state, or initialization information (i.e., without content values, the “Update” attribute, or the “Init” attribute). Type declaration entries must be referenced in an “ExpTypes” experiment attribute.

For example, suppose in your script you have a 10x10x10 matrix of integers, and several variables that keep track of positions in that matrix by keeping x, y, and z coordinates. Sections of your script could look like this:

```
MyExperiment::
  .
  .
  .
  ExpTypes: MatrixType CellType
  ExpVariables: MatrixA MatrixB CellArrayM CellArrayN
                TempCell
  .
  .
  .

#
#10x10x10 Matrix of Integers
#
MatrixType:: Array
  Count: 10
```

```
    Type: Array
    Count:> 10
    Type:> Array
        Count:>> 10
        Type:>> integer

#
# A Cell
#
CellType:: Record
    x: integer
    y: integer
    z: integer

.
.
.

MatrixA::
    Type: MatrixType

MatrixB::
    Type: MatrixType

TempCell::
    Type: CellType

CellArrayA::
    Type: Array
    Type:> CellType
    Count:> 5

CellArrayB::
    Type: Array
    Type:> CellType
    Count:> 7
```

Built-in Composite Types

A common use of Trial Manager Variables is making data collected during the run of the experiment available in the script for building new trials. For this purpose, there are three built-in composite variable types: Point, Input, and Response.

Point

The Point type stores vertical and horizontal coordinates. It is defined as follows:

```
Point:: Record
    v: Integer # Vertical
    h: Integer # Horizontal
```

Input

The Input type stores information about the current state of the input devices. It is defined as follows:

```
Input:: Record
    InputType: Integer      # Type of input
    Time: Long_Integer     # Time of input
    state: Integer         # BBox state on input
```

```

key: Integer           # Key pressed on input
keymap: Array          # [used internally]
  type:> Long_Integer
  count:> 4
where: Point           # Mouse position
button: Integer        # whether button was pressed

```

Response

The response type stores response information collected by an RT[] action. Variables of this type can be used with RT[] (see “RT[Label, RelativeToEvent, StorageVariable, Flag]”, p421). It is defined as follows:

```

Response:: Record
  PutUpByEvent: integer # Event that Put up RT[]
  DuringEvent: integer  # Event response occurred in
  RemovedByEvent: integer # Event that would remove RT[]
  RelativeEvent: integer # Event response is Relative to
  Label: Integer        # Not currently supported
  Input: Input          # Input information

```

13.5.1.5 Built-in Variables

The following variables are already defined when you run any experiment. They are automatically maintained by the Trial Manager.

TrialNum — This is the number of the current trial. The first trial is numbered 1.

RTData — This is an array of all of the data collected by RT[] so far. The items are of type *Response* and the size of the array is dynamically adjusted to hold all of the responses. You can pass a variable of integer type to RT[] to get the index into RTData of the last filled-in record.

13.5.2 Using Trial Manager Variables

Variable declarations and initial values are read in at the beginning of an experiment run. After the variables are read in, their values can only be changed by using the *AddToList[]*, *NewListItem[]*, *RemoveFromList[]*, or *Set[]* actions (or RT[] action for Response variables). See “14.1.1.5 Trial Variable Actions”, p422.

A variable’s value can be used from PsyScript to build each trial if the “Update” attribute is set to *True*. Otherwise, variables are used in *variable expressions* for parameters to certain actions, or as parameters to the *Start[]*, *End[]*, and *When[]* conditions.

13.5.3 Variable Expression Syntax

Variable expressions are strings of variables, values and operators that can be evaluated by the Trial Manager. Variable expressions are written as *string literals* within PsyScript; the

expressions are passed intact as action for condition parameters to the Trial Manager, where they are evaluated at run time.

For example, the `Beep[]` action is triggered in this event for the first trial only:

```
AnEvent::  
  Stimulus: "Hello World"  
  EventActions: [Start[{TrialNumber == 1}]] => [Beep[]]
```

The `Start[]` condition takes as its parameter a trial variable expression which must be True for the condition to trigger at the start of the event or trial. Notice that the expression `TrialNumber == 1` is placed within curly braces; this shelters the expression from PsyScript, so that it is not evaluated, and the string “`TrialNumber == 1`” is passed on as the parameter to `Start[]`, to be evaluated at run time.

All trial variable expressions are evaluated to numbers. For logical expressions, 0 is False, and anything else is True.

Within an expression, sub-expressions within parentheses are evaluated first, followed by constants and variable names, then operations in descending order of precedence. Operations of equal precedence are taken from left to right, unless otherwise stated.

Here are the operators in descending order of precedence:

record->*selector*: selects the field on the right from the record on the left. (Right associative)

array.*n*: returns the *n*th element of *array*.

*a***b*, *a*/*b*, *a*%*b*: returns the product, quotient, or modulo of *a* and *b*.

a+*b*, *a*-*b*: returns the sum or difference of *a* and *b*.

a=*b*, *a*==*b*, *a*>*b*, *a*<*b*, *a*>=*b*, *a*<=*b*: returns the logical value — 0 or 1 — of the order relationship of *a* and *b*.

!*a*: returns the logical negation of *a*: 0 if *a* is True, 1 if *a* is False.

a&&*b*: returns 1 if *a* and *b* are both True, 0 otherwise.

a || *b*: returns 1 if either *a* or *b* is True, 0 otherwise.



Chapter 14. Actions and Devices Reference

14.1 Actions Reference

14.1.1 Standard Actions

14.1.1.1 Trial Termination Actions

QuitTrial []

This action ends the current trial. Any trial actions which execute on the `End` [] condition will be performed.

14.1.1.2 Event Scheduling Actions

RunEvent [*Event*]

This action initiates the specified event, just as if it had been started by a regular “StartRef”-based scheduling.

If the specified event has already been run and has ended, it will be run again. If the event is currently running, then `RunEvent` [] does nothing.

EndEvent [*Event*]

This action ends the specified event, just as if its “Duration” condition had been met.

AbortEvent [*Event*]

This action aborts the specified event, clearing the stimulus, if appropriate. It also:

- deactivates any actions that are linked to the event.
- marks the time the event ended, and computes the event's duration.

ScheduleEvent [*Event*, *StartRef*]

This action schedules the specified event to be run at the specified starting time, just as if its “StartRef” attribute had been assigned the given *StartRef* value. See also `RunEvent []`.

UnscheduleEvent [*Event*]

This action removes an event from the run schedule; the event may have been scheduled with a “StartRef” attribute value or with the `ScheduleEvent []` action. If the event has already been executed, `UnscheduleEvent []` does nothing.

ChanceEvent [*Event*, *Chance*]

This action is like `RunEvent []`, except that the event is run only with the probability given in *Chance*. *Chance* can be a number between 0 and 1, or a trial variable expression that evaluates to a number.

14.1.1.3 Unscheduled Stimulus Display Actions

ShowStim [*Event*, *AttributesEvent*]

This action presents the stimulus for the specified event, without actually executing the event (i.e., its duration conditions are not watched, actions are not posted, etc.)

Usually, you will not specify *AttributesEvent*. If you specify two different events, the “Stimulus” value for the *Event* is combined with the non-“Stimulus” attributes from *AttributesEvent*.

This action also marks the time at which the stimulus began, and records this as the onset time for *Event*; it does not, however, perform the scheduling operations of `RunEvent []` and `ScheduleEvent []`.

ClearStim [*Event*]

This action clears the stimulus associated with the specified event. It also marks the time at which this occurs, and computes the actual duration for the associated event.

It does *not* perform the scheduling operations of `EndEvent []`; these will be performed when the event actually ends (if it is currently running).

MaskStim [*Event*, *Mask*, *AttributesEvent*]

This action masks the stimulus associated with the specified event.

By default, `MaskStim []` uses the mask stimulus specified as the “Mask” attribute for the event; however, an optional a mask stimulus (for `Text` stimuli, this is a character) can be specified as the second argument.

An optional *AttributesEvent* can be specified for non-“Stimulus” attributes, just as with *ShowEvent* [].

14.1.1.4 Miscellaneous Actions

RT[*Label*, *RelativeToEvent*, *StorageVariable*, *Flag*]

This action records the state of all input devices and stores this either: a) in the data file, or b) in a specified variable of type *Response*, or c) in both the data file and a variable. Where the information is stored depends on the values of the *StorageVariable* and *Flag* parameters; the default is to the data file.

The *Label* string is stored along with the regular response time information; this label is used by the experiment designer to annotate the recorded data and is meaningless to PsyScope. The default label is “”.

The *RelativeToEvent* parameter changes which event is used to calculate response time; the recorded response time will be the difference between start time of this event and the time at which a response was received. The default *RelativeToEvent* is the one owning the action.

The *StorageVariable* optional parameter specifies a variable with either a numerical or *Response* type. If the variable’s type is *Response*, the response data is copied into this variable. If the variable’s type is numerical (*Integer*, *Long_Integer*, etc.), the variable is filled in with an index into the standard response list — *RTData* — when the response was recorded.

The *Flag* optional parameter is used when a variable of type *Response* is specified for *StorageVariable*; its only non-default value is *VAR_ONLY*, which indicates that the response information should not be written to the data file (only to the variable).

BBoxOut[*Value*, *Mode*]

This is an action which sends bbox output codes directly to the box, without the need for a *BBox* event. The syntax is:

Value an integer value in [0..255] representing the decimal value of the binary lines to be manipulated

Mode (optional: defaults to *copy_mode*)

"*copy_mode*" - change the bbox output state to *Value*.

"*assert_mode*" - turn on the bits that are on in *Value*.

"*deassert_mode*" - turn off the bits that are on in *Value*.

"*xor_mode*" - exclusive-or the current state with *Value*.

Beep [*SND_ResourceName*]

This action plays the sound stored in the specified ‘snd ’ resource. If no sound name is specified, "correct beep" is used.

The resource must be in an open resource file – either one that was placed in the “PsyScope Extensions” folder, or that was listed in the “Resources” attribute of the experiment.

CancelAction [*Action, Event, Condition*]

This action removes either a specified action or all actions (controlled by *Action*), from the action list of either a specified event or all events (controlled by *Event*), and triggered by either a particular condition or any condition (controlled by *Condition*).

If an action name is specified in *Action*, only actions with that name will be removed.

If an event name is given in *Event*, only actions within that event will be removed.

If a condition name is given in *Condition*, only actions that depend on the given condition will be removed.

Note: CancelAction[] can only remove actions that have not yet been executed.

ScriptEval [*entry*]

This action is used to directly evaluate the entry named *entry*. Usually, evaluating the entry will change some value in the script, and that value will be used for compiling future trials.

14.1.1.5 Trial Variable Actions

Set [*List, Value*]

Given a trial variable expression that resolves to a variable reference in *List*, the variable’s value is set to the result of evaluating *Value*. (See also “13.5 Trial Manager Variables”, p412.)

AddToList [*List, Value*]

This action is used to add a value to a trial variable of *List* type. *List* should be a variable expression that evaluates to a list variable, and the result of evaluating *Value* is appended to this list. (See also “13.5 Trial Manager Variables”, p412.)

RemoveFromList[*List*, *Index*]

Given a trial variable of type `List` in *List* and an index into this list in *Index*, the indexed item is removed from the list. The list is indexed starting with 1. (See also “13.5 Trial Manager Variables”, p412.)

NewListItem[*List*]

This action extends the size of the trial variable list given in *List*; the value of the new item in the list is undefined. (See also “13.5 Trial Manager Variables”, p412.)

14.1.1.6 Factor Format Actions**QuitBlock**[*Block*, *AccessLists*]

This action is used to skip any remaining trials in the current block. The current trial continues to execute normally.

If there are multiple levels of blocks in the experiment hierarchy, then `QuitBlock[]` quits within the lowest-level block by default, continuing within that block’s owner (if there are more blocks to execute). To quit a higher-level block, you can specify which block to quit in *Block*. Alternatively, you can specify which block to quit as a number; this number specifies how many hierarchical levels of blocks to quit (thus, the default behavior is equivalent to specifying 1 in *Block*).

By default, when trials in the block are skipped, any lists connected to the blocks are left unaccessed for the trials which are not executed. If *AccessLists* is set to `True`, then for each factor set connected to the block and its owners, a cell is selected for each trial that is not executed; this insures that cells are assigned to trials consistently, whether or not they are run.

`QuitBlock[]` does not terminate the current trial; see also “QuitTrial[]”, p419.

RerunTrial[*TrialNumber*, *When*, *Order*]

This action tags the specified trial, or the current trial if none is specified, to be run again. A trial is specified using its absolute trial number; this specification can be in the form of a variable expression.

The *When* parameter specifies when the trial should be re-run; the possible values are `Mix` and `End`. `Mix` specifies that the trial re-run should be mixed in with the remaining first-run trials, while `End` specifies that the re-run should be delayed until all of the first-runs are done. If a trial is re-re-run with `End`, the second-time re-runs will be performed after the first-time re-runs are complete.

The *Order* parameter specifies an order within the two *When* types; the possible values are `Start`, `End`, and `Random`. `Start` specifies that the trial should be re-run before any other trials currently scheduled for re-run in its set (`Mix` or `End`). `End` specifies that it should be re-run after the other re-run trials. `Random` specifies that it should be rescheduled at a random position within its re-run set.

NextCrossing [*Factor*]

This action causes a new cell to be selected in a factor set — whichever set includes *Factor*. The new cell will be used for the next trial.

This action is intended for use on factor sets with crossing type *Static* (see “13.3.7.2 Scripting Crossing Types”, p389).

14.1.2 Type-specific Actions

Various stimulus types include type-specific actions. These actions can generally be executed by any event, but may sometimes require as a parameter the name of an event with a particular type. The type-specific actions are listed with each type in “14.2 Stimulus Types Reference”, p424.

14.2 Stimulus Types Reference

14.2.1 Text

14.2.1.1 Text and Screen Attributes

Stimulus: *stimulus-string*
Defaults: *none*

The “Stimulus” attribute should contain a single sting to display.

Port: LEFT/RIGHT/CENTER/*x-value width value*
 TOP/BOTTOM/CENTER/*y-value height value border_width*
AlignmentPoint:> *h-pos-spot v-pos-spot*
BorderWidth:> *border-width*
Shape:> *shape*
Defaults: CENTER 100% CENTER 100% 0
 AlignmentPoint:> (Depends on port position)
 BorderWidth:> 1
 Shape:> RECTANGULAR

This attribute describes the position and size of the port in which the stimulus is to be presented. All of the sub-attributes and parameters have defaults, but you cannot skip a parameter if you which to define any that follow it.

The first and fourth tokens specify the position of the port relative to the screen. They can be:

A keyword: LEFT, RIGHT, or CENTER for the x-position, and TOP, BOTTOM, or CENTER for the y-position; the (hot spot of the) port is justified in the specified way to the screen; e.g., CENTER centers the box on the screen

(horizontally or vertically), and `TOP` aligns the box so that the top end is against the top of the screen.

An integer; this specifies a number of pixels by which the (hot spot in the) port should be offset from the top left of the screen.

A percentage; this specifies that the port should be offset to a point some percentage of the screen away from the top and left edges of the screen.

Each of the second and fifth tokens — the *width* and *height* values — can be:

An integer; this specifies the width or height of the port in pixels.

A percentage; this specifies that the width or height of the port should be a certain percentage of the size of the screen.

The seventh token — *border_width* — specifies the width of the port’s border in pixels.

The “AlignmentPoint” sub-attribute specifies the position of the hot spot, relative to the port. It is the hot spot that is actually positioned by the x-position and y-position information, and then the port is positioned to obtain the right relationship with the hot spot. If the hot spot is not specified, the default is `Center` for percentage- or pixel-based positionings, and the same as the port positioning for other alignments (e.g. `Right h-pos-spot` for `Right x-value`).

This “Shape” sub-attribute specifies the shape of the port. The possible shapes are `RECTANGULAR`, `ROUNDED`, and `OVAL`. `Rectangular` produces a rectangular port whose dimensions and positions are specified as shown above. `Rounded` produces a rounded-corner rectangle with the same dimensions. `Oval` will produce an oval inscribed in the rectangle of the given dimensions.

Note: All drawing within a port is clipped to remain within the borders of the shape of the port.

X:	<code>LEFT/RIGHT/CENTER/x-value</code>	<code>SCREEN/PORT</code>	<code>RIGHT/LEFT/CENTER</code>
Defaults:	<code>CENTER</code>	<code>PORT</code>	<code>CENTER</code>
Graphic:	<i>not available</i>		

Note: The graphic environment uses `Position` instead of `X`.

This attribute specifies the horizontal position and alignment of the text on the screen. All of the parameters have defaults, but you cannot skip a parameter if you wish to define any that follow it. “X” attributes override values in the “Position” attribute.

The first value specifies the position: left margin, right margin, centered, an integer value for the position in pixels, or a percentage value.

The second value specifies whether the position (and possibly percentage) is relative to the screen or the port. If the point is relative to the port, a percentage in the position specification is taken as percentages of the port width; if it is relative to the screen, the width of the screen is used.

The third value specifies the justification of the stimulus on the point specified by the position value (e.g. RIGHT means that the text is drawn to the right of the position).

If either FOLLOW or STAY_PUT is used as a value of the “Feature” attribute (see below), then the position of the stimulus is computed relative to the prior stimulus. In either of these cases, the second (relative-to) value of the “X” attribute is ignored, and the first (position value) is used as the horizontal offset (in pixels) between the prior stimulus and the current one.

Y:	TOP/BOTTOM/CENTER/ <i>y-value</i>	SCREEN/PORT	TOP/BOTTOM/CENTER
Defaults:	CENTER	PORT	CENTER
Graphic:	<i>not available</i>		

Note: The graphic environment uses Position instead of Y.

This attribute specifies the vertical position and alignment of the text on the screen. All of the parameters have defaults, but you cannot skip a parameter if you which to define any that follow it. “Y” attributes override values in the “Position” attribute.

The first value specifies the position: top margin, bottom margin, centered, an integer value for the position in pixels, or a percentage value.

The second value specifies whether the position (and possibly percentage) is relative to the screen or the port. If the point is relative to the port, a percentage in the position specification is taken as percentages of the port height; if it is relative to the screen, the height of the screen is used.

The third value specifies the justification of the stimulus on the point specified by the position value (e.g. TOP means that the text is drawn just above the position).

If either FOLLOW or STAY_PUT is used as a value of the “Feature” attribute (see below), then the position of the stimulus is computed relative to the prior stimulus. In either of these cases, the second (relative-to) value of the “Y” attribute is ignored, and the first (position value) is used as the horizontal offset (in pixels) between the prior stimulus and the current one.

Position: LEFT/RIGHT/CENTER/*x-value* SCREEN/PORT RIGHT/LEFT/CENTER
 TOP/BOTTOM/CENTER/*y-value* SCREEN/PORT TOP/BOTTOM/CENTER
 Defaults: CENTER PORT CENTER CENTER PORT CENTER

This attribute is the concatenating of “X” and “Y” formats. If “X” and/or “Y” is present, its values override “Position” values.

Font: *name*
 Default: 0 (*Chicago* on most systems)

This attribute specifies the font name or 'FOND' resource number (which must be in Macintosh System file or in one of the resources opened by PsyScope; see also “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216).

Size: *Size*
 Defaults: 12

The “Size” attribute is the point size of the stimulus text.

Face: Bold *and/or* Italic *and/or* Underline *and/or* Outline
and/or Shadow *and/or* Extended *and/or* Condensed
 Defaults: NULL (Plain text)

Any and all of these values can be used; combine text faces as you wish, but the face specifications must all be together in one string, i.e., the list of faces should be quoted together.

Mode: COPY/OR/XOR/ERASE/INVCOPY/INVOR/INXOR/INVERASE
 Default: COPY

This attribute is the QuickDraw transfer mode. This determines how new stimuli interact with the background and other stimuli already present in the port at the destination location.

COPY simply replaces anything in the destination location with the text stimulus, writing over that location without regard for what was already there.

OR draws the text without affecting pixels except where the pixels for a letter is placed, thus “overlying” the destination with the text.

XOR draws the text without affecting pixels except where the pixels for a letter is placed; pixels here which are off will be turned on, and pixels which are on are turned off.

ERASE inverts the text before it draws it. In black-and-white, it essentially writes in white instead of black.

INVCOPY, INVOR, and INXOR perform similar operations, but invert the text first. These are not recommended modes.

(You can also use the Macintosh toolbox numerical value, if you really want to.)

Color: RED/GREEN/BLUE/YELLOW/MAGENTA/CYAN/WHITE/*rgb-string*
Default: BLACK

This attribute specifies the color of the text. The color value can be specified using a keyword or the “Red Green Blue” system.

For the RGB system, three integers are specified, each in the range 0 to 32677, and representing the amount of red, green and blue to be used, respectively; these three number should be together in one string.

If no color attribute is given, the stimulus will be drawn in the default foreground color for the screen device. This color is initially black, but can be changed by the Experiment attribute “DefaultColor”, or by the screen functions `SetDefaultColor` and `ReverseVideo` (see “`SetDefaultColor[Color]`”, p431 and “`ReverseVideo[]`”, p431).

Style: “*Font*” + “*Size*” + “*Face*” + “*Mode*” + “*Color*”

This attribute may be used to set many attributes of the text stimulus as described above all at once; its format is the concatenation of the formats of the attributes listed above. If present, the individual attributes override the “Style” attribute.

Mask: *mask-char*
Defaults: “ ”

This attribute contains a character used to mask the stimulus when `MaskStim[]` is executed from the script, or the “ClearType” event attribute (see “13.1.6.3 Standard Event Attributes”, p367) is set to `MASK`.

Feature: FOLLOW *and/or* STAY_PUT *and/or* MASKED
Defaults: NULL
Graphic: **Special**

This attribute contains a list of special stimulus features. The values of this attribute are specified as keywords, which are described below.

`FOLLOW` causes the stimulus to be positioned after the end of the prior stimulus. The first (position) value of the `X` specifies how much space (in pixels) should be left between the end of the prior stimulus and the beginning of the current one. If the positioning of the stimulus will cause it to extend beyond the right margin of the port, then it is begun at the left edge of the next “line” (i.e., down an amount determined by the `SIZE` attribute). That is, lines of stimuli that use the `FOLLOW` feature are “wrapped” within the port.

`STAY_PUT` causes the stimulus to be positioned at the same place as the prior stimulus, offset horizontally by an amount (in pixels) specified in the first (position) value of the `X` attribute, and vertically by an amount (in pixels) specified in the first (position) value of the `Y` attribute.

MASKED causes the stimulus to be drawn in its masked form when it is originally displayed. If there is no mask character specified for the stimulus, it will not be drawn.

Degradation: (2 decimal values, each between 0.0 and 1.0)
 Defaults: 0.0 0.0

This attribute is used to delete pixels from the text display and/or add pixels to the whitespace around the text (presumably to make it harder to recognize). The first value specifies the probability with which each pixel of the stimulus is deleted, and the second value specifies a corresponding probability for adding pixels to the background.

Flip: HORIZONTAL *and/or* VERTICAL
 Default: NULL

This attribute specifies whether to flip the bitmap horizontally and/or vertically before displaying.

14.2.1.2 Text and Screen Experiment Attributes

These attributes apply to all screen stimulus types, including Text, Document, Paragraph, PICT, and Pasteboard.

DefaultColor or **ForeColor:** *color*
 Default: BLACK

This experiment attribute is used to determine the default (foreground) color of screen stimuli. The color is specified in the same way as in the color Text attribute.

BackColor: *color*
 Default: BLACK

This experiment attribute specifies the background color of the screen when running an experiment. The color is specified in the same way as in the color Text attribute.

Origin: MenuBar/UpperLeft/Top/Left/*co-ordinates*
 Default: UpperLeft

This attribute specifies the point on to be used as (0,0) for positioning stimulus ports and stimuli. Most useful on systems with multiple screens. Possible values are:

MenuBar — the upper left corner of the screen with the menu-bar

UpperLeft — the upper left corner of the smallest rectangle enclosing all the screens

Top — horizontally as with MenuBar, vertically as with UpperLeft

Left — horizontally as with `UpperLeft`, vertically as with `MenuBar`

Horizontal and vertical coordinates may also be specified, in pixels down and to the right from the top left corner of the menu bar screen.

MonitorOrder: `Random/Sequential/Rotating/sequence`
Default: `Random`

This attribute specifies the order in which a stimulus spanning multiple monitors will have its parts drawn to the different monitors. The monitors are numbered the same way as they are in the system software’s “Monitors” control panel. Possible values are:

`Random` — drawn in random order

`Sequential` — drawn in increasing order of monitor number

`Rotating` — drawn in order of monitor number, but with the starting point increasing each time

A sequence of monitor numbers may also be given to define your own order.

14.2.1.3 Text and Screen Actions

ClearScreen []

This action erases the screen, using the global background color. (See also “Back-Color: color”, p429.)

DrawAllPortBorders []

This action draws the borders of all stimulus ports that are used by screen events in the experiment.

If different trials use different stimulus ports, then this action can only draw ports that it knows are going to be used. If the experiment is precompiled (see “6.5.1 Pre-Compiling”, p246), then all ports will always be known at runtime. Otherwise, the only ports that can be drawn are those used by events which have been compiled so far.

DrawPortBorder [*Event*]

This action draws the border for the stimulus port of the specified event.

RemovePortBorder [*Event*]

This action removes the border for the stimulus port of the specified event.

ClearPort [*Event*]

This action clears the contents of the stimulus port for the specified event. It does not erase the border of the port.

SetDefaultColor [*Color*]

This function changes the default drawing color. This value can also be set once with the experiment attribute “DefaultColor” or “ForeColor” (see “DefaultColor or ForeColor: color”, p429).

SetBackColor [*Color*]

This function changed the default background color. This value can also be set once with the experiment attribute “BackColor” (see “BackColor: color”, p429).

ReverseVideo []

This function switches the default foreground and background colors. See also “DefaultColor or ForeColor: color”, p429 and “BackColor: color”, p429. If no colors have been set, then text stimuli will appear white against a black background, as opposed to the Macintosh’s usual black-against-white.

14.2.2 Document

14.2.2.1 Formatting characters

You can create `Document` files by using the PsyScope text editor, or any editor (such as MacWrite or Microsoft Word) and specifying that the file be saved as “Text Only”. Make sure that the files do not contain any tabs.

You can specify different text faces within a text document with a standard set of text formatting characters: `@b`, `@i`, `@u`, `@e`. The `@b` character is placed at the beginning of any string of text that you want to display in **bold**. The `@i` character is placed at the beginning of any string of text that you want to display in *italics*. The `@u` character is placed at the beginning of any string of text that you want to display in underline. The `@e` character is used to turn off any of these faces.

14.2.2.2 Document Attributes

Stimulus: *document*
Defaults: *none*
Graphic: **File**

The “Stimulus” attribute should contain the name of the file to display. See “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215 for information on path name conventions.

Font, Size, Face, Mode, Color, and Style

These attributes work like the standard `Text` attributes. They apply to all of the text; the formatting characters are applied afterwards.

Port

This attribute work like the standard `Text` attribute.

14.2.3 Paragraph

The `Paragraph` stimulus type is identical to the `Document` stimulus type, except that it gets its text from the script itself instead of from a file specified in the script.

14.2.3.1 Paragraph Attributes

Stimulus: `paragraph-text`

Defaults: `none`

Graphic: **Paragraph**

The “Stimulus” attribute contains the text to display. This text can contain linefeeds and formatting characters, just the text of a `Document` stimulus.

Font, Size, Face, Mode, Color, and Style

These attributes work like the standard `Text` attributes. They apply to all of the text; the formatting characters are applied afterwards.

Port

This attribute work like the standard `Text` attribute.

14.2.4 KeySequence

A `KeySequence` event is a `Paragraph`-like event that accepts keyboard input from the subject until the event ends.

Note: `KeySequence` is not an input type; the input string is recorded in the data file by using the `KEY_SEQUENCE` flag in the “DataFields” experiment attribute and recording an `RT[]` at the end of the event.

14.2.4.1 KeySequence Attributes

Stimulus: *paragraph-text*
Defaults: *none*
Graphic: **Prompt**

The “Stimulus” attribute contains the text to display as a prompt. This text can contain linefeeds and formatting characters, just the text of a Document stimulus.

Font, Size, Face, Mode, Color, and Style

These attributes work like the standard Text attributes. They apply to all of the text, including the subject’s input; the formatting characters are applied afterwards.

Port

This attribute work like the standard Text attribute.

14.2.5 PICT

14.2.5.1 PICT Attributes

Stimulus: *pict-resource-name-or-file-name*
Defaults: *none*
Graphic: **Picture**

The “Stimulus” attribute should contain either the name of a ‘PICT’ resource, or the pathname of a PICT file to display (see “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215 for information on path name conventions). The search starts by looking for the resource.

Port, Mode, and Degradation

These attributes are specified in exactly the same manner as for Text.

Feature: KEEP_PICT & DEFAULT_COLORS & PICT_ACTUAL_SIZE
Defaults: NULL
Graphic: **Special**

This attribute contains a list of special stimulus features. The values of this attribute are specified as keywords, which are described below.

KEEP_PICT specifies that the picture should stay in memory after it has been loaded from the disk. This may use a lot of memory, but it can greatly reduce the delay between trials. (See “Part 2: Graphic Environment Reference, Keeping Stimuli in Memory”, p251.)

DEFAULT_COLORS: If the default drawing colors have been changed, colors PICTs may need this flag to display in reasonable colors (by ignoring the foreground and background color settings).

PICT_ACTUAL_SIZE: If this flag is on, the PICT will be drawn in its native size, centered in its port. PICTs are usually scaled to fit in their ports.

Depth: 1/2/4/8/32
Defaults: *current screen depth*

This specifies the pixel depth (the number of bits used for one pixel) of the offscreen buffer used to hold the picture in memory. It can be any accepted Macintosh screen depth that is less than or equal to the current depth of the screen. If the picture is monochromatic, it is advisable to use a depth of 1 for reasons of space conservation. (See “Part 2: Graphic Environment Reference, Loading PICTs”, p248).

14.2.6 Pasteboard

14.2.6.1 Pasteboard Attributes

All pasteboard attribute names have the prefix “PBoard” to distinguish them from the attributes of the stimuli that are contained within the pasteboard. (Without this distinction, attribute inheritance would cause problems.)

Stimuli: *stimuli-list*
Defaults: *none*

The “Stimuli” attribute should contain the names of the entries defining its constituent stimuli. There is no restriction of the number of stimuli that can be in a PasteBoard.

The stimuli of a PasteBoard will look just like events, except that they should have a “StimType” attribute in place of an “EventType” attribute.

PBoardMode, PBoardDepth, and PBoardDegradation

These attributes are used in exactly the same manner as the “Mode”, “Depth”, and “Degradation” attributes of Text and PICT stimuli, and have the same meaning.

PBoardPort

This attribute specifies the stimulus port in which the pict is to be displayed, specified in the same fashion as the Port attribute for other screen stimuli. Each of the sub-stimuli of a pasteboard also have ports (specified relative to the screen), and their port defaults to the pasteboard port.

Aside from the default, sub-stimuli port are defined independently of the Pasteboard port; their drawing will be clipped to the region in which their ports intersect the pasteboard port.

14.2.6.2 Pasteboard Experiment Attributes

PBoardNameDelimiter: *separator*
Default: "+"

This experiment attribute specifies a character that will be used to separate the names of the different stimuli in the stimulus data field of the data file.

14.2.7 SoundLabel

The SoundLabel type is for playing SoundEdit™ and SoundDesigner II™ sound files.

14.2.7.1 SoundLabel Attributes

Stimulus: *label*
Defaults: NULL
Graphic: **Sound**

The “Stimulus” attribute should contain a label string to be found and played in the SoundEdit™ file. NULL specifies that the whole file should be played.

SoundFile: *file-name*
Defaults: *none*
Graphic: **File**

This is the file from which the labelled sound is to be taken. If there is no such file or the file does not contain the specified label, an error will result. (See “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215 for more information on path name conventions.)

Volume: (integer value between 0 and 255)
Defaults: 255

This specifies the volume of the sound, where 0 indicates silence and 255 indicates full volume. This attribute is only available on Macintoshes running system version 6.0.5 or later.

For systems 6.0.7 and later, the *absolute maximum* volume is the volume set in the “Sound” Control Panel.

Channel: LEFT/RIGHT
Defaults: NONE

This specifies the stereo channel the sound should be played on. It is ignored for two-channel sounds. It is only available using Macintosh system software 6.0.7 or later.

Warning! Use of the channel attribute may cause some delay in the start of the sound because of higher Sound Manager overhead, and some loss of volume or fidelity in the sound may also result.

Feature: KEEP_SOUND
Defaults: NULL

The “Feature” attribute contains a list of special stimulus features. The values of this attribute are specified as keywords, which are described below.

KEEP_SOUND specifies that the sound be kept in memory across trials and only unloaded at the end of the experiment. This is useful if the sound is used in every trial, and the scripter would like to prevent it from being reloaded (causing a disk-access) every trial. It means that the memory used by the sound won’t be released until the experiment has ended, however. (See “Part 2: Graphic Environment Reference, Keeping Stimuli in Memory”, p251.)

14.2.8 BBox

The BBox type is for controlling the output lights on the button box.

14.2.8.1 .BBox Attributes

Stimulus: *line-switches*
Defaults: NULL

The “Stimulus” attribute specifies which line-outs (or lights) on the button box to change and how to change them. Lines which are not mentioned are not affected. At the end of the event, the lines are changed to the opposite of the “Stimulus” specification, regardless of their initial state.

The format of each the *line-switches* is one of:

LINE*X* or LINE*X*_ON, where *X* is an number from 1 to 8; this turns on the line output at the beginning of the event, and then off at the end. If *x* is between 1 and 3, a light on the button box is also turned on and off.

LINE*X*_OFF, where *X* is an number from 1 to 8; this turns off the line output at the beginning of the event, and then on at the end. If *x* is between 1 and

3, a light on the button box is also turned off and on.

LIGHTX or LIGHTX_ON, where X is an number from 1 to 3; this is the same as LINEX_ON.

LIGHTX_OFF, where X is an number from 1 to 3; this is the same as LINEX_OFF.

14.2.8.2 BBox Experiment Attributes

BBoxInitialState: *line-switches*
Default: NULL

This attribute specifies and initial output state for the button box using the same list of keywords used for the BBox “Stimulus” attribute. Unlike BBox events, the initial state turns off all lines that are not specified in the “BBoxInitialState” attribute.

BBoxPort: A or B
Default: A

This attribute specifies which serial port the button box is connected to. Port A is the modem port; B is the printer port.

14.3 Conditions and Inputs

The following keywords can be used to specify conditions required to trigger an action. Except for START and END, they can also be used in duration specifications.

Start [*variable-expression*]

This condition matches the start of the event or trial that owns it. If an optional variable expression is given, the condition will match only if the expression is true.

End [*variable-expression*]

This condition matches the end of the event or trial that owns it. If an optional variable expression is given, the condition will match only if the expression is true.

When [*variable-expression*]

This condition matches when the given variable expression becomes true.

ScriptWhen [*entry-name*]

This condition matches when the value of the referenced entry is True. Only one entry name can be specified.

Key [ANY/key-list]

Given the ANY parameter, this condition matches when any non-modifier key is hit on the standard Macintosh keyboard.

Otherwise, a list of key combinations is specified; the condition will match when any one of the specified combinations is hit.

A key combination is made up of any number of modifiers keys and a base (non-modifier) key. The base key can be specified as:

- the letter, number, or symbol that appears on the key on the keyboard. If a letter is specified, the case (uppercase/lowercase) of the letter is ignored. If a symbol is given that requires the shift key to be typed, the shift key will be required for a match (but the SHIFT- prefix should *not* be used – see below).
- the ASCII- keyword followed by a number. The condition will match when the correct combination of keys is pressed to type the character with the given ASCII value. Only the shift key may be used to generate the ASCII character; characters generated by using the option key can not be matched this way.
- the CODE- keyword followed by a number which corresponds to a unique key on the keyboard; key code numbers are defined in Macintosh programming documentation. This method of specifying a key is somewhat keyboard-specific, but allows you to use all keys on the keyboard. Modifier keys cannot be matched this way.
- the SPACE keyword, representing the space bar.
- the RETURN keyword, representing the return key.

To specify a key combination containing modifiers, prefixes are added to one of the base forms described above. The CMD- prefix matches only when the command key is held down at the same time as the base key. Similarly, CTL- matches with the control key and OPT- matches with the option key. SHIFT- matches with the shift key only when it is not already represented in the base character (e.g.: SHIFT-* can never match, since * – which is the same as SHIFT-8 – already contains SHIFT).

Note: Uppercase letters are translated to lowercase letters in the matching process; thus, A matches the “a” key without shift, while SHIFT-A matches with the shift key.

Note: Be sure to quote a modified, ASCII-, or CODE- combination, since “-” is a scripting operator.

Mouse [CLICK/CLICK_DOWN/CLICK_UP/MOVE]

This condition matches when a mouse action is made; the relevant action depends on the parameter:

CLICK, CLICK_DOWN — matches when the mouse button is pressed

CLICK_UP — matches when the mouse button is released

MOVE — matches when the mouse is moved

BBox [*button-list*]

With the ANY parameter, this condition matches when any button is hit (or any input line is activated, including the voice key) on the C.M.U. button box; with the ANY_UP parameter, it matches when any button is released (or any input line is deactivated, or voice input ends). The ANY_BOTH parameter can be used for ANY and ANY_UP combined.

Otherwise, a list of button/line states are specified, and the condition will match when one of the states occurs. States can have the following forms:

BUTTONX, BUTTONX_DOWN, BUTTONX_UP, or BUTTONX_BOTH where X is 1 (red), 2 (yellow), or 3 (green); these refer to the three regular buttons on the top of the button box.

VOICE, VOICE_END, VOICE_BOTH; these refer to the standard microphone input on the button box.

LINEX, LINEX_ON, LINEX_OFF, or LINEX_BOTH where X is between 1 and 8; LINES 1-3 are the same as BUTTONS 1-3, and LINE7 is the same as VOICE. The other lines refer to hardware inputs on the back of the button box.

The BUTTONX, BUTTONX_DOWN, VOICE, LINEX, and LINEX_ON parameters match when the corresponding inputs are turned “on” (a button is pressed, a sound is detected, or an input line is activated). BUTTONX_UP, VOICE_END, and LINEX_OFF match when the corresponding inputs are turned “off” (a button is released, no more sound is detected, or an input line is deactivated). BUTTONX_BOTH, VOICE_BOTH, and LINEX_BOTH match both “on” and “off” states.



Chapter 15. Trial Manager Technical Reference

A PsyScope *trial* is simply a set of events scheduled to run in some known temporal or conditional relationship with one another, and a set of actions associated with these events set to run dependent on certain conditions:

To prepare for running the events, stimuli must be loaded into memory by various output devices.

When event are run, stimuli must be displayed by the output devices.

As a result of running the events and executing the actions, data are collected and must be stored.

Throughout the trial, input devices must be polled to determine whether the necessary conditions to execute an action have been satisfied.

The *Trial Manager* is the part of PsyScope responsible for managing these functions, as well as managing the functions of the various input and output devices.

Some topic which are appropriate to this chapter have been covered elsewhere in this manual. Where that is the case the appropriate chapter and section have been cross-referenced. For more general information regarding running experiments, see “Part 2: Graphic Environment Reference, 6.5 Space and Speed”, p246.

15.1 Running Trials

15.1.1 Loading Stimuli

Loading Stimuli is discussed in “Part 2: Graphic Environment Reference, 6.5.2 Loading Stimuli”, p247.

15.1.2 Running Events and Actions

15.1.2.1 The START Event

The first event in every trial is the *START event*. This event is automatically defined by PsyScope to have the event name `START`, and event number 0. All trial preparatory activities are done during the start event.

The `START` event begins at the end of the previous trial (for the first trial, it begins when the experiment is initialized), and lasts until all trial preparation is completed, or until the number of milliseconds specified in the “ITI” trial attribute has passed, whichever is *longer*. See “Part 2: Graphic Environment Reference, 6.5.2.2 Controlling The Load Procedure”, p249.

During the `START` event the following things are done:

- Any data that were collected during the trial are stored to disk, unless the `STORE_DATA_AT_END` flag has been set.
- The upcoming trial is compiled by the script interpreter, unless the trials were pre-compiled.
- Internal memory structures needed to run the trial are set up.
- All stimuli needed for the trial are loaded, except those explicitly set to be loaded later. See “Part 2: Graphic Environment Reference, Load Time”, p180.
- The event dependencies are analyzed.
- Trial actions are posted, and those with `START[]` conditions are triggered.

Because of the special nature of the `START` event, no other events may run concurrently with it, and thus, no events may be scheduled relative to the start of the `START` event. In other words, no event may have a “StartRef” of the form *milliseconds after start of START*. The proper form for events schedule relative to the start of the trial is *milliseconds after end of START* (or even *milliseconds after START*), which will schedule the event to begin the given number of milliseconds after the end of the `START` event. Thus the *end* of the start event, is the time at which all other events may start running.

15.1.2.2 The Life of an Event

Every event is started by a `RunEvent[]` action, and ended by an `EndEvent[]` action. For the vast majority of events, these actions are implicit—scheduled automatically by the Trial Manager after the event dependencies of the trial have been analyzed

Starting an Event

When `RunEvent[]` starts an event, it does the following things, in this order:

- If the event’s “LoadTime” attribute is set, the stimulus is loaded. (See “Part 2: Graphic Environment Reference, Load Time”, p180.)
- Put all of the event’s actions on the action queue, and trigger any actions based on `Start[]` conditions (which have no parameters or a trial variable expression parameter whose value is True).
- Call `ShowStim[]` to display the stimulus and mark the actual onset time for the event.
- For events with timed durations, schedule the `EndEvent[]` action that will end the event.
- Schedule the start of any other events which depend on the start of the current event, i.e., any events with “StartRef” values of the form *milliseconds* after start of event.

The Life of the Stimulus

Generally, running an event is linked with displaying a stimulus. In most cases, the stimulus lasts for the duration of the event. It is possible, however, for the stimulus to be cleared before or after the end of the event.

The `ClearStim[]` action will clear the stimulus of an event. If it is called before the end of the event, it will clear the stimulus, and mark the end time of the event for calculation of the actual duration. The event itself will not end until an `EndEvent[]` action for it is executed — as defined in the “Duration” attribute of the event.

If the event has a “ClearType” attribute of `NO_CLEAR`, the stimulus will not be cleared at the end of the event, and will continue to be displayed indefinitely, until `ClearStim[]` is called for that event. In this case, the actual end time of the event that will be recorded is the time at which the event was ended, not the time at which the stimulus was cleared. See below (“15.1.2.4 Event Statistics”, p444) for more information.

Ending an Event

When `EndEvent[]` is called to end an event, it performs the following functions:

- If the stimulus has not already been cleared, and the “ClearType” of the event is not `NO_CLEAR`, call `ClearStim[]` to clear the stimulus and mark the end time of the event. Otherwise, if the “ClearType” of the event is `NO_CLEAR`, just mark the end time of the event.
- Remove any actions on the action queue that are active until the end of this event, and trigger any actions based on `End[]` conditions (which have no parameters or a trial variable expression parameter whose value is True)
- Calculate the actual duration of the event from the actual onset and the end time.
- If the “LoadTime” attribute of the event is set, unload the stimulus for the event.

- Schedule the start of any other events which are dependent on the end of the current event, i.e., any events with “StartRef” values of the form *milliseconds after end of event*.

15.1.2.3 Running an Event More than Once Per Trial

It is possible to run an event more than once in a single trial. This can be done through the event schedule, or the `RunEvent[]` or `ScheduleEvent[]` actions. The only restriction is that only one “instance” of an event may be running at any one time.

Frequently it is useful to have an event re-schedule itself. There are two ways to do this. The first is to give the event a “StartRef” that references itself. For example, an event called `EventA` could have a “StartRef” `100 after end of EventA`. This would cause `EventA` to run again 100ms after it ends. In order for this event to run the first time, it would have to be run by a `RunEvent[]` or `ScheduleEvent[]` action attached to another event.

Another way for an event to re-schedule itself would be to use the `ScheduleEvent[]` action. For example, `EventA` it could have `ScheduleEvent[EventA]` as an action with the condition `END[]`. This would cause `EventA` to continually run again as soon as it has ended.

ScheduleEvent[] vs. RunEvent[]

It is important, in the latter example above that `ScheduleEvent[]` and not `RunEvent[]` be used. Unlike `RunEvent[]`, which immediately runs an event, `ScheduleEvent[]` simply puts the event on the schedule to be run. In the latter example above, this allows `EventA` to end before the trial manager attempts to run it again. Had `RunEvent[]` been used instead of `ScheduleEvent[]` nothing would have happened, because at the time `RunEvent[]` would have been called, `EventA` was already running, and the Trial Manager would not have attempted to start it running again.

15.1.2.4 Event Statistics

Each event stores its *actual onset* and *actual duration*. The **actual onset** of an event is the actual time at which the stimulus was displayed, which could be different from the time that it was scheduled to be displayed because of overhead or latency within the output device (e.g. screen refresh latency) or other factors. Similarly, the **actual duration** is the actual number of milliseconds from the time the stimulus was displayed until the time it was cleared, or the event ended, whichever came first.

Each event stores only *one* actual onset and actual duration, so if an event is run more than once, it will only store the *last* recorded onset and duration.

15.1.2.5 The Life of an Action

The Trial Manager maintains an *action queue*, in which it keeps track of all active actions. When an event is run, it puts all of its actions on the action queue, where it is monitored by the Trial Manager, and eventually executed or removed.

Each action on the action queue has a set of conditions, which may become true and cause it to be executed, a number of instances, which determines the maximum number of times it can be executed, and an active-until value, which determines when the action can no longer be executed, no matter how many instances are remaining.

When an action is put on the queue, its conditions are immediately checked. If it has a `Start[]` condition (with no parameters or a trial variable expression parameter whose value is `True`), the action is immediately executed and its instances value is decremented by 1. After the event that put up the action has started, it is repeatedly checked to see if any of its conditions have become true. If so, it is executed and its instances value is decremented by 1.

When an action's instances reach 0, or the event that it was specified to be active until ends, it is removed from the queue. Actions that are set active until `TRIAL_END` will remain on the queue until all their instances are exhausted, or until the trial ends.

15.1.2.6 Ending a Trial

A trial ends when there are no events currently scheduled to run or end, and the only actions remaining on the action queue are set to be active until `TRIAL_END`. A trial can be ended prematurely only by calling the `QuitTrial[]` action, which will abort all currently running events, and remove all actions except trial actions schedule to execute upon the end of the trial.

15.2 Screen Stimulus Display

15.2.1 Screen Stimulus Loading

Loading of screen stimuli is covered in “Part 2: Graphic Environment Reference, Loading Text”, p247 and “Part 2: Graphic Environment Reference, Loading PICTs”, p248.

15.2.2 How a Screen Stimulus is Drawn and Cleared

Screen stimuli can be divided into two categories: *bitmapped*, and *non-bitmapped*. Event types `Text`, `PICT`, and `PasteBoard` are bitmapped stimulus types. `Document`, `Paragraph`, and `KeySequence` are non-bitmapped stimulus types.

Bitmapped Stimuli

Bitmapped stimulus types are drawn into an off-screen bitmap when they're loaded, and this bitmap is drawn to the screen when the stimulus is displayed, for the fastest possible drawing time.

When a bitmapped stimulus is drawn, PsyScope first sets the foreground and background colors, then waits for the next screen-refresh signal (see below, “15.2.2.1 Screen Timing”, p446) and then copies the bitmap to the screen using the copy-mode specified in the “mode” attribute for that event (see “14.2.1 Text”, p424).

The stimulus is cleared by the same process, except that the foreground color is set to be the same as the background color and the mode is set to `COPY`.

Non-bitmapped Stimuli

Non-bitmapped stimulus types were designed for situation in which millisecond timing is either non-critical, or in the case of the `KeySequence` type, impossible. They are drawn directly to the screen, text form, and depending on variables such as their size and what font they are in, could take a considerable amount of time to draw (see “Part 2: Graphic Environment Reference, Loading Text”, p247 for more information on fonts and font loading). To clear non-bitmapped stimuli, the entire port containing the stimulus is cleared.

15.2.2.1 Screen Timing

The biggest problem when drawing stimuli on the Macintosh screen tachistoscopically is that the image on a CRT screen is only refreshed periodically, generally on the order of 16 milliseconds, or less depending on the hardware.

The screen is refreshed by a beam of electrons within the CRT which sweeps across each line of the screen from left to right, starting with the top line and ending with the bottom. The time that it takes to do this is the refresh rate. Each time the electron beam returns to the upper left corner of the screen, the computer receives a signal, known as the *retrace signal*, or simply *retrace*.

In order to provide accurate timing within the constraints of this system, PsyScope employs a method known as *retrace synching* to reduce the error involved. When a bitmapped screen stimulus is to be drawn, PsyScope waits until it receive the retrace signal, and then draws the bitmap to the screen. The time at which the retrace signal is received is recorded as the actual onset time. This way the amount of error is constant and calculable for a particular position on the screen and doesn't depend on the size of the stimulus. (This assumes the drawing is done before the next retrace)

Timing on Multiple Screens

On systems with multiple screens drawing screen stimuli is somewhat more complicated, because each screen may have its own retrace rate, and even in the case that all the screens have the same retrace rate, it is highly unlikely that they will be in phase.

When a bitmapped stimulus is drawn on a system with multiple screens, it is broken up into the various parts that intersect each screen, and the parts are drawn sequentially, each one synched to the retrace of the screen it is being drawn on. In order to prevent experimental bias caused by the parts of the image always being drawn in the same order, it is possible to use the “MonitorOrder” experiment attribute to control the order in which stimuli are drawn to the monitors (see “13.1.6.1 Standard Experiment Attributes”, p360).

15.3 Playing Sound Stimuli

15.3.1 Loading Sounds

Loading sounds is covered in “Part 2: Graphic Environment Reference, Loading Sounds”, p249.

15.3.2 Sound Timing

15.3.2.1 Actual Duration vs. Recorded Duration

Sound stimuli in PsyScope have the advantage of playing asynchronously and having an inherent duration, thus not depending on the Trial Manager to clear them when they have finished. Unfortunately, PsyScope still must schedule and `EndEvent[]`, and a `ClearStim[]` action for each sound event, and record an actual duration for the sound when the `ClearStim[]` action is executed.

The duration of the sound is known when the sound is loaded, and the `EndEvent[]` is scheduled to run at that time, but if screen stimuli are being displayed concurrently with the sound, it is possible for the drawing or clearing of a screen stimulus to delay the execution of the `ClearStim[]` action, and thus cause the reported duration of the event to be longer than expected.

Ψ

Chapter 16. Configuring the User Environment

For each experiment, there will be a number of variables you may want to adjust each time the experiment is run, e.g., duration, intertrial intervals, stimulus appearance. Besides defining the experiment to run, the script can be used to define menus and display information for the Console.

16.1 Setting up the Menu

PsyScope allows you to design your own menus that will appear in the menu bar when the experiment is run. (These will appear along with the standard PsyScope menus — **File**, **Edit**, **Run**, **Utilities**, and **Windows**.)

To define your own menus, you must:

1. Create an entry named “Menu” that specifies all of the user-defined menus that will appear in the menu bar (this entry is already there in the standard script template). The content of the “Menu” entry should be a list of references to other entries; these entries — called *menu entries* — will define the individual menus.

For example, a menu bar that has **Subject** and **Stimulus** menus (in addition to the standard PsyScope menus) would be defined like this in the script file:

```
Menu:: Subject Stimulus
```

and the menu bar would look like this when the experiment is run:

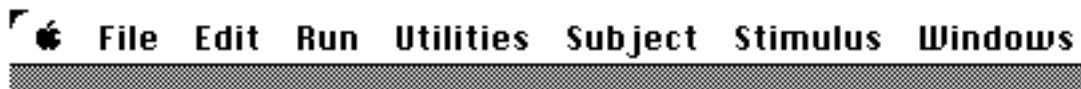


Figure 197 – Example menu bar

2. Create a separate *menu entry* for each individual menu. The content of each menu entry should be a list of references to *item entries*; each *item entry* defines a single item in the menu.

For example, the “Subject” entry in the example below defines a menu with contain five items:

```
Menu:: Subject Stimulus
```

```
Subject:: "Subject Number" "Subject Name" Age Group Handedness
```

The five items that will be under the **Subject** menu are: **Subject Number**, **Subject Name**, **Age**, **Group**, and **Handedness**:

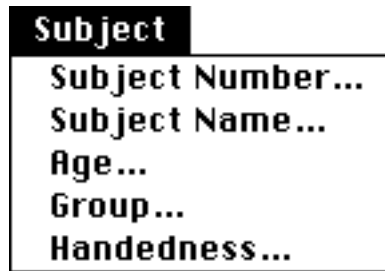


Figure 198 – Example script menu

3. Create each *item entry*. The content of an item entry will vary, depending on what kind of information is represented by the entry.

Note: Normally, you would use the Subject Info facilities of the graphic environment to set up your subject info items. This chapter uses subject information items as examples — independent of the standard Subject Info system — because it makes a good example. Nevertheless, you can use the information in this chapter to integrate items created in the standard Subject Info system into the menu system. See also “6.2 Subject Info”, p224.

16.1.1 Item Entries

Menu items are used for changing the parameters of your experiment and for obtaining information about your subjects. Usually, each item in a menu is associated with a single dialog.

When a menu item is selected, PsyScope looks in the script file for the matching entry. Thus, if a user selected **Subject Name** from the **Subject** menu above, PsyScope would search the script file for an entry named “Subject Name”, which might look like this:

```
Subject Name:: "George the Subject"  
  Dialog: Standard  
  Type: String
```

The “Dialog” attribute tells PsyScope how to go about getting new information from the user (i.e., what type of dialog to present so that the value of the entry can be changed). In this example, the script tells PsyScope to use a standard dialog box (made up of an editable text field, an **OK** button, and a **Cancel** button).

Input methods that can be specified in the “Dialog” attribute are called *dialog extensions*. The “Function” attribute can be used instead of the “Dialog” attribute, in which case the method is called a *function extension*. Some other dialog and function extensions include Check, Buttons, Checkboxes, Picture, FileLists, and LogInfo. These are all listed in “Chapter 17. Dialog and Function Extensions”, p467

The content of an item entry contains the current value of the menu item; in this example, the current value is the string `George the Subject`. This value will be placed in the text field of the dialog, which the user can then edit.

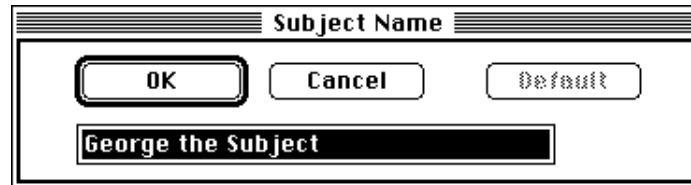


Figure 199 – Standard dialog

The “Type” attribute of the entry tells PsyScope that the expression that goes into the text field is `String`; i.e., it can be anything. If the input is not of the specified type, an error message to the user says so.

Remember that, in the script, a string that contains spaces must be enclosed in quotes. If the content of the menu item entry contained no quotes, like this:

```
Subject Name:: George the Subject
Dialog: Standard
Type: String
```

the content field would be treated as three strings instead of one; the Standard dialog would then display three edit text boxes, one for “George”, one for “the”, and one for “Subject”. Quotes insure that strings are treated as one expression. (If the original name had no spaces, but the user changed the name to one with spaces, quotes would be automatically placed by PsyScope.)

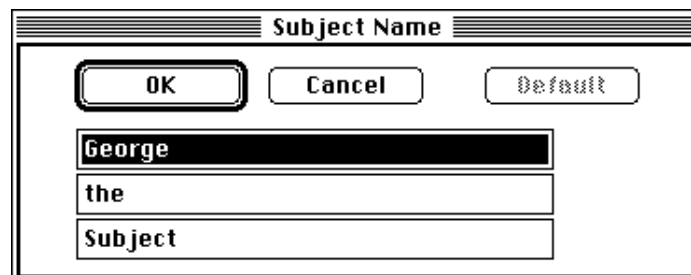


Figure 200 – Standard dialog with three fields

The “Type” attribute is specific to the `Standard` dialog extension. There are many other dialog- and function-specific attributes, all described in “Chapter 17. Dialog and Function

Extensions”, p467. In this section, we will describe other item entry attributes which are independent of the dialog or function extension.

16.1.1.1 Checkmarks

If a checkmark should be placed beside a menu item, the item entry should have an attribute “Check” with the value `True`.

The `Check` function can be used for item entries that store a `True` or `False` value. When `Check` is set as the function for a menu item entry (in the “Function” or “Dialog” attribute), the value of the “Check” attribute will be toggled each time the menu item is selected. Additionally, if the entry content has a `True` or `False` value, it will also be set to match the “Check” attribute. When the `Check` function is run, no dialog appears.

The menu handler looks for a “Check” attribute for any item entry and puts a check mark if the value of the attribute is `True`; this is independent of the dialog or function that is used for the item. The `Check` function is provided as a simple “dialog” that toggles the value of the “Check” attribute.

See also “Function: Check”, p476 for more information.

16.1.1.2 Range Checking

In some situations, you may want the dialog to accept only a certain range of values. For instance, suppose you defined the “Age” menu item as:

```
Age:: 21
      Dialog: Standard
      Type: Integer
      Range: (@Age>=16) && (@Age<=65)
      RangeFailMsg: "Age must be between 16 and 65"
```

Choosing **Age** from the menus produces a `Standard` dialog box with “21” in the text field. The “Type” attribute specifies that only integers may be typed into the text field.

The “Range” attribute specifies — through a `PsyScript` expression — that the dialog should only accept values that satisfy both of the following conditions:

- greater than or equal to 16: `(@Age>=16)`
- less than or equal to 65: `(@Age<=65)`

In other words, the dialog will only accept age values from 16 to 65. If the user types a value into the text field that is outside of the specified range (such as “14”), an error message will pop up, notifying the user that a range violation has occurred.

Whenever the user hits **OK** in a dialog, the “Range” attribute is checked for a `True` or `False` value; if the value is `False`, the user sees an error message and the dialog does not close.

You can specify the error message that you want to pop up by using the “RangeFailMsg” attribute. In the example entry, the “RangeFailMsg” attribute specifies the error message “Age must be between 16 and 65”.

16.1.1.3 Open/Close Alert

Sometimes, it is useful to check something in the script and possibly alert the user before or after a particular dialog is opened. This can be done by defining in the item entry an “OpenAlert” or “CloseAlert” attribute with a corresponding “OpenAlertMsg” or “CloseAlertMsg” attribute.

These attributes are used in the same way as the “Range” and “RangeFailMsg” attributes. For example, if the value of the “OpenAlert” attribute is `False` at the time the dialog is about to be opened, the “OpenAlertMsg” is given (and the dialog is not opened).

16.1.1.4 Menu Disabling

Sometimes, you will want a menu item to be enabled or disabled based on the value of other items. This is accomplished by using the “Enabled” item entry attribute.

For example, in the following menu, the **Hospitalization** item displays a dialog box with two buttons: one labelled **InPt** (for “Inpatient”) and one labelled **OutPt** (for “Outpatient”). (The `Buttons` dialog used for “Hospitalization” is described in “Dialog: Buttons”, p472.)

```
Medical Info::
    Hospitalization Medication
    "Months Since Hospitalization"

Hospitalization:: InPt
    Dialog: Buttons
    Buttons: InPt OutPt

Months Since Hospitalization:: 3
    Type: Integer
    Enabled: @Hospitalization == OutPt
```

The **Months Since Hospitalization** item should only be enabled if the response to **Hospitalization** is **OutPt** — that is, the subject is an outpatient; thus, the content of the “Enabled” attribute of the “Months Since Hospitalization” entry specifies the condition `@Hospitalization == OutPt`. Until the subject chooses the **OutPt** button in the **Hospitalization** dialog, the **Months Since Hospitalization** item will remain disabled.

“Enabled” may be an attribute of a menu entry also; if the value of the “Enabled” attribute is `False`, the entire menu is disabled.

16.1.1.5 ‘MenuName’ and ‘ItemName’

Sometimes, you may want a menu or a menu item to display a different name from the one that you use to refer to it in the script (since convenient entry names often are not nice menu item names). This is accomplished with the “MenuName” and “ItemName” attributes.

In the following example, the menu name displayed on the menu bar will be **Personal Info**, even though the menu entry is named “Subject” in the script.

```
Menu:: Subject Stimulus Duration

Subject:: "Subject Number" "Subject Name" Subject_Age Group
         Handedness "Medical Info"
         MenuName: "Personal Info"
```

The “ItemName” attribute is used to set the name of an individual menu item. In the example below, the item name in the menu will be **Age**, even though the item entry is named “Subject Age”.

```
Subject:: "Subject Number" "Subject Name" "Subject Age" Group
         Handedness
         "Medical Info"
         MenuName: "Personal Info"

Subject Age:: 21
             ItemName: Age
             Dialog: Standard
```

16.1.1.6 Title

Most dialogs will have a title to let the user know what information is being set by the dialog. The title of the dialog defaults to the name of the menu item that it is attached to, but it can be specified though a “Title” attribute.

```
Subject:: "Subject Number" "Subject Name" "Subject Age" Group
         MenuName: "Personal Info"

Subject Number:: 1
              Dialog: Standard
              Type: Integer

Subject Name:: George
             ItemName: Name
             Dialog: Standard
             Type: String
             Title: "Subject Name"

Subject Age:: 21
            ItemName: Age
            Dialog: Standard
            Type: Integer
            Title: "Subject Age"

Group:: High
      Dialog: Buttons
      Buttons: High Low
```

In the example above, the menu will be named **Personal Info** in the menu bar, and will have four items: **Subject Number**, **Name**, **Age**, and **Group**. The dialog titles for each dialog, respectively, will be **Subject Number**, **Subject Name**, **Subject Age**, and **Group**.

16.1.2 Submenus

Submenus (menus under menus) can be created through the script. Submenu entries are used in the same place as item entries (i.e., in the content of a menu entry), but they are recognized as submenu entries by having a “Submenus” attribute. The content of a “Submenus” attribute is the same as the content of a menu entry; the content and other attributes of a submenu entry are ignored (except the “Enabled” attribute; see “16.1.1.4 Menu Disabling”, p453).

For example:

```
Menus:: Subject Stimulus

Subject:: "Subject Number" "Subject Name" Age Group Handedness
         "Medical Info"

Medical Info::
         SubMenus: Hospitalization Medication
         "Months Since Hospitalization"
```

The submenu created from the example entries above would look like this:



Figure 201 – Example script submenu

16.2 The Console

The information that is displayed in the console can be configured in the script through a “Console” entry. The content of the “Console” entry should contain a list of references to other entries; the names of the referenced entries will appear with their values in the console window.

Items attached to the console are directly analogous to menu items. The “ItemName” and “Title” attributes may be used just as in the menu system. (The “Enabled” attribute is not supported, however.)

For example, the following Console entry:

```
Console:: "Subject Name"
```

links “Subject Name” to the console:

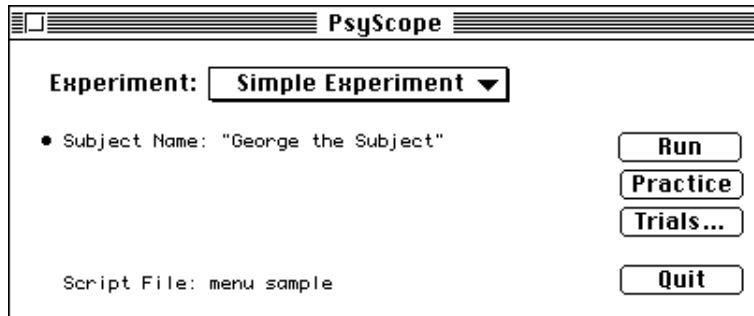


Figure 202 – Console window with item

16.3 Custom Options

Items can be attached to the **Options** submenu of the **Edit** menu by using the “Options” entry.

The **Options** submenu automatically displays a number of items that are discussed in “Part 2: Graphic Environment Reference, 7.6 Options”, p265. However, the **Options** submenu also displays a **Custom** item, which is a sub-submenu with user-defined items. Item entries referenced from the “Options” entry will appear in this submenu.

For example:

```
Options:: Cycles
Cycles:: 2
        Dialog: Standard
        Type: Integer
```

16.4 File Names

When files are needed by the script (e.g., input files for instructions or the output data file), the user can set the location and name of the file through a menu item. This can be done by including a menu item that uses the `FileLists` dialog.

The `FileLists` dialog can be used in several modes, described in “Dialog: FileLists”, p480. Here, we use the most elaborate mode, in which several sets of file names are handled by the dialog (and each set can contain multiple file names).

```

Menus:: Subject Stimulus

Subject:: "Subject Number" "Subject Name" Age Group Handedness
         "Medical Info"

Stimulus:: "Set Files"

Set Files:: "Sound Files" "Data File"
           Dialog: FileLists

Sound Files:: "My Disk:Sounds:Bloop" "My Disk:Sounds:Beep"
             Type: Locked
             Mode: Get

Data File:: "My Disk:Data Folder:Experiment Data"
           Type: Single
           Mode: Put

```

The content field of the “Set Files” contains a references to other entries; each of these entries represents a single set of files. The content of each set entry contains one or more file names.

This “Set Files” entry refers to two sets of files: “Sound Files” and “Data File”. The “Sound Files” set has two file names: “Bloop” and “Beep”. The “Data File” set has just one file: “Experiment Data”.

Files are always listed with their paths, although this path can be a *partial path*. See “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215 for more details.

Each file set entry has a “Type” attribute. The four possible values for the “Type” attribute are:

SINGLE - This is used when only one file name is specified in the content field of the entry and you do not want the subject to be able to add more files. With the “Type” attribute set to **SINGLE**, the specified file name can be changed, but no new file names can be added.

LOCKED - This is used when more than one file name is specified in the content field of the entry. The type **LOCKED** means that the present file names can be changed, but file names cannot be deleted from the list, and no new file names can be added to the list. In other words, the names of file names can be changed, but the number of file names in the list is constant.

UNLOCKED - The **UNLOCKED** type means that file names can be deleted from the list and new ones added, so that the number of file names in the list is variable.

DIRECTORY - When the **DIRECTORY** type is chosen, the content of the entry will specify a folder instead of a file. Lists of directories are not allowed.

Besides a “Type” attribute, each set entry also has a “Mode” attribute. The “Mode” attribute for “Sound Files” is set to **GET**; this means that the dialog for “Sound Files” will only get the names of files that already exist. The “Mode” attribute for “Data File” is set to **PUT**; this means that the dialog for “Data File” will allow the user to create a new file. **Put** never creates a *file* — it just creates a *file path name*.

The modification of file sets may be disabled with the “Enabled” attribute. If “Enabled” is `False`, the user will not be able to change, add, or remove file names in the set.

See also “Dialog: FileLists”, p480.

16.5 Log File

Basic log file information can be found in “Part 2: Graphic Environment Reference, 6.1.5 The Log File”, p222.

The current log file is specified in the “Log File” entry of the script. If the “Log File” entry is not present and you change the log file with **Change Log File...**, the entry will be created.

16.5.1 Log File Format

The log file follows the formatting convention similar to the one used by scripts. See “12.1 Components of a Script”, p319 for basic information on the scripting syntax. Information is stored after keywords that are followed by double- or single-colons.

Every time PsyScope is launched, a new “Application” keyword — with double-colons, so it looks like an entry — is written in the current log file, and the name of the PsyScope application, the date and the time are entered on its “content”. By default, each subsequent bit of information recorded in the log file is stored as an “attribute” of the “Application” entry.

Multi-line comments are placed in curly-brackets (“{ }”) like string values in the scripting language. Most other information, however, is not quoted.

The only keyword besides “Application” that is followed by double-colons is the “Subject-Name” keyword; this has to do with how the log file is used to calculate subject and run number (see “Part 2: Graphic Environment Reference, 6.2.3 Subject Number Calculation”, p230).

16.5.2 Logging script information

The `LOG` extension can be used to record the names of entries from the script followed by their values. To use the `LOG` extension, include a “Function” attribute in one of your script entries, with the value `LOG`. In the content of the entry, list the names of the other entries that you want to be included in the log. The name of the each script entry listed will be included in the log, followed by its content, when that entry is executed.

Note: The entry can be executed by putting it in the menus (see “16.1 Setting up the Menus”, p449), the Console (see “16.2 The Console”, p455), or in an execution entry (see “16.6.2 Execution Entries”, p460).

For example, consider the following set of entries:

```
SubjectName:: "Oscar the PsyScope Junky"
Age:: 12
Favorite flavor of ice cream:: pistachio
Log Subject Info:: SubjectName Age "Favorite flavor of ice cream"
                  Function: Log
```

When “Log Subject Info” is executed, the following will be added to the log file:

```
SubjectName:: "Oscar the PsyScope Junky"
Age: "Hello World"
Favorite flavor of ice cream: pistachio
```

Note: A single, rather than a double colon appears after each of the entry names in the log file, except for “SubjectName” which retains its double colon. See “16.5.1 Log File Format”, p458.

See also “Part 4: Scripting Reference, Function: LogInfo”, p482.

16.6 Special Entries

This chapter has so far described a number of special entries — including “Menus”, “Console”, and “Options” — which permit the user to customize the PsyScope environment in various ways. In addition to these, there are a number of other special entries used by PsyScope, as discussed below. *The names for these entries are reserved, and should not be used for any other purposes in a script.*

16.6.1 Experiments

“Experiments” is, of course, a reserved entry name. See “13.1.4 The ‘Experiments’ Entry”, p359.

16.6.2 Execution Entries

There are several *execution entry* names that you can use to automatically run dialogs or functions associated to item entries. The content of each execution entries references a list of item entries to run at the execution entry's particular time. These items do not have to be linked to the menu system of Console.

Items referenced in an execution entry are executed in the order which they are listed. If any one of the dialogs is cancelled (i.e., the user hits the **Cancel** button), then the operation associated with the execution entry (see below) is also cancelled, as well as the execution of other entries.

The execution entries, and their execution times, are shown in the table below.

Table 3: Execution Entries

Entry Name	Time	Cancels
“StartUp”	Script or project opened, after the log file is opened	Opening script or project
“ExperimentStart”	Script or project loading done or switched to new experiment, about to accept user input	Nothing
“BuildStart”	About to run, practice, or compile experiment	Run, practice, or compile
“RunStart”	About to run, after “Build-Start”	Run
“PracticeStart”	About to practice, after “BuildStart”	Practice
“RunEnd”/“PracticeEnd”	Done running or practicing, didn't break	Nothing
“RunBreak”/“PracticeBreak”	Stopped running or practicing due to break	Nothing
“ExperimentClose”	About to close script or switch scripts	Closing script or switching
“Shutdown”	About to close script (after “ExperimentClose”) or project — before “Save changes?” alert.	Closing script or project

16.6.3 Resources

The “Resources” entry can be used to open a resource file (or files) whenever the script is opened in PsyScope. The content of the “Resources” entry should be a list of the resource file names (with paths; see “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215).

This feature can be useful for storing script-specific dialogs, beep sounds, or picts in a separate resource file. See also “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216.

16.7 PsyScopeStdLib

“PsyScopeStdLib” is a built-in script fragment (a ‘TEXT’ resource in the PsyScope application) that is automatically included in a script. It contains a number of entry definitions that are commonly used in setting up an experiment and user environment.

If, for some reason, you do not want “PsyScopeStdLib” included in a script, you can suppress it by including the line `#NoIncludeStdLib` in your script; this is not recommended, especially for Factor format scripts. See also “Part 4: Scripting Reference, 12.4.4 #NoIncludeStdLib”, p326.

16.7.1 CurrentExperiment

The “Experiments” entry in each script has a “Current” attribute. This attribute is used by “CurrentExperiment” to reference one of the experiment entries in the script. A reference to “CurrentExperiment” is synonymous with a reference to the experiment currently selected for the script. Thus, “CurrentExperiment” may be useful when more than one experiment is defined in a single script and a reference to the currently selected script is desired.

16.7.2 Standard Menu Items

“PsyScopeStdLib” contains a number of pre-defined menu items, complete with their dialog configurations. To use them, you can include one of the pre-defined menus (see “16.7.3 Standard Menus”, p464) in your script or you can add the individual menu item you want to use in one of your own menus.

16.7.2.1 UserLevelMenuItem

The “UserLevelMenuItem” item entry (with item name **User Level**) uses the `Button` dialog. Using this dialog, the level of the user can be set **Programmer** (all privileges), **Graduate** (most privileges), or **Research Assistant** (fewest privileges). Many of the standard menu items make themselves disabled if the user level is too low.

By default, the user level is stored within “PsyScopeStdLib”; because “PsyScopeStdLib” is read-only, a change to the user level will not be saved when the script is changed or re-loaded. However, if the script contains a “UserLevel” entry, the value is stored there.

The user level set up is entirely script-based; nothing in PsyScope itself depends on the user level setting. It is there so that the user level can be used by scripters to enable or disable custom menu items.

16.7.2.2 SettingsMenuItem

The “SettingsMenuItem” item entry (with item name **Settings**) uses a scrolling checkbox dialog for setting special experiment flags, including those needed by installed extensions. This is the dialog that is used for **Special** in the graphic interface; see “Part 2: Graphic Environment Reference, Special”, p165.

The changes made in the “SettingsMenuItem” dialog will be reflected in the “Flags” experiment entry unless the script contains a “SettingsMenuItemOutput” entry, in which case the values are written there instead.

16.7.2.3 DataFieldsMenuItem

The “DataFieldsMenuItem” item entry (with item name **Data Output**) uses a checkbox dialog for setting the kind of information that will be recorded in the data file. This is the dialog that is used for **Data Info** in the graphic interface; see “Part 2: Graphic Environment Reference, Data Info”, p161.

The changes made in the “DataFieldsMenuItem” dialog will be reflected in the “DataFields” experiment entry unless the script contains a “DataFieldsMenuItemOutput” entry, in which case the values are written there instead.

16.7.2.4 ReverseVideoMenuItem

The “ReverseVideoMenuItem” item entry (with item name **ReverseVideo**) uses the dialog shown below to set the default foreground and background attributes in the experiment entry.

If the **On** button is selected, the “ForeColor” attribute is set to `white` and the “BackColor” attribute is set to `black`. If the **Off** button is selected, the “ForeColor” attribute is set to `black` and the “BackColor” attribute is set to `white`.

See “Part 4: Scripting Reference, 14.2.1.2 Text and Screen Experiment Attributes”, p429 for more details on the attributes.

16.7.2.5 InputDevicesMenuItem

The “InputDevicesMenuItem” item entry (with item name **Input Devices**) uses a scrolling checkbox dialog for setting which of the available input devices will be enabled during the experiment execution. This is the dialog that is used for **Input Devices** in the graphic interface; see “Part 2: Graphic Environment Reference, Input Devices”, p164.

The changes made in the “InputDevicesMenuItem” dialog will be reflected in the “Input-Devices” experiment entry unless the script contains a “InputDevicesMenuItemOutput” entry, in which case the values are written there instead.

16.7.2.6 TimerMenuItem

The “TimerMenuItem” item entry (with item name **Timer**) uses a scrolling radio button dialog for setting which of the available timing devices will be used during the experiment execution. This is the dialog that is used for **Timer** in the graphic interface; see “Part 2: Graphic Environment Reference, Timer”, p164.

The changes made in the “TimerMenuItem” dialog will be reflected in the “Timer” experiment entry unless the script contains a “TimerMenuItemOutput” entry, in which case the values are written there instead.

16.7.2.7 OptimizeMenuItem

The “OptimizeMenuItem” item entry (with item name **Optimization**) sets the kind of optimization that will be attempted when the experiment is compiled. The dialog that is used depends on the experiment format.

The changes made in the “OptimizeMenuItem” dialog will be reflected in the “Optimize” experiment entry unless the script contains a “OptimizeMenuItemOutput” entry, in which case the values are written there instead.

(See “Part 2: Graphic Environment Reference, Optimization”, p167 and “13.2.5 StimList/EventList Optimization”, p373 for information on optimization.)

16.7.2.8 Test BBox

The “Test BBox” item entry (with item name **Test Button Box**) opens the button box mask-setting and testing dialog. Use of this dialog for testing is described in “Chapter 19. Configuring the Button Box”, p513.

The button box mask set in the “Test BBox” dialog will be reflected in a “TempBBox-Mask” entry in “PsyScopeStdLib” unless the script contains a “BBoxMask” entry, in which case the mask is written there instead.

16.7.2.9 TurnOffBBox

The “TurnOffBBox” item entry (with item name **Don't Use BBox**) has no dialog; it modifies the “InputDevices” experiment entry to insure that BBox is not present, and checks the “Timer” experiment entry, changing BBox (if present) to Macintosh.

16.7.3 Standard Menus

Rather than adding individual menu items to your own menus, you can include a complete pre-defined menu in your experiment. “PsyScopeStdLib” contains two menu definitions: “AllStdLibMenuItems” (with menu name **Configuration**) and “StandardMenuItems”.

As its name implies, “AllStdLibMenuItems” contains all of the menu items described in the previous section. The “AllStdLibMenuItems” entry can also be used as a menu item; it uses the EntryList dialog to show all of the standard items in list form instead of menu form.

“StandardMenuItems” contains most of these menu items; a script created with the **New Script...** item in the **Design** menu uses this item set by default.

16.8 SubjectInfoLib

Keeping track of subjects is a common need in running experiments, and — at one time — there was no graphic environment in PsyScope to help you do things. “SubjectInfoLib” contains a menu and set of dialogs for doing this. To use “SubjectInfoLib”, you must include the line `#include "SubjectInfoLib"` in your script.

16.8.1 The Subject Menu

“SubjectInfoLib” contains a **Subject** menu, which can be used by including `Subject` in the menu list kept in the “Menus” entry. (Do not write a separate “Subject” menu entry; the entry is defined in “SubjectInfoLib”.) PsyScope will then create a **Subject** menu with the following items: **Subject Info**, **Name**, **Group**, **Number**, **Run**, **Age**, **Sex**, and **Data File**.

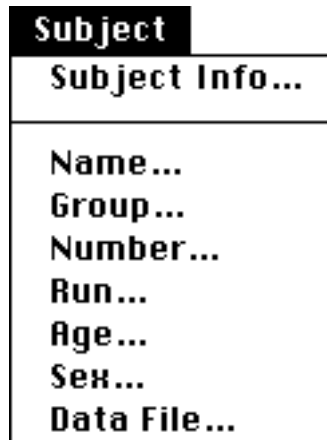


Figure 203 – “SubjectInfoLib” Menu

Selecting the **Subject Info** item opens an `EntryList` dialog (see “Dialog: `EntryList`”, p477) showing all of the current subject information, and allowing the user to double-click on any of the items to change it. Double-clicking on any item in the list will open the dialog box for that item.

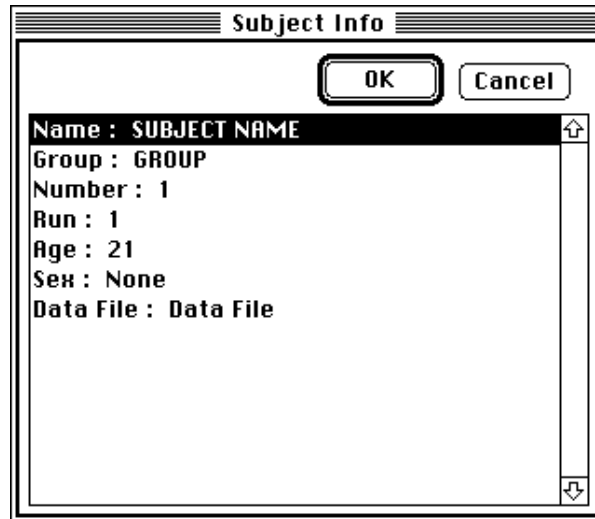


Figure 204 – “SubjectInfoLib” Subject Info dialog

Each of the items on the **Subject Info** list is the same as the items included in the **Subject** menu.

By default, the value of each item is kept in special entries within “SubjectInfoLib”; however, this information will disappear once PsyScope is turned off. If you want to keep the information from an item’s dialog, you must create an entry in your script; the name of the entry to which each item will write is shown in the following table:

Table 4: Subject Info Item Entry Outputs

Item	Menu Item Definition	Entry Name
Name	“Subject_Name”	“SubjectName”
Group	“Subject_Group”	“Group”
Number	“Subject_Number”	“SubjectNumber”
Run	“Subject_Run”	“RunNumber”
Age	“Subject_Age”	“SubjectAge”
Sex	“Subject_Sex”	“SubjectSex”
Data File	“Subject_DataFile”	“Data File”

It is expected that much of this information will be used in conduction with the `MakeFileName`, `Log` and `GetSubjNum` functions (see “17.2.6 Log File Related Functions”, p482).

You can customize the default **Subject** menu by including a menu entry in your script titled “SubjectMenuItems”. If this entry exists in the script, the **Subject** menu will use the items listed there instead of the standard items. Of course, you may also create a your **Subject** menu from scratch.

The items in the **Subject Info** dialog default to be the same as the **Subject** menu items. If you customize the “Subject” menu by including a “SubjectMenuItems” entry in your script, the **Subject Info** dialog will automatically be changed to match the menu. If you want to specify different items for the **Subject Info** dialog than you have in the **Subject** menu, you can do this by defining a “SubjectInfoDialogItems” entry.



Chapter 17. Dialog and Function Extensions

Dialog and function extensions are PsyScope Extensions that interact directly with the scripting language; i.e., these extensions know about entries and attributes, and can read and write tokens and expressions (see “Chapter 12. PsyScript Reference”).

Dialog extensions provide user interface dialogs so that information in the script can be viewed or modified by a non-scripting user. For example, the `Style` dialog reads tokens which represent a text style description (as in a `Text` event’s “Style” attribute), and allows the user to change this style. The `Standard` dialog reads one or more simple string values, and lets the user edit them.

Function extensions are similar to dialogs, except that there is no interaction with the user. The `LogInfo` function, for instance, reads tokens from the script and writes them to the log file. Functions can also modify the script; the `SubjNumAndGroup` function reads subject information from the script, and writes back subject number information.

Dialog and function extensions are always executed in the context of a **dialog entry**. This entry usually has a “Dialog”, “Function”, or “DCOD” attribute which names the extension to be run. An entry is **executed** when its dialog or function is called and given the entry as its context.

An entry is executed when:

- The entry is referenced in a script-defined menu and the item is selected by the user. (See “16.1 Setting up the Menus”, p449.)
- The entry is referenced in the Console and the item is selected by the user. (See “16.2 The Console”, p455.)
- The entry is referenced by an execution entry that is run. (See “16.6.2 Execution Entries”, p460.)
- The `Run()` PsyScript function is executed. (See “12.14.1.5 Other Operations”, p354.)
- The graphic environment invokes a dialog. (See the PSYXs programmer’s manual.)

From the PsyScope extension manager’s point of view, there is no difference between a dialog and a function; the distinction is, however, important to the user. A function extension should be specified through a “Function” attribute as opposed to a “Dialog” attribute; this affects only the menu configuration: no ellipses will be shown in the menu for entries using the “Function” attribute instead of the “Dialog” or “DCOD” attribute.

Most dialogs look for other attributes within the dialog entry to further specify the service that it should perform. “Msg” is a standard attribute is used by many dialogs; it is special in that setting the “Msg” attribute is referred to as “sending the dialog a message”. (Messages are also important when dialogs are invoked by the graphic environment, rather than through the script.)

Developer Note: Dialog and Function extensions are PSYXs of type ‘DCOD’. The “Msg” attribute of a dialog entry is handled specially, passing it in to the DCOD as the PSYX message rather than the usual pDialogCall message.

17.1 Calling Sequence for Dialogs

When an entry is executed, the following procedure is followed:

- 1) If the entry has a “RunBefore” attribute, it is used as an execution entry (see “16.6.2 Execution Entries”, p460); the dialogs/functions for the entries referenced in this attribute are run.
- 2) If the entry has an “OpenAlert” attribute and the attribute’s value is `True`, then the string value of the “OpenAlertMsg” attribute is presented in a user alert. (If there is no “OpenAlertMsg” attribute, no alert is displayed.)
- 3) The extension that is specified in the “Dialog”, “Function”, or “DCOD” attribute is loaded into memory and run.
- 4) The extension will have control at this point. For dialog extensions, the user’s actions within the dialog should be immediately reflected in the script.
- 5) If the extension uses a modal dialog and the user closes the dialog with **Cancel**:
 - 5.a1) The script is reverted to its former state and nothing else happens. The “former state” to which the script is returned is the state of the script *after* the “RunBefore” entries were executed.
 - 5.a2) If the dialog entry has a “RunFailMsg” attribute, the string value of the attribute is presented in a user alert.otherwise (i.e. there is no modal dialog or the user hits **OK**):
 - 5.b1) If the entry has a “Range” attribute and the value is `False`, execution returns to step 3. If there is a “RangeFailMsg” attribute, the string value of the attribute is presented in a user alert.
 - 5.b2) If the entry has a “CloseAlert” attribute and its value is `True`, then the string value of the “CloseAlertMsg” attribute is presented in a user alert. (If there is no “CloseAlertMsg”, no alert is displayed.)

- 5.b3) If the entry has a “RunAfter” attribute, it is used as an execution entry and the entries referenced in this attribute are run.

In addition, if there is a “NoDirty” attribute of the dialog entry with a `True` value, then the save state of the script will not be affected by changes made to it by the dialog (i.e. the **Save Script** menu item will not be hilited, even if a change is made to the script). This feature is available so that dialog operations that happen automatically need not force a “Save script?” dialog on the user.

17.2 Standard Dialogs and Functions Reference

The following is a detailed list of the set of predefined dialogs available in PsyScope. Wherever the term “attribute” is used below, it refers to an attribute of the dialog entry. All attributes must be specified, except:

- Attribute types are followed by square brackets (“[]”). The value inside the brackets is the default value and will be used in the absence of the attribute.
- Attributes are followed by “(optional)”. These need not be specified and have no default values.

17.2.1 Standard Configurable Dialogs

The standard dialogs are a collection of configurable dialogs for performing commonly needed dialog services. They include tools for setting strings or selecting from a set of radio buttons, a set of checkboxes, a scrollable list, or a pop-up menu.

17.2.1.1 The Standard Attributes

There are certain attributes which all of the standard dialogs recognize; there are listed here. All of these attributes are optional.

There is a special standard dialog, `Custom`, which is made up of standard parts rather than being a simple standard dialog. When an asterisk below is marked with an asterisk (*), the attributes are recognized by `Custom` parts, rather than by the `Custom` dialog itself, and should therefore be specified in the *part entry* instead of the *dialog entry*. (See “Dialog: Custom”, p473 below for more details.)

EnableParts *

This attribute is a list of boolean values that determine whether each control item is enabled or not: `True` means it is enabled, `False` means disabled. The first “EnableParts” value applies to the first item in the dialog, the second one applies to the second item, and so on.

The “EnableParts” attribute is read each time a change is made to any of the dialog items and the enabled state of each item is updated accordingly. If fewer enable values are spec-

ified than there are items, the values that are specified are used for the first few items, and the rest are always enabled.

Prompts *

For the `Standard/Fields` dialog, this attribute contains a text prompts to be placed in front of the edit boxes. There should be one prompt for each edit box.

For the other dialog types, this is a list of names that the user will see on the buttons, checkboxes, list items, or menu items. There should be one prompt for each item value; if there are fewer prompts than items, the actual values are used for the items without prompts. (See also “Dialog: Buttons”, p472.)

Default or Defaults *

All modal standard dialogs have a **Default** button. If the “Defaults” attribute is present, the button is enabled and the user can hit this button to set the items to the values in the “Defaults” attribute. There should be one default value for each item in the dialog.

Width *

This attribute specifies the width of the controls in the dialog, specified in pixels. The default varies depending on the part type, but it is usually about 400 pixels for items that take up a whole line in the dialog. One width is used for all of the items.

Height *

This attribute specifies the height of the controls in the dialog, specified in pixels and including the space between items. The default height varies depending on the part type, but it is usually about 20 pixels for items that take up a whole line in the dialog. One height is used for all of the items.

Margin *

This attribute specifies the horizontal position in pixels from the left of the dialog at which the item is to start. One margin value is used for all of the items.

Placement *

This attribute specifies a starting position in the dialog to place the items. The default is at the top left for the first set of items, or below the last set of items in a `Custom` dialog. This attribute overrides the “NewLine” attribute available in a `Custom` dialog.

Async

If this attribute has a `True` value, the dialog will be non-modal. The default is a modal dialog.

SetUp * and TakeDown *

The expressions in “SetUp” are evaluated before item values are read from the script when the dialog is being set up. The expressions in “TakeDown” are evaluated after item values are written back out to the script. The values of these attributes are ignored; they can be used for doing side-effect work on the script.

(For Custom dialogs, “SetUp” and “TakeDown” are read after the reflection operations are performed; see also “Dialog: Custom”, p473.)

DLOGx and DLOGy

These attributes set the position of the dialog on the screen. “DLOGx” and “DLOGy” specify the position of the top left corner of the dialog; -1 can be used to center the dialog in either direction. The default position is (-1, 50).

DLOG and PartIDs *

These are for advanced users who wish to create their own ‘DLOG’ and ‘DITL’ resources and assign parts to dialog items. The ‘DITL’ must have an **OK** button (item 1), a **Cancel** button (item 2), a user item at the same position and size of the **OK** button for the **OK** border (item 3), and a user item in which the dialog title is to be written (item 4).

The “DLOG” attribute specifies the name of a ‘DLOG’ resource to read.

“PartIDs” is a list of dialog item numbers for the parts in the dialog. If there are too few PartIDs, new items are created in the dialog. The standard dialogs assume that the dialog items are of the correct type.

17.2.1.2 Messages to the Standard Dialogs

Note: This feature is intended for developers and is useful only for operations outside the script (e.g. some interactions with the graphic environment).

Messages to the standard dialogs are used in a special way: the message is used as the definition for an inline entry to build on the dialog entry. Thus:

```
StandardValue:: 3
  Dialog: Standard
  Msg: "Type: Number"
```

is equivalent to:

```
StandardValue:: 3
  Type: Number
  Dialog: Standard
```

Thus, the message can contain attribute definitions that will be prefixed to the attributes that are defined in the normal way for the dialog entry. “RunBefore” and “RunAfter” attributes defined in the message are specially handled so that they work in the expected way.

17.2.1.3 Standard Dialog Descriptions

Dialog: **Buttons**

Entry Content: a single value representing the currently selected radio button

Attributes: “Buttons”
 “Direction” [Vertical]
 “Compact” [False]

This dialog has a list of radio buttons; the value of the entry will be a single value, equal to the value of one of the radio buttons. The values for the buttons are specified in the “Buttons” attribute. (The value of a button may be different from the name of the button that the user sees; see “Prompts*”, p470.)

The “Direction” attribute determines the orientation of the list of buttons within the dialog; its value can be `Vertical` or `Horizontal`.

If the “Compact” attribute has a `True` value, then the default height of the buttons is made smaller to allow more buttons in the dialog. This attribute is overridden by the “Height” attribute.

See also “17.2.1.1 The Standard Attributes”, p469.

Example:

```
ISI:: 500
     Dialog: Buttons
     Buttons: 500 3000
     Prompts: "Short ISI" "Long ISI"
```

The dialog will have two buttons labeled “Short ISI” and “Long ISI”. If the user selects “Short ISI”, the value of the entry “ISI” will remain 500; if the user selects “Long ISI”, the value of “ISI” will change to 3000.

Dialog: **CheckBoxes**

Entry Content: a list of values representing all the currently selected checkboxes

Attributes: “CheckBoxes”
 “Direction” [Vertical]
 “OffValues” (optional)
 “LeaveOthers” [False]
 “Compact” [False]

This dialog is similar to the `Buttons` dialog, except that the user sees a list of checkboxes. The number of items in the content of the entry usually depends on the number of boxes checked when the user hits **OK**.

The “Checkboxes” attribute assigns a value to each checkbox in the dialog; these are the values put in the entry’s content when the box is checked. (The name of the checkbox that the user sees may be different if the “Prompts” attribute is used; see “Prompts *”, p470.)

The “OffValues” attribute can be used to assign values to the *unchecked* state of each checkbox. (This attribute can be used to keep the number and location of the output values of the dialog constant.)

The checked/unchecked values of the checkboxes are written to the entry content in the same order that the checkboxes are listed in the “Checkboxes” attribute.

If the value of “LeaveOthers” is `True`, then values in the content of the entry which are not related to any of the checkboxes will be ignored. If it is `False`, then unrecognized tokens will be deleted.

If the “Compact” attribute has a `True` value, then the default height of the checkboxes is made smaller to allow more buttons in the dialog. This attribute is overridden by the “Height” attribute.

See also “17.2.1.1 The Standard Attributes”, p469.

Dialog: **Custom**

Entry Content: a list of references to part entries, if there is no “Parts” attribute

Attributes: “Parts” (optional if list in content)

Part entry attributes: “PartType” [Standard]
 “NewLine” [True]
 “Reflect” (optional)
 “ReflectOut” (optional)

This dialog can be used to mix dialog item (or “part”) types. It takes a list of references to entries which specify a set of parts; each part set contains one of the following:

- a set standard text fields
- a set of radio buttons
- a set of checkboxes
- a scrolling list
- a pop-up menu

Each part set is defined in a separate entry, called a *part entry*. The format of a part entry is basically the same as for a dialog entry having that type of part, except that the part set type is defined by a “PartType” attribute instead of a “Dialog” attribute. Also, there are a few additional attributes which part entries can have.

The “Parts” attribute of the dialog entry should contain a list of references to the part entries. The parts are added to the dialog in the same order which they are listed in this attribute. If there is no “Parts” attribute, the dialog entry’s content is read for the list of references.

The values manipulated by the parts of a `Custom` dialog are written to into the content of the part entries. However, these values can be moved in special ways between the dialog and part entries by using the “Reflect” and “ReflectOut” part entry attributes.

For all parts with a “Reflect” value `True`, the `Custom` dialog tries to copy items from the content of the dialog entry to the part entries. A value is moved only if it makes sense for the part type; e.g., the value matches one of the button names for a button part. (Standard parts always accept reflected values.)

“ReflectOut” works in the opposite way: for all of the parts with a `True` value for this attribute, the values in the content of the part entry are copied back to the dialog entry.

Note: On most cases, using token references at the content of the dialog entry works better than “Reflect” and “ReflectOut”. These attributes are provided for advanced use.

The `Custom` dialog also looks for a “NewLine” attribute within each part entry; if it is `False` the part set will be positioned to the right of the previous part set in the dialog; the “NewLine” attribute defaults to `True`, in which case the new part set begins below the previous set.

Dialog: `ItemList`

Entry Content: value(s) representing the currently selected list item(s)

Attributes:

- “ListItems”
- “LineCount” [4]
- “Single” [`True`]
- “RSRCItems” (optional, overrides “ListItems”)

This dialog is very similar to the `Buttons` dialog, except that the user sees a scrolling list of items and selects one. If the “Single” attribute is `False`, however, this dialog works more like the `Checkboxes` dialog, because the user may select more than one list item.

The “LineCount” attribute can be used to set the number of items visible at once in the list (i.e., how tall the list is, in items).

If a “RSRCItems” attribute is present, the dialog shows a list of available resources of the specified type, and the “ListItems” attribute is ignored. For example, a value of `PICT` in the “RSRCItems” attribute creates a list of all available ‘PICT’ resources.

See also “Dialog: Buttons”, p472 and “17.2.1.1 The Standard Attributes”, p469.

Dialog: PopUp

Entry Content: a single tokens representing the currently selected menu item

Attributes: "PopUpItems"

This dialog works just like the `Buttons` dialog, except that the choices are shown in a pop-up menu, instead of a list of radio buttons.

See also "Dialog: Buttons", p472 and "17.2.1.1 The Standard Attributes", p469.

Dialog: Standard or Fields

Entry Content: an arbitrary list of text values

Attributes: "Type" or "Types" [string]
 "Static" [False]
 "PromptWidth" (optional)
 "TitleLine" [False, True overrides "Static" and "Margin"]
 "Lines" [1]
 "ForceCount" (optional)
 "Initialize" (optional)
 "VRange" (optional)
 "StringLength" (optional)

This dialog lets the user modify the tokens on the content line of the entry. If there are n tokens in the dialog entry content, the dialog will have n edit fields.

The "Types" attribute specifies a list of types (where each type is one of `string`, `integer`, `number`, `rational`, or `boolean`), in parallel with the values in the content line. The dialog makes sure that the value of the i th field has the correct form for the i th type specified in the "Types" attribute. The default type is `string`, which places no constraints on the field's format (except possibly its length; see below).

There are some special conventions that allow the blocking of types and prompts for the dialog. If there are less tokens in the "Types" attribute than in the content, then the values of the "Types" attribute will be re-used (starting from the beginning of the list) until all of the tokens are typed. The "Prompt" attribute works the same way. Thus, if an entry has several tokens that all should be integers, `integer` only needs to be specified once in the "Types" attribute.

If the "Static" attribute is present with a `True` value, the user will be shown the value of fields in the dialog, but will not be able to edit them. (When it is not possible to re-assign a field value due to the use of a `PsyScript` expression, that individual field will automatically be made static.)

The "PromptWidth" attribute can be used to specify how much room to allow for a prompt in front of each of the edit boxes. The same prompt width is used for all of the fields.

The "Lines" attribute determines the number of visible lines of text in each edit field. This attribute also applies for all fields.

The “ForceCount” attribute can be used to force the number of edit fields in the dialog, even if more or less values initially appear in the entry’s content; when **OK** is hit, one token will be created for each edit field.

When “ForceCount” is used and the number of fields is greater than the number of initial values, you can specify a list of default values in the “Initialize” attribute. There should be one value for each field.

The “VRange” attribute can be used with the number types (`integer`, `number`, and `rational`) to set a lower and upper bound on the values. The bounds are given as a list of two numbers in “VRange”, with the lower bound first.

The “StringLength” attribute can be used with the `string` type to constrain the number of characters allowed in the string to a single value. (“StringLength” is an exact constraint, not an upper bound.)

See also “17.2.1.1 The Standard Attributes”, p469.

Example:

```
My Entry:: Binky 25 "3/4" Juliet 16 "2 3/8"  
  Dialog: Standard  
  Types: string integer rational  
  Prompts: "Name" "Age" "Favorite Fraction"
```

The dialog will have six edit boxes. Each set of three boxes will be labeled “Name”, “Age”, and “Favorite Fraction”. The values in the first and fourth boxes can be anything; the values in the second and fifth boxes must be integers, and the values in the third and sixth boxes must be rational. (Note that the blocking convention is used here to set prompts and type.)

17.2.2 Miscellaneous Dialogs and Functions

Function: **check**

Entry Content: nothing or one token which is true or false

Attributes: “Check”

This function reads the value of the “Check” attribute and toggles its boolean value. If there is a token on the content line, it sets it to `True` or `False`, as determined by the current value of the “Check” attribute. The menu handler looks for a “Check” attribute for all menu items; when it is `True`, the menu item is checked.

The “Check” function should be used with a “Function” attribute instead of a “Dialog” attribute; this is so that, if the entry is linked to the menu system (see “16.1 Setting up the Menus”, p449), ellipses will not appear after the item name (since no dialog is opened).

Dialog: EntryList

Entry Content: a list of entry references (ignored if “EntryList” attribute is present)

Attributes: “EntryList” (optional if entries are listed in the content)

This dialog gives the user a list of entries and their values (similar to the central Console; see “16.2 The Console”, p455); when the user double-clicks on one of the list items, the dialog for the selected entry is run. The “EntryList” attribute is a list of references for the displayed entries; if the attribute is not present, the content line of the dialog entry is read for the references. If the user hits **Cancel** to close the EntryList dialog, all changes made to entries the list are canceled also.

If any of the referenced entries have an “ItemName” or “Enabled” attribute, these will be used for the list item similar to the way that they are used for menu items. (See “16.1.1.4 Menu Disabling”, p453 and “16.1.1.5 ‘MenuName’ and ‘ItemName’”, p453.)

Dialog: Editor

Entry Content: a single text value

This dialog opens a standard text editor window for editing the string in the dialog entry’s content. This string can be any length, and may include carriage returns. (The graphic environment uses this dialog to the set stimulus for Paragraph events.)

17.2.3 Input Device Dialogs

Dialog: BBox

Entry Content: a list of tokens for a BBox[] event mask

This dialog can be used to test the button box, set the serial port to which the button box is connected, and create a button box input condition parameter list. The tokens of the content line will be something like `BUTTON1` or `VOICE_END` which may be used in the experiment structure to specify which responses should be recorded. (See “14.3 Conditions and Inputs”, p437 for more information.)

If the port is set, the results are output to the “BBoxPort” experiment attribute. (See “14.2.8.2 BBox Experiment Attributes”, p437.)

Dialog: MouseState

Entry Content: a list of tokens for a Mouse[] event mask

This dialog can be used to create a list of condition parameters for the Mouse input device. See also “14.3 Conditions and Inputs”, p437.

Dialog: `keyState`

Entry Content: a list of tokens for a `key[]` event mask

This dialog can be used to create a list of condition parameters for the `key` input device. See also “14.3 Conditions and Inputs”, p437.

17.2.4 Stimulus Attribute Dialogs

Dialog: `positions`

Entry Content: one port/point specification or reference

Attributes: “Locked” [False]
Port entry attrbs: “Shape” (optional)
 “AlignmentPoint” (optional)
 “Points” (optional)
 “Locked” [False]

This dialog is used to specify the location and appearance of a port — and the location of a set of associated points — to be used in an experiment. The dialog has two modes: *port mode* — in which the content output is a port description — and *point mode* — in which the content output is a point description. Port mode is default; point mode is specified by using the “PointMode” message.

Ports and points are created as separate entries, independent of the dialog entry; a list of ports defined in the script is kept in a “PortNames” entry. Each port has an associated list of points kept in its “Points” attribute.

The output of the dialog in the content of the entry is `PortName(port)` or `PointName(point)`; these are PsyScript functions that simply perform `GetToks()`. If the initial content of the dialog entry is not empty and not already in this form, then the content is assumed to be an actual port or point description; an extra port/position is shown in the user’s list representing this port/point.

In port mode, if the “Locked” attribute of the dialog entry is `True`, the user will not be able to create and delete port specifications. If the value of the “Locked” attribute in the port entry is `False`, the user will be able to add or remove points associated to the port.

Each port description in the content of a port entry has this form:

h_pos width v_pos height border_thick

Each point description in the content of a point entry has this form:

h_pos h_relative_to h_align v_pos v_relative_to v_align

The meanings of these specifications are given in “14.2.1.1 Text and Screen Attributes”, p424. The use of the “Shape” and “AlignmentPoint” attributes is also explained. (Both “Shape” and “AlignmentPoint” are specified as attributes of a port entry.)

Example:

```
PortNames:: PortOne PortTwo
          Dialog: Positions

PortOne:: Center 50% Center 50% 1
          Points: PointForOne

PointForOne:: Left Box Left Center Box Center

PortTwo:: Left 50 Top 50 0
          Points: PointAForTwo PointBForTwo

PointAForTwo:: Center Screen Center Center Screen Center

PointBForTwo:: Right Screen Right Bottom Screen Bottom
```

In this example, the first port is centered on the screen, half as wide and tall as the screen, and has a 1-pixel border. It has one point that is aligned with the middle of its left edge.

The second port is in the top left corner, fifty pixels wide and fifty pixels high. It has two points associated with it: one in the center of the screen and one at the bottom right of the screen.

Dialog: **style**

Entry Content: a list of tokens describing a text font, size, face, etc. of the form:

font size face mode color

Attributes: “Sample” [“Stimulus”]

This dialog is used to specify a format to be used for text drawing. The output is usually used as the value of the “Style” attribute of a text event in an experiment

font is the name of the font or a ‘FOND’ resource number.

size is the size of the text in pixels.

face is a string of face specifications; the string may be empty or contain any combination of bold, italic, underline, outline, shadow, condense, and extended as a single string.

mode determines the transfer mode of the text and should be *one* of copy, or, xor, or inverse.

color can be one of the defined colors - red, blue, green, black, cyan, magenta, yellow, or white - or a string containing three numbers between -32768 and 32767 giving the red, green, and blue components of the color.

The dialog displays a sample string so the user can see how the text will look. The “Sample” attribute specifies what string should be used as sample text.

Example:

```
My Text:: Geneva 14 "bold italic" copy black
      Dialog: Style
```

This entry specifies a text style using 14 point Geneva in boldface italics.

17.2.5 File Name Dialogs

Dialog: **FileLists**

Entry Content: either a list of direct references, or file(s) with full pathname(s)

Attributes: “JustOne” [False, overrides “OnlyOneSet”]
 “OnlyOneSet” [False]

File entry Attributes: “FileType” [Single]
 “Mode” [Get]
 “ReverseNotation” [False]
 “Enable” [True]

This dialog works in three modes:

- If “JustOne” is `True`, the content of the entry is a single file (or directory) name to be set by the user.
- If “OnlyOneSet” is `True`, the dialog entry is used as the only *file entry*.
- If both “JustOne” and “OnlyOneSet” are `False`, the content of the dialog entry is a list of references to file entries.

A *file entry* is used to hold a list of file/directory names having the same type. The names are listed in the file entry’s content. The attributes “FileType”, “Mode”, “ReverseNotation”, and “MacType” specify what kind of files should be listed at the file entry. (These attributes are specified in the dialog entry for “JustOne” mode.)

The “FileType” attribute can be `single`, `locked`, `multiple`, or `directory`. If the type is `multiple`, the user will be able to add or delete file names in the list. A file type of `directory` specifies that the tokens in the content are directory paths rather than file names. `single` and `locked` are synonymous: files in the list can be changed, but not added or deleted.

The “Mode” attribute can be `get` or `put`; `get` mode means that the user should select files that already exist, while `put` mode will request the name of a file to create. (The dialog does not create the file in `put` mode; only a name is generated.) The “Mode” attribute does not apply when the file type is `directory`.

All file names are written to the script using relative paths, when possible (see “Part 2: Graphic Environment Reference, 6.1.2.1 Relative Paths”, p215).

For `get` mode, a 4-letter file type can be specified in the “MacType” attribute. For example, the value `TEXT` would allow the user to select only text files. Up to four types can be specified.

By default, the file names on the token line will be output according to the usual Macintosh convention; e.g., “My Disk:Folder A:File X”. If the “ReverseNotation” attribute is `True`, the output will be of the form “File X @ My Disk:Folder A”. (See also “Part 2: Graphic Environment Reference, 6.1.2.2 Reverse Notation”, p215.)

If the “Enable” attribute is set to `False`, the filename(s) can be seen, but not changed.

The file type and modes can also be specified through the extension message. This message can contain the keywords `justone`, `onlyoneset`, `single`, `locked`, `multiple`, `directory`, `get`, and `put`. A file type is specified with `MacTypeXXXX` where `XXXX` is the 4-letter type (up to four types can be listed; no separator is used between multiple types).

Function: `MakeFileName`

Entry Content: a single string (returned by the dialog)

Attributes: “Strings”
 “UseInitials” (optional, read as token-inherited)

This function creates a new string (usually to be used as a file name) based on information provided in the “Strings” attribute. It takes the tokens listed in the “Strings” attribute, concatenates them, and returns the resulting string in the content of the entry.

If the “UseInitials” attribute is set to `True`, `MakeFileName` takes only the first letter of each word (space-delimited) in a value. The “UseInitials” attribute is read as an inherited attribute for each token, so that initialing can be restricted to only a few values by creating inline token references which include the “UseInitials” attribute. (See “12.11 Inherited Attributes”, p344 and “12.11.1 Inheritance and Token Reference Inline Entries”, p346.)

Example:

```
The Experiment:: "Who Cares"
      Extension: Exp
      ...

SubjectName:: "William the Conqueror"

SubjectGroup:: Control

SubjectNumber:: 3

Auto Data File:: "WtC-C3.Exp Data File"
      Function: MakeFileName
      Strings:(@SubjectName UseInitials: TRUE) "-"
```

```
(@SubjectGroup UseInitials: TRUE)
@SubjectNumber "."
"The Experiment"->extension
>Data File"
```

Note that the values of the “SubjectName” and “Group” entries are initialed, while the extension attribute of the “The Experiment” entry and the string “Data File” are not.

Function: `Picture`

Entry Content: a ‘PICT’ file name or ‘PICT’ resource name

This dialog lets the user find a ‘PICT’ file using the standard file dialog, or choose a ‘PICT’ resource from one of the open resource files (or from one of the files listed in the “Resources” experiment attribute; see “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216).

17.2.6 Log File Related Functions

The `LogInfo` and `GetSubjNum`, and `SubjectNumAndGroup` functions work closely together; using the log file, they keep track of the subjects that are run, how many times each subject has been run, and assign subject to groups in the experiment.

Function: `LogInfo`

Entry Content: a list of references

This function is used to record information in the current log file. For each reference in the content of the dialog entry, it records (on a single line in the log file) the referenced entry name followed by the content of that entry. Entry names recorded in the log file by the `LogInfo` function are recognized as keywords by the `GetSubjNum` function.

For example:

```
RunStart:: LogSubjInfo
LogSubjInfo:: SubjectName SubjectNumber
              Function: LogInfo
SubjectName:: "Oscar the PsyScope Junky"
SubjectNumber:: 42
```

In this example, at the beginning of the experiment the “RunStart” execution entry will cause “SubjectName” and “SubjectNumber” to be logged like this:

```
SubjectName:: "Oscar the PsyScope Junky"
SubjectNumber: 42
```

The special script-like formatting is built into `LogInfo`: if an entry is named “SubjectName”, it is followed by double-colons; otherwise it is inset and followed by a single colon. See also “16.5 Log File”, p458.

Dialog: `GetSubjNum`

Entry Content: first value is ignored, but set to `True` after running

Attributes: “GroupSpecs”
 “Modulo (optional)”
 “SubjectNumber” (created if missing)
 “RunNumber” (created if missing)
 “SubjectCount” (created if missing)
 “RunCount” (created if missing)
 “GroupRunCount” (created if missing)
 “RestrictAlert” (optional)
 “Options” (optional)

This function assumes: 1) That the current log file contains all of the information needed to determine how the current subject fits into the experiment group structure, and 2) there is a “SubjectName” entry that contains the name of the current subject.

`GetSubjName` is given some information about the current subject (in the “GroupSpecs” attribute); this information determines a group which — by definition — includes the current subject. The task of the `GetSubjNum` dialog is to determine how many other subjects have been in this group and how many times this particular subject has been run in the experiment.

`GetSubjNum` calculates five numbers, returned in attributes of the dialog entry:

“SubjectNumber”: The number of the current subject within the current group of the current experiment (starting with 1).

“RunNumber”: The number of this run for the current subject within the current group of the current experiment (starting with 1).

“SubjectCount”: The number of the current subject within *all* groups of the current experiment (starting with 1).

“RunCount”: The number of this run for *all* subjects within *all* groups of the current experiment (starting with 1).

“GroupRunCount”: The number of this run for *all* subjects within the current group of the current experiment (starting with 1).

These attributes are created if they do not already exist; if there is an entry with the same name as the attribute, the content of the entry is set, as well.

For `GetSubjNum` to work properly, therefore, information about subjects that have been run in the experiment must be recorded in the log file. `PsyScope` automatically records the running of each experiment in the log file with the “ExperimentRun”

keyword; however, the `LogInfo` function must be used to record the rest of the information about subjects (before the experiment is run; this can be done with the “RunStart” execution entry).

`GetSubjNum` searches the log file for keywords followed by values. `GetSubjNum` expects to find a “SubjectName” keyword for each subject that has been run, preceding the “ExperimentRun” keyword; the keywords and information between “SubjectName” and “ExperimentRun” are used to determine whether the logged subject belongs to the current group.

The “GroupSpecs” attribute specifies the criteria that must be matched for a logged subject to be in the current group; if no “GroupSpecs” are given, all subjects that were run for the current experiment are placed into the same group. The value of the “GroupSpecs” attribute is a list of pairs, where each pair is a criteria keyword to find in the log file and the value that the keyword must have.

The keyword search is started after a “SubjectName” keyword is found and ends when the “ExperimentRun” keyword is found for the current experiment (or is restarted if another “SubjectName” keyword is encountered). The logged subject is determined to be in the current group if all of the criteria keywords are found and have the correct value.

The “Modulo” attribute can be used to modify the criteria matching procedure; it owns a list of sub-attributes, one for each keyword listed in “GroupSpecs”. A 0 value in such a sub-attribute indicates that the corresponding criteria should match exactly, but any other number n indicates that the given value and logged value should match numerically modulo n . The constant `Groups` can be used to specify n as the number of groups linked to the experiment.

The “RestrictAlert” attribute controls when the user is warned about duplicate subject names in the log file; its value can be:

`False` - Always warn about duplicate subject names.

`Experiment` - Warn if the previous subject name was recorded for the current experiment.

`Script` - Warn if the previous subject name was recorded since the script was opened.

`Application` - Warn if the previous subject name was recorded since PsychoScope was started.

`Never` - Do not warn about duplicate subject names.

The “Options” attribute can contain one or more flags:

`USE_HIGHEST` - By default, the value returned in the “SubjectNumber” attribute is one greater than the number of subjects found; with this flag, the value is set to be one greater than the highest “SubjectNumber” of all of the

subjects found. The `USE_HIGHEST` option should be used if the log file is missing information about subjects who have been run earlier (e.g., it has been switched or purged) and subject numbering should continue sequentially.

Note: The `use_highest` option requires that the “SubjectNumber” for each subject previously run has been recorded in the log file.

`RESTRICT_ALERT` - This flag is equivalent to setting the “RestrictAlert” attribute to `Experiment`.

`ONE_EXPERIMENT_PER_ENTRY` - As described below, group matching is performed by checking keywords that occur in the log file between an instance of “SubjectName” and the “ExperimentRun” keyword. If this flag is on and an “ExperimentRun” keyword is encountered that is *not* for the current experiment, then the match checking starts again; otherwise, matching simply continues until another “SubjectName” or “ExperimentRun” keyword is found.

Example:

```
RunStart:: EvaluateSubjectNumber LogSubjInfo

LogSubjInfo:: SubjectName SubjectNumber
              Function: Log

EvaluateSubjectNumber:: TRUE
                      Function: GetSubjNum
                      Options: USE_HIGHEST
                      SubjectNumber: @SubjectNumber

SubjectName:: "Oscar the PsyScope junky"

SubjectNumber:: 12
```

“RunStart” is an execution entry (see “16.6.2 Execution Entries”, p460) that will call the dialogs associated with the “EvaluateSubjectNumber” and “LogSubjectInfo” entries – in that order – just before the experiment is run. This will calculate the subject number from the log file (via `GetSubjNum`), and then record the subject’s name and number in the log file.

Dialog: `SubjectNumAndGroup`

Entry Content: ignored

Attributes: “ComputeNumbers” [True]
 “AutoGroup” [True]
 “Notify” [False]
 `GetSubjNum` attributes

This function first calls `GetSubjNum` with the same dialog entry, unless “ComputeNumbers” is `False`. If “AutoGroup” is `False`, it does nothing else.

Otherwise, this function assumes that the current script is in Factor format and it sets the group in the current experiment structure; the selection is based on the information in “GroupSpecs” and criteria information defined at the group entries in the script.

The selection works much like `GetSubjNum`, but instead of reading the log file a “Criteria” attribute is checked for each group; sub-attributes of the “Criteria” attribute and their values take the place of keywords. The groups are checked in order until one of the groups matches the criteria in “GroupSpecs”. If no matching group is found, the user is alerted.

If the dialog entry has a “Notify” attribute with a `True` value, the user is given a standard alert dialog telling which group was selected by the function.

Part 5:

Appendices

Chapter 18. Error Messages 489

Chapter 19. Configuring the Button Box 513

Chapter 20. Creating Picture Resources 517

Chapter 21. Creating SoundEdit™ Sound Files 519



Chapter 18. Error Messages

18.1 Error Numbers

Each error message has a number. The numbers are assigned as follows:

n – Macintosh errors

M*n* – Global and memory errors; see “18.2 Global and Memory Errors”, p489.

S*n* – PsyScript errors; see “18.3 PsyScript Errors”, p490.

IN*n* – User environment errors; see “18.4 User Environment Errors”, p493.

D*n* – Graphic environment errors; see “18.5 Graphic Environment Errors”, p495.

FACT*n* – Factor format compiler errors; see “18.6 Factor Format Errors”, p499.

TM*n* – Trial Manager errors; see “18.7 Trial Manager Errors”, p502.

SCR*n* – Screen Manager errors; see “18.8 Screen Manager Errors”, p507.

SND*n* – Sound Manager errors; see “18.9 Sound Manager Errors”, p510.

BB*n* – Button Box errors; see “18.10 Button Box Errors”, p511.

PsyExtension errors are reported by prefixing an error number with its id; check the documentation for the particular extension for a listing of the errors it returns.

18.2 Global and Memory Errors

M1 **Insufficient memory errors**

Try giving your copy of PsyScope a larger Multifinder partition through the **Get Info** dialog in Finder.

M2 **File error**

The specified file could not be located or it was already open.

M98 Memory errors

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

18.3 PsyScript Errors

S0 Entry or attribute not found

No entry or attribute with the given name was found. The given name is incorrect or the entry/attribute is not in the script.

S1 Syntax errors

See “Part 4: Scripting Reference, 12.2.3 Entry Syntax”, p321 for the syntactic rules for a script.

S2 Unknown tokens

There are extraneous characters following the closing bracket or parenthesis of an inline entry definition (without intervening blanks) and the interpreter doesn’t know what to do with them. See also “Part 4: Scripting Reference, 12.9 Inline Entries”, p335.

S4 Bad access type

The valid access types are `Sequential`, `Random`, `RRandom`, or a number. See also “Part 4: Scripting Reference, 12.8.2 Access Type”, p331.

S5 Wrong sizes for linking

Only lists of the same size may be linked. See also “Part 4: Scripting Reference, 12.8.3 Linking”, p332.

S8 If Conditional

There was an error reading the value for the switch (the first parameter) in an `if()` function call.

S9 TRUE Value

There was no second argument (always required) in an `if()` function call.

S10 Iteration parameter missing

The `Iterate()` function requires three reference parameters; one or more are missing.

S11 Duplicator count too large

The duplication count (applied with the `~` operator) appears to be larger than PsyScope can handle.

S13 Illegal use of assignment

Only the value of literal tokens may be reassigned; an assignment is being applied to a non-reference expression.

S14 Function requires different parameters

The parameters to the function are not of the correct type or count. See “Part 4: Scripting Reference, 12.14 Script Operators and Functions Summary”, p349.

S15 Infinite loop

The interpreter encountered a very complex expression or self-referencing expression which it could not evaluate within its finite stack space. Increasing PsyScope’s memory partition (using **Get Info** in Finder) can sometimes allow PsyScript to evaluate more complex expressions. Self-referencing expressions are scripting errors that cause infinite loops; for example: `Bad:: @Bad.`

S16 Bad index for previous current

An operation attempted to use one of the previous current items for a list (using `GetPrevCurrent()`, etc.), but the given index is greater than the number of previous items remembered (as designated by the “SaveCurrents” attribute). See also “Part 4: Scripting Reference, 12.8.6 SaveCurrents”, p334.

S17 Can’t evaluate reference

A non-string reference value was encountered where a strictly string-valued value was expected. This can only happen if you use inline references, the `^` operator, or one of the reference-returning functions (e.g. `Column()` or `Inherited()`).

S18 Zero-length list

An attempt was made to use an entry with no tokens as a list, but lists cannot be empty. See also “Part 4: Scripting Reference, 12.8 Lists”, p329.

S19 Invalid sublist size

A sublist cannot be made such that it has zero or negative length, and a mapping may not be made with the mapping list larger than the parent list. See also “Part 4: Scripting Reference, 12.8.7 Sublisting”, p334.

S20 Lists are linked

This is an indication of circularly linked lists in your script. Make sure that if list “A” is linked to list “B”, that list “B” is not somehow linked back to list “A”. See also “Part 4: Scripting Reference, 12.8.3 Linking”, p332.

S21 Save currents size

If the number of previous current items to save is greater than 4, then version tracking of the entry internal to PsyScope will not work properly; in particular, compiling StimList/EventList format scripts with optimization on will not necessarily produce correct results. See also “Part 4: Scripting Reference, 12.8.6 SaveCurrents”, p334.

S22 No #PsyScope

Any file that is to be used as a script must begin with `#PsyScope`. See “Part 4: Scripting Reference, 12.4.1 #PsyScope”, p324.

S23 File for ref not found

No file was found for the name given in `FileRef()`. See also “Part 4: Scripting Reference, 12.10 Using a File as an Entry”, p343.

S24 Wrong version

The number following `#PsyScope` at the beginning of this script is not the same as the current version of PsyScope. It is not required that the version number be correct; this is just a warning to let you know that something may be wrong. There may be an **Update** button in the warning dialog; **Update** does not attempt to correct your script for the new versions; it simply updates the version number at the top of the script.

S25 Extra “:”

An extra colon was found in an unexpected context. The script may be corrupted.

S26 Extra “>”

An extra greater-than sign was found in an unexpected context. The script may be corrupted, or the scripter may have unintentionally added too many `>`s when creating sub-attributes.

S27 More/Less weights than tokens

The number of weights that were specified for the list (in the “Weights” attribute) does not match the number of tokens that there are in the list. If there were too few weights, 1 will be assigned to the unweighted tokens. See also “Part 4: Scripting Reference, 12.8.4 Weights, Multiple, Grip”, p332.

S28 Zero total weight

All tokens in the list were assigned weight 0; this is equivalent to an empty list, which is illegal. The weights will be ignored. See also “Part 4: Scripting Reference, 12.8.4 Weights, Multiple, Grip”, p332.

S29 Error reading weights

There was a script error in reading the weights for a list from the “Weights” attribute. All tokens will be assigned a weight of 1. See also “Part 4: Scripting Reference, 12.8.4 Weights, Multiple, Grip”, p332.

S30 Token weight too large

The maximum weight that can be assigned to a token in a list is 255; this weight is the product of the individual weight in the “Weights” attribute, the “Mult” attribute factor, and the “Grip” attribute factor. See also “Part 4: Scripting Reference, 12.8.4 Weights, Multiple, Grip”, p332.

S31 Not enough tokens

In evaluating a script function, there were fewer tokens than expected by the function so that a parameter tag replacement failed. See also “Part 4: Scripting Reference, 12.9.6.3 Parameter Tags”, p342.

S32 Entry has no owner

`OWNER` was evaluated in the content of a global entry; `OWNER` can only be used in the content of attributes and inline entries. See also “Part 4: Scripting Reference, 12.2.4.1 THIS and OWNER”, p324.

S33 Can't set/add/delete token in fileRef/inline

File reference and inline entry contents can not be changed with operations that set, add, or delete token values. The error has occurred because a file reference or inline entry was used in such a way. See also “Part 4: Scripting Reference, 12.9.1 Inline Entries vs. Regular Entries”, p336.

S34 {} and “” in a string

Tokens in the script cannot contain both quotes and curly-braces because there is no way to quote both at the same time in a single string. See also “Part 4: Scripting Reference, 12.6.1 Literals”, p326.

S97 File errors

A file error was encountered which is outside the scope of the application.

S98 Memory errors

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

S99 Miscellaneous

These errors should be self-explanatory.

18.4 User Environment Errors

IN0 Memory low

When memory is uncomfortably low, you should close unused windows — especially Editor and graphic environment windows — and perhaps reset the experiment. When memory is critically low, you should save your work and exit PsyScope. To give PsyScope more memory, use **Get Info** in the Finder and increase PsyScope's memory partition.

IN1 Project could not be opened

The project does not exist, or there was some disk or file error in opening the project. Be sure that the project is not already open in another running copy of PsyScope. If the project continues to be unopenable, you may have to trash it and create a new project. See also “Part 2: Graphic Environment Reference, 6.1.1 Using Projects”, p213.

IN2 “Experiments” entry not found

Every script should have an “Experiments” entry with at least one experiment. See also “Part 4: Scripting Reference, 13.1.4 The ‘Experiments’ Entry”, p359.

IN3 Error opening Log File

The log file path is bad or there was some disk or file error in opening the log file; specify a new one. See also “Part 4: Scripting Reference, 16.5 Log File”, p458.

IN4 Settings file busy:

The preferences file (usually kept in the System folder) may only be opened by one copy of PsyScope. To correct the problem, close running copies of PsyScope, or put a copy of the preferences file in the same folder as PsyScope, and that copy will be used instead of the one in the System Folder. See also “Part 2: Graphic Environment Reference, 7.6 Options”, p265.

IN5 Entry not found

The reference given in the “Experiments” entry does not have an entry in the script. See also “Part 4: Scripting Reference, 13.1.4 The ‘Experiments’ Entry”, p359.

IN6 Range check failed

The entry for the dialog that was just closed has a “Range” attribute with a `False` value; presumably, the scripter wanted to alert you of some error in the entry’s information. See also “Part 4: Scripting Reference, 16.1.1.2 Range Checking”, p452.

IN7 External not found

No PsyScope Extension was found to handle a request. Probably, the extension name has been incorrectly typed, or you need to install an Extension in your “PsyScope Extensions” folder and restart PsyScope. See also “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

IN9 Unknown message

The specified extension does not know the message specified in the “Msg” attribute. See also “, Chapter 17. Dialog and Function Extensions”.

IN10 Menu entry deleted

The entry related to the menu item has been removed from the script or otherwise modified with the interactive editor; reinitializing will restore the menu item if it still exists. See also “Part 2: Graphic Environment Reference, 7.1.4 Utilities Menu”, p254.

IN11 Menu entry not found

No entry was found in the script to define the given menu item. Reinitializing may solve the problem. See also “Part 4: Scripting Reference, 16.1 Setting up the Menus”, p449 and “Part 2: Graphic Environment Reference, 7.1.4 Utilities Menu”, p254.

IN12 Script disabled

The reading of the script has been disabled in the **Tools** menu, presumably so that the interactive editor could be used without interruption. Re-enable the script by checking the **Script Enabled** menu item. See also “Part 2: Graphic Environment Reference, 7.3.1 Editor Menu Items”, p258.

IN13 Error loading experiment builder

There was a disk or memory error in trying to start up the appropriate extension to build the experiment. You may need to install an extension in you “PsyScope Extensions” folder. See also “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

IN18 Autoload not found

The autoload file that was set in the preferences cannot be found. Autoload can be turned off with the **No Autoload** checkbox in General Options (when no script or project is loaded). See also “Part 2: Graphic Environment Reference, 7.6.1 General Options”, p266.

IN19 Same name in project

A project cannot contain two scripts with the same name. Scripts that contain experiments having the same name are allowed, but only one of the experiments can be active (i.e. in the switching menu). See Part 3 for documentation on maintaining projects. See also “Part 2: Graphic Environment Reference, 6.1.1 Using Projects”, p213.

IN20 Bad XRES message

An error was encountered with an extension of type ‘XRES’. Either the wrong message was given in the “Msg” sub-attribute of an “ExtResult” attribute in a script function, or there is a problem with the extension (in which case you must consult the extension’s documentation).

IN22 Help wrong version

The wrong version of Help has been opened, an incorrect version of Help is in your “PsyScope Extensions” folder (even though the correct version may be there as well), or your Help file has been corrupted.

IN97 File error

A file error was encountered which is outside the scope of the application.

IN98 Memory errors

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

IN99 Miscellaneous

These messages should be self-explanatory.

18.5 Graphic Environment Errors

D0 Object list missing

Select **Check Links** from the **Design** menu, or open the Design window; then try again.

D1 Factor format only

Experiments can be defined using a variety of formats in the scripting language; only one format — Factor — is supported by the interface. See also “Part 4: Scripting Reference, 13.1.3 Script Formats”, p358.

D2 Check links warning

Removing the builder data loses some non-critical information: icon and window positioning, field type and custom attribute definitions, subject info type definitions.

D3 Bad name

This name may already be in use in the script, or it may contain special reserved keywords or symbols. See “Part 2: Graphic Environment Reference, 5.2.5 New Object Name Dialog”, p114.

D4 Entry not found

The entry does not exist in the script. This error must be the result of a scripting error or a corrupted file.

D5 Circular references

Blocks cannot own themselves, since this leads to a circular definition.

D6 Can't edit in trash

You must first remove the object from the trash (using the **Recover** button in the View Trash window) before you can edit it. See also “Part 2: Graphic Environment Reference, 5.2.9 View Trash Dialog”, p115.

D7 Unknown factor

In verifying the script, a reference was found in some object to a factor that either isn't connected to the object, or is not connected to all of the object's potential inheritance paths. You can ignore this and subsequent errors, or have the offending reference removed. See also “Part 2: Graphic Environment Reference, Linking Lists to the Hierarchy”, p135.

D8 Attribute name in use

Attributes must have unique names within a single object.

D9 Disappearing entry

An object has been deleted from the script — probably by direct editing — that some window had been using.

D10 Can't remove experiment

You cannot delete an experiment by deleting it in the graphic environment. There is no simple way to delete an experiment from the script.

D11 Too many utility submenus

The extensions list is used to build the text-inserting menus in the **Tools** menu; using a large number of extensions may cause the number of potential menus to exceed the number allowed, in which case the menu system is truncated.

D12 Bad Builder Data

If there is a scripting error in the “BuilderData” entry, the graphic environment cannot recover; you will have to edit the script in text form, or select **Check Links** from the **Design** menu with option held down to correct this problem.

D13 Already connected

You cannot multiply connect objects through the Design window. If you need to include an object in another multiple times, you must open the owning object's dialog and import the object to be multiply included.

D14 Can't preview

To preview a block or template, it must be connected to the main experiment structure.

D15 Linked to Trial End

Setting the duration of an event to **End of Trial** implies that nothing else will happen in the trial after that event ends, so you cannot link the start of another event to it's end.

D16 Delete all or one

When a custom attribute is added to a group, block, or template, it is also added to all objects of the same type in the hierarchy. When you delete a custom attribute, you can delete all of these, or just the one in the object you are editing,

D17 Can't edit as text

Certain attribute types cannot be edited in text form (by hitting Command-Right arrow) because their values are not simple (literal) tokens in the script.

D18 Value too large

To edit the attribute as text (by hitting Command-Right arrow), it must be less than 256 characters long.

D19 Attrib definition error

An error was encountered in reading the list of fields or custom attributes for an object as recorded in the "BuilderData" entry. This error is essentially non-correctable except by editing the script, but the data will be repaired when an attribute is added or deleted.

D20 No place to put a factor

Factors must be connected to the experiment, a group, a block, or a template. If you attempt to create a new list within an event or stimulus that is not connected to an object higher in the hierarchy, there will be no place for the list to be automatically connected.

D21 Standard attribute name

When creating custom attributes, it is best to avoid names of attributes that are used by stimuli, because you may inadvertently set an incorrect default value for some stimulus attribute.

D22 Bad condition list

The value of a conditions list has been corrupted in the script; deleting the current state lets you reset it to the correct value.

D23 Event doesn't accept type

Events which can contain sub-stimuli usually accept only certain types of stimuli. Consult the event type reference in “Part 2: Graphic Environment Reference, 5.8.7.3 Stimulus Attributes”, p181 or the manual for your custom extension.

D24 Error in action description

The value of a conditions-actions pairs list has been corrupted in the script; deleting the current state lets you reset it to the correct value.

D25 Can't modify factor table

Factors can be moved into or out of a factor table only through the Factor Table window.

D26 Already connected

A list cannot be both imported into a factor table and included in a separate list set of the object. To import the list, first remove the current link to the table object.

D27 Create nested?

Creating a new factor with the level of another factor selected lets you nest the new factor within the selected level.

D28 Subject Info setup

See “Part 2: Graphic Environment Reference, 6.2 Subject Info”, p224 for information on how the subject tracking is handled.

D29 Group already connected

You cannot include a group in an experiment multiple times.

D30 Info item does not exist

Certain Subject Info items must be defined in order to link indices to subject information. See “Part 2: Graphic Environment Reference, 6.2 Subject Info”, p224 for information on how the subject tracking is handled.

D31 Experiment has no groups

There must be groups defined in the experiment to perform certain types of links.

D32 No definition

The standard text field subject info type is always a plain text box; you cannot parameterize it more. See also “Part 2: Graphic Environment Reference, Define Subject Info Item Dialog”, p229.

D33 All logs together

Because of the way that the interface handles logging information, you must do all of the logging within a time segment together; you can rearrange the logs relative to each other however you want, but SubjectName should usually be the first Subject Info item logged in a set. See also “Part 2: Graphic Environment Reference, 6.2 Subject Info”, p224 and “Part 2: Graphic Environment Reference, 6.2.3.1 Logging and Scheduling Correctly”, p231.

D34 Modulo

Taking the modulo of a non-number will always return 0.

D35 Incompatible for connection

Two objects are incompatible if neither object is able to own the other. See “Part 2: Graphic Environment Reference, 5.2.1 Objects and the Experiment Hierarchy”, p107 for information on legal hierarchies.

D36 Lose information

Because factor table attribute dependencies are stored within the factor table object, converting a table to a template will lose all information about 1) factors defined in the table, and 2) how attributes of objects connected to the table vary based on the factors in the table.

D37 Transform tool

See “Part 2: Graphic Environment Reference, Template <-> Table Transform Tool”, p111 for information on using the transform tool.

D38 Circular link

Some part of the experiment hierarchy is incorrect, and a circularity was discovered at the link mentioned (though this may not be the incorrect link). The interface will remove the link for display purposes, but it will not attempt to correct the script; you must do this yourself using object dialogs, the scissors tool, or directly in the script using a text editor.

D39 Unknown object

An object was found in the scripted experiment hierarchy that the graphic environment was unable to identify.

D97 File error

A file error was encountered which is outside the scope of the application.

D98 Memory errors

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

D99 Miscellaneous

These messages should be self-explanatory.

18.6 Factor Format Errors

FACT1 No tag recorded

The tag name passed into a `GetTag()` call has not yet been used with a `SetTag()` call. See “13.3.6.4 Factor Format Tags”, p386 for information on using tags.

FACT2 Tag wrong format

The tag name that was passed into a `GetTag()` call was set in a different context than the current one; i.e., the `SetTag()` call was on a list of references (or tokens) and the `GetTag()` call was made for a list of tokens (or references). See “13.3.6.4 Factor Format Tags”, p386 for information on using tags.

FACT3 Error reading attribute

A script error occurred in trying to read the value of the attribute.

FACT4 Bad factor name

Somehow, a “Blocks” attribute — that was used in building the experiment structure — has disappeared.

FACT5 “Events” attribute missing

In following the experiment hierarchy to build a trial, no “Events” attribute was found. This may be because no events were added to the template, because some link higher in the hierarchy is missing, or because some branch of the experiment definition structure is incomplete.

FACT6 ODEV not found

An event has an event type that was unknown; this may be because the event type is bad, or because a special PsyScope extension needs to be placed in the “PsyScope Extensions” folder. See also “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

FACT7 Error getting factor name

The `FactorAttrib()` function — which implements **Vary by List** — must have at least one parameter, specifying the factor to link to. The name was either missing, or a script error blocked reading the name. See also “Part 2: Graphic Environment Reference, Vary by List”, p152 or “Part 4: Scripting Reference, 13.3.6.1 Factoring Event Attributes”, p384.

FACT8 Error getting attribute name

The attribute name parameter (the second one) of a `FactorAttrib()` call was a reference; it must be a string literal. See also “Part 4: Scripting Reference, 13.3.6.1 Factoring Event Attributes”, p384.

FACT9 Factor not found

The factor name passed into a `FactorAttrib()` call is the name of a non-existent or unconnected factor. See also “Part 2: Graphic Environment Reference, Linking Lists to the Hierarchy”, p135.

FACT10 Error getting tag name

The `SetTag()` and `GetTag()` functions must have at least one parameter, specifying the tag name. The name was either missing, or a script error blocked reading the name. See also “Part 4: Scripting Reference, 13.3.6.4 Factor Format Tags”, p386.

FACT11 Error getting attribute name

The `GroupAttrib()`, `BlockAttrib()`, `TrialAttrib()`, and `RunModeAttrib()` functions must have one parameter, specifying the name of the attribute to read as a string literal. The name was either missing, the first parameter was a direct reference, or a script error blocked reading the name. See also “Part 4: Scripting Reference, 13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.

FACT12 Attribute missing in group, block, or template

A `GroupAttrib()` (**Vary by Group**), `BlockAttrib()` (**Vary by Block**), or `TrialAttrib()` (**Vary by Template**) call failed because the specified attribute was not found in the current group, block, or template. For graphic environment users, the attribute should be defined and given a value as a custom group, block, or template attribute. See also “Part 2: Graphic Environment Reference, 5.8.2.2 Custom Attribute Sets”, p156 or “Part 4: Scripting Reference, 13.3.6.2 Linking Event Attributes to Template, Block, and Group Attributes”, p385.

FACT13 Attribute missing in RunMode

A `RunModeAttrib()` call — which implements **Vary by Run Mode**—failed because the specified attribute was not found in the Experiment. For scripters, the name of the attribute that is searched for is “*RunName*” in run mode or “*PracticeName*”, where *Name* is the parameter passed to `RunModeAttrib()`. For graphic environment users, the attribute should be defined and given a value as a custom run and practice attribute. See also “Part 2: Graphic Environment Reference, 5.8.3.1 Custom Run and Custom Practice Attributes”, p158 or “Part 4: Scripting Reference, 13.3.6.3 Linking Event Attributes to the Run Mode”, p385.

FACT14 Unknown Latin square factor

A factor that was specified in the “*Set_LatinSets*” attribute was not a factor in that set, where *Set* is the name of the set. The named factor may not exist, or it may be in a different factor set. See also “Part 4: Scripting Reference, Scripting Latin Square Partitions”, p390.

FACT15 Error getting level reference

A script error was encountered in getting a reference to the current level of a (non-list) factor.

FACT16 List factor attribute not found

The attribute (or field) referenced by a `FactorAttrib()` call — which implements **Vary by List** — was not found as an attribute (field) of the factor. See also “Part 2: Graphic Environment Reference, Vary by List”, p152 or “Part 4: Scripting Reference, 13.3.6.1 Factoring Event Attributes”, p384.

FACT17 Fee level attribute not found

The attribute (or field) referenced by a `FactorAttrib()` call — which implements **Vary by List** — was not found as an attribute (field) of the current level. See also “Part 2: Graphic Environment Reference, Vary by List”, p152 or “Part 4: Scripting Reference, 13.3.6.1 Factoring Event Attributes”, p384.

FACT18 Error getting multiple

A `FactorAttrib()` call — which implements **Vary by List** — was made to a list factor attribute with multiple-token values, but there was a script error obtaining a reference in the attribute parallel to the current level. Any list factor attribute with a “Multiple” sub-attribute must have references as its values, with each set of tokens for a value at the reference; list files with multiple-token values must include a `#NonLiterals` identifier at the beginning of the file, and values in that column but be surrounded by an extra set of square brackets. See also “Part 4: Scripting Reference, 13.3.7 Scripting Factors”, p387, “Part 4: Scripting Reference, 12.10 Using a File as an Entry”, p343, or “Part 2: Graphic Environment Reference, List Files with Non-literals”, p131.

18.7 Trial Manager Errors

TM01 Name is not an event name in trial *n*

In defining trial *n*, the script used the name given someplace where an event name is needed, and that name is not the name of an event in that trial. The name is most likely used in a “StartRef” attribute or as an action parameter.

TM03 Event number *n* does not exist in trial *m*

In defining trial *m*, the script refers to event number *n*, which doesn’t exist. The number is most likely used in a “StartRef” attribute or as an action parameter.

TM04 Duration for event is not valid. Will default to 0 msec.

A syntax error of some sort caused the duration given for the listed event not to be recognized. It will default to 0 milliseconds unless the run is cancelled.

TM11 Event list for trial is bad; No events depend on the START event (event 0)

Every trial must have at least one event scheduled relative to the beginning of the trial (the start event). If every event is unscheduled or scheduled to start at some time relative to another event in the trial, the Trial Manager will not know which one to run first, and will give this error.

TM12 No expression given for TIME duration in event

The `Time[]` duration specifier takes a variable expression as its argument. This could be a constant (e.g. `Time[500]`), a single variable (e.g. `Time[X]`), or a combination of variables (e.g. `Time[{X+LastRT}]`); none of these were specified.

TM13 Unable to parse StartRef

The start reference that was given had some sort of syntax error and was unreadable by the Trial Manager. See also “Part 4: Scripting Reference, 13.4.2 Start Reference”, p411.

TM14 Start Reference doesn't specify start or end of an event. End will be used.

Start references should specify whether the delay given is relative to the beginning or end of the event referred to. This warning is given when neither start nor end is specified. The trial manager will assume end. See also “Part 4: Scripting Reference, 13.4.2 Start Reference”, p411.

- TM15 There is no event n in trial m (referenced in startref s)**
A numeric reference to event n was given in a start reference for some other event, and event n does not exist for that trial.
- TM16 Invalid name ... (referenced in StartRef ...)**
The name given is not a valid name for use in a start reference. All names used in start references must be names of events in the current trial. Keywords such as `TRIAL_END`, `LAST`, etc. are not valid. See also “Part 4: Scripting Reference, 13.4.2 Start Reference”, p411.
- TM30 ... is not a valid action name**
The name given is not the name of an action available in the system. Check the spelling of the action name. Also, if the action is one that is provided by a PsyScope Extension, check to make sure that the extension is in the “PsyScope Extensions” folder. See also “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.
- TM32 ... too many params specified for action "..."**
More than the maximum number of parameters were specified for the given action. See also “Part 4: Scripting Reference, 14.1 Actions Reference”, p419 or the documentation for PsyScope Extensions that you are using.
- TM33 FORCE_ALL and instances<0 => infinite loop in action"..."**
Setting the number of instances for an action to a negative value indicates an infinite number of instances. This, in combination with active until **All Instances** (`FORCE_ALL`), will cause an infinite loop. If an action with these parameters is put up, the trial can only be ended by an explicit call to `QuitTrial[]`. See also “Part 2: Graphic Environment Reference, 5.9 Conditions and Actions”, p192 or “Part 4: Scripting Reference, 13.4.1.2 Instances and ActiveUntil”, p410.
- TM34 ... in "... " should be an integer**
An action required an integer parameter, but the value given wasn't an integer. See also “Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions”, p198, “Part 4: Scripting Reference, 14.1 Actions Reference”, p419, or the documentation for PsyScope Extensions that you are using.
- TM35 Incorrect number of parameters for action ...**
The wrong number of parameters were given for an action. See also “Part 4: Scripting Reference, 14.1 Actions Reference”, p419 or the documentation for PsyScope Extensions that you are using.
- TM36 RT referenced event ... that has not yet occurred**
The `RT[]` action must record response times relative to the actual start time of an event. For this to occur, the referenced (relative) event must have already begun at the time that the `RT[]` action is triggered. By default, this will be the case. However, if an event is specified as the **Relative To** parameter of `RT[]`, and that event has not started at the time that `RT[]` triggers, then this error will occur.

TM37 Neither action nor source event specified for CancelAction

CancelAction[] requires that you give either the name of an action to cancel, or the name of a event whose actions to cancel, or both. Neither were specified.

TM38 Event ... is already running Start action(s) not cancelled

TM39 Event ... has already ended End action(s) not cancelled

CancelAction[] was unable to cancel the start or end actions for the given event because that event has already started or ended.

TM40 Call to ... requires an event reference as its first parameter

A number of actions that involve the running or scheduling of events require an event to be specified as their first parameter. This parameter cannot be defaulted and must be specified. See also “Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions”, p198 or “Part 4: Scripting Reference, 14.1 Actions Reference”, p419.

TM41 RT: VAR_ONLY specified, but no variable given. Data won't be stored.

The VAR_ONLY flag was specified (which tells the Trial Manager to record response information only in a variable, and not in the data file), but no variable was specified to store the data, so it won't be stored. See also “Part 2: Graphic Environment Reference, RT[]”, p203 or “Part 4: Scripting Reference, 14.1.1.4 Miscellaneous Actions”, p421.

TM42 Timeout trying to play stimulus.

In playing asynchronous stimuli, such as sounds, the Trial Manager gives the output device a maximum of 100ms to begin playing the stimulus. This error most likely indicates a low-level problem with the output device in question.

TM43 Too few parameters for n given, m required.

The named action requires at least m parameters but only n were given. See also “Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions”, p198, “Part 4: Scripting Reference, 14.1 Actions Reference”, p419, or the documentation for PsyScope Extensions that you are using.

TM44 ChanceEvent probability ... is not in the range [0.0,1.0].

The ChanceEvent[] action takes a probability between 0 and 1, inclusive. The probability given is not in that range. See “Part 2: Graphic Environment Reference, ChanceEvent[]”, p200 or “Part 4: Scripting Reference, 14.1.1.2 Event Scheduling Actions”, p419.

TM45 No list specified for ... action.

TM46 Argument is not a list in ... action.

The action named requires a Trial Manger list variable as a parameter, and that parameter was not specified, or was not a list. See also “Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions”, p198, “Part 4: Scripting Reference, 14.1 Actions Reference”, p419, or the documentation for PsyScope Extensions that you are using.

TM47 Index argument is not a number in ... action.

The action required an index to a list or an array as one of its arguments, and the variable expression given did not give a number as its result. See also “Part 4: Scripting Reference, 14.1.1.5 Trial Variable Actions”, p422.

TM48 Types don't match in ... action.

The action given requires two Trial Manager Variables of the same type as parameters (most likely to assign one to the value of the other). The parameters given weren't of matching types. See also “Part 4: Scripting Reference, 14.1.1.5 Trial Variable Actions”, p422.

TM65 Stimulus will not be seen: it will be cleared after duration = 0

An event duration has been set to 0 milliseconds. The Trial Manager will not display the stimulus. See also “Part 2: Graphic Environment Reference, Duration”, p179 or “Part 4: Scripting Reference, 13.4.3 Duration”, p411.

TM68 Unable to open resource file ...

A resource file name was specified in the **Resources** (“Resources”) experiment attribute, but that file could not be found or for some reason could not be opened by PsyScope. Check to make sure that the file is available and in a folder that PsyScope has access to. See also “Part 2: Graphic Environment Reference, Resources”, p167 and “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216.

TM69 Unable to open ...

PsyScope could not find the named file or for some reason was unable to open it.

TM70 Not enough memory available for Instructions or Debriefing

There is not enough memory available for PsyScope to set up the structures necessary for displaying an instruction or debriefing file.

TM98 Internal error — Program error

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

TM100 Referenced variable is not an array or list in: ...

A variable expression was set up using the `.` operator — which is used to index an array or list — but the variable being indexed (i.e. the variable to the left of the dot) is not an array or a list. See also “Part 4: Scripting Reference, 13.5.3 Variable Expression Syntax”, p417.

TM101 Error parsing expression: ...

There was a syntax error in the Trial Manager Variable expression shown. See also “Part 4: Scripting Reference, 13.5.3 Variable Expression Syntax”, p417.

TM102 No such variable ...

The variable shown was used in a variable expression, but is not defined as a Trial Manager variable in the current script. See also “Part 4: Scripting Reference, 13.5.1 Declaring Variables”, p412.

TM103 Invalid field selector in expression. ...

A keyword given to select a field of a record type variable was not the name of any of the fields of that record. "->" is the selection operator. The variable to the left of the arrow is the record, and the name to the right of the arrow is the name of the field to be selected. See also "Part 4: Scripting Reference, 13.5.3 Variable Expression Syntax", p417.

TM105 Selection variable is not a record in: ...

In the expression shown, an attempt is made to select a field (using the selection operator "->") of a variable that is not a record, and thus has no fields. See also "Part 4: Scripting Reference, 13.5.3 Variable Expression Syntax", p417.

TM106 Unmatched parenthesis in expression: ...

The variable expression contains one or more parentheses with no matching counterpart. See also "Part 4: Scripting Reference, 13.5.3 Variable Expression Syntax", p417.

TM107 Cannot reset a constant.

A constant value (e.g 5, 2.03) was used in a way that would require that its value be changed. For example, the action `Set[5 {x+3}]`, would require that 5 be changed to the value of `{x+3}`; this is illegal.

TM108 Cannot reset read only variable.

Most built in variables, such as `TrialNum`, are read-only variables; their values cannot be changed by `Set[]` or other actions.

TM109 Can't change ... variable "... " to type ...

In the process of evaluating a variable expression, or executing an action, the Trial Manager was asked to convert the named variable from the first type to the second and was unable to do so. Only variables of types `Integer`, `Long_Integer`, and `Float` can be converted.

TM110 Can't convert unassigned variable.

An expression contained a variable that did not yet have a type or value assigned to it, and the trial manager was asked to convert it to another type but was unable to.

TM111 Can't convert to null type.

In evaluating an expression, the Trial Manager was asked to convert the type of a value to `NULL`, and was unable to.

TM112 Type conversion error.

The Trial Manager was unable to do a type conversion, because either the source type or destination type was not one of `Integer`, `Long_Integer`, or `Float`.

TM113 Unable to read "... " into ... variable

While initializing variables, a variable definition was encountered that required that the quoted string be read into the named variable, but the Trial Manager was unable to read the string as an appropriate value for the that variable.

TM114 Unable to convert value to string

The Trial Manager was unable to write out the value of a variable to the script or data file. Only variables of type `Integer`, `Long_Integer`, and `Float` can be written to the data file.

TM130 Variable ... can't be written to the data file because it doesn't exist.

One of the variables listed in the “DataVariables” experiment attribute has not been declared. See also “Part 4: Scripting Reference, 13.5.1 Declaring Variables”, p412.

TM200 Unable to find input device ... Continue without it?

The named input device — listed in **Input Devices** (“InputDevices”) — could not be found; possibly, it was misspelled in the script, or the extension file for that device is not in the PsyScope Extensions folder. Clicking **OK** will run the script, ignoring references to the missing device. See also “Part 2: Graphic Environment Reference, Input Devices”, p164 or “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360; also, “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

TM201 Cannot connect to ... It doesn't look like an input device...

An extension was found by the given name, but it doesn't behave like an input device and cannot be used; clicking **OK** will run the script, ignoring all references to the device in question.

TM202 Unable to connect IDEV: ...

An error occurred while trying to connect to the named input device, and no more information is available. This is very likely an internal error of some sort in the extension for that ‘IDEV’.

TM260 No timer specified, Macintosh timer will be used.

No timer was specified in the **Timer** (“Timer”) experiment attribute. Clicking **OK** will cause the default Macintosh internal timer to be used instead. See also “Part 2: Graphic Environment Reference, Timer”, p164 or “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360.

TM260 Timer ... not found.

The timer named in the **Timer** (“Timer”) experiment attribute was not found. Make sure the name is spelled correctly, and, if necessary, the extension file for that timer is included in the “PsyScope Extensions” folder. See also “Part 2: Graphic Environment Reference, Timer”, p164 or “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360; also, “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

18.8 Screen Manager Errors

SCR01 Unable to find file: ...

The named file could not be found. Make sure it's in the search area set for the script. See “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215 regarding finding files.

SCR02 Unable to find PICT resource or file ...

The Screen Manager was trying to find a PICT resource or file and could not. The name given could be either the name of a resource or the name of a file. If it was intended to be a resource, make sure that the resource file is listed in the **Resources** (“Resources”) experiment attribute, or is in the “PsyScope Extensions” folder. If it was intended to be a file, make sure that the file is somewhere where pscope will find it. See “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215, “Part 2: Graphic Environment Reference, Resources”, p167 or “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360, and “Part 2: Graphic Environment Reference, 6.1.3.1 PsyScope Extensions”, p216.

SCR03 ... is not a valid depth for an offscreen bitMap or pixmap.

The depth specified for a **Pasteboard** or **PICT** stimulus — in the **Depth** (“Depth”/“PBoardDepth”) attribute — was not a valid bit depth. Valid depths are 1, 2, 4, 8, 16, and 24, and are limited by the maximum depth of the deepest screen on the current machine. See also “Part 2: Graphic Environment Reference, PICT Attributes”, p182 and “Part 2: Graphic Environment Reference, Pasteboard Attributes”, p183 or “Part 4: Scripting Reference, 14.2.5.1 PICT Attributes”, p433 and “Part 4: Scripting Reference, 14.2.6.1 Pasteboard Attributes”, p434.

SCR05 Not enough available memory to keep PICT ... in memory.

There’s not enough memory available to use the **Keep picture in memory** (**KEEP_PICT**) flag with the current script. If **Keep picture in memory** is absolutely necessary, try to increase the memory partition for PsyScope (using **Get Info** in Finder). See “Part 2: Graphic Environment Reference, 6.5 Space and Speed”, p246 regarding optimizing memory usage. See also “Part 2: Graphic Environment Reference, PICT Attributes”, p182 or “Part 4: Scripting Reference, 14.2.5.1 PICT Attributes”, p433.

SCR06 Error reading PICT ...

There was a disk related error reading the PICT. There is some problem with the PICT file.

SCR08 Not enough available memory to save screen

Use the **Don’t save screens** (**NO_SAVE_SCREEN**) experiment flag in the **Special** (“Flags”) experiment attribute to prevent the Screen Manager from automatically saving the screen on breaks. See “Part 2: Graphic Environment Reference, Special”, p165 or “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360. See also “Part 2: Graphic Environment Reference, 6.5 Space and Speed”, p246 regarding optimizing memory usage.

SCR12 Unable to read Origin attribute.

The “Origin” attribute was not specified correctly. See “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360.

SCR13 You did not specify the proper number of monitors...

A custom monitor ordering was specified in the “MonitorOrder” attribute, but too few or too many monitors were given. See also “Part 4: Scripting Reference, 13.1.6.1 Standard Experiment Attributes”, p360.

SCR20 Unable to open document ...

There was an operating system level error while trying to open the named document file. Make sure the file exists and that it has not been corrupted.

SCR21 ... in ... should be an integer.

A screen action required a parameter to be an integer, but the parameter that was given couldn't be read as an integer. See also "Part 2: Graphic Environment Reference, 5.9.4.1 Available Actions", p198 or "Part 4: Scripting Reference, 14.2.1.3 Text and Screen Actions", p430.

SCR22 ... event type takes only one stimulus.

The named screen event type takes only a single stimulus, but multiple stimuli were specified (in the "Stimulus" or "Stimuli" attribute) for a single event.

SCR23 No stimulus specified for ... type event.

No stimulus was specified for some event; specify a stimulus in the event's **Stimulus** (**Text** event), **Picture** (**PICT** event), **File** (**Document** event), **Paragraph** (**Paragraph** event), **Prompt** (**KeySequence** event) or **Stimuli** (**Pasteboard** event) stimulus attribute (all "Stimulus" or "Stimuli" in PsyScript). See also "Part 2: Graphic Environment Reference, 5.8.7.3 Stimulus Attributes", p181 or "Part 4: Scripting Reference, 14.2 Stimulus Types Reference", p424.

SCR24 Unable to determine PasteBoard sub-stimulus type.

The type of a sub-stimulus for a **Pasteboard** event was ambiguously defined. Neither a "StimType" attribute nor an "EventType" attribute for the given stimulus could be found. See also "Part 4: Scripting Reference, 14.2.6 Pasteboard", p434.

SCR25 Incompatible Run File Version

The run file being loaded is not compatible with the current version of the Screen 'ODEV'.

SCR26 Error opening ShowDoc file ...

The **ShowDoc**[] action was unable to find the document to show, or there was an error loading the document. Make sure that the document is in the right place and that it is intact and has not been corrupted. See also "Part 2: Graphic Environment Reference, 6.1.2 Path Names", p215.

SCR27 Not enough memory available to keep document ... in memory.

There is not enough memory available to load the document; increase the memory partition size for PsyScope (using **Get Info** in Finder) or decrease memory usage. See also "Part 2: Graphic Environment Reference, 6.5 Space and Speed", p246 regarding optimizing memory usage.

SCR28 Error allocating TextEdit Record for Document or Paragraph.

There was an error allocating structures necessary to load a **Document** or **Paragraph** stimulus. Most likely there was not enough memory available. See "Part 2: Graphic Environment Reference, 6.5 Space and Speed", p246 about optimizing memory usage.

SCR98 Internal error

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

18.9 Sound Manager Errors

SND01 Sound has no SoundFile attribute

No file was specified from which to extract the sound to be played. Set the file in the **File** (“SoundFile”) stimulus attribute. See also “Part 2: Graphic Environment Reference, Sound Attributes”, p184 or “Part 4: Scripting Reference, 14.2.7.1 SoundLabel Attributes”, p435.

SND02 Unable to open sound file: ...

There was a operating system level error while trying to open the named document file. Make sure the file exists and that it has not been corrupted.

SND03 Unable to find sound file: ...

PsyScope couldn't find the sound file. Make sure it exists, it's name is spelled correctly in the script, and it is in a folder where PsyScope will find it. See also “Part 2: Graphic Environment Reference, 6.1.2 Path Names”, p215.

SND04 Error getting sound label ... from file ...

There was an error trying to get the labelled sound from the sound file. Most likely there is no label by that name in the file.

SND05 Load Error: Unable to read sound ... from file ...

SND06 Load Error: Unable to get ... resource from ...

The sound file that was specified to contain the sound is corrupt, and the sound could not be read from it. Restore the file from backup or recreate it.

SND07 Load Error: Unable to allocate memory for sound ...

There's not enough memory available to load the sound. See “Part 2: Graphic Environment Reference, 6.5 Space and Speed”, p246 for more on optimizing memory usage.

SND08 Sound type ... not supported.

The sound file was of a type not supported by PsyScope. SoundEdit™ and Sound-Designer II™ are the formats currently supported. See also “Part 2: Graphic Environment Reference, Sound Event Type”, p179.

SND09 Error ... getting file info for ...

There was a operating system error trying to get information about the sound file. Make sure the file exists and can be found.

SND10 Unable to initialize sound channel, error ... while loading ... from file ...

A sound was specified to be played with the **Play in parallel** (`PARALLEL`) feature in the **Special** (“Feature”) attribute, but a new sound channel could not be set up for it. Most likely there are already too many channels open. Each new sound to be

played in parallel requires a new channel. The maximum number of open channels possible varies depending on machine configuration and system resources available. See also “Part 2: Graphic Environment Reference, Sound Attributes”, p184 or “Part 4: Scripting Reference, 14.2.7.1 SoundLabel Attributes”, p435.

SND11 System 7 is required to play Sound Designer Files

Macintosh System 7.0 or higher is required to play Digidesign™ Sound Designer II sound files.

SND12 Unable to allocate sound channel. Error: ...

SND13 Unable to initialize sound channel. Error: ...

There was a problem allocating or initializing the default sound channel while initializing the PsyScope sound device. Most likely all sound resources are being used by another application, or there’s not enough memory available to open a new sound channel. It’s possible that a crash of an application that uses sound (including a previous crash of PsyScope) could have left sound channels allocated, in which case a reboot is necessary. See “Part 2: Graphic Environment Reference, 6.5 Space and Speed”, p246 for more on optimizing memory usage.

SND14 SoundLabel event type takes only one stimulus.

More than one stimulus was specified for a SoundLabel event type in the “Stimulus” attribute; only one is allowed. See also “Part 4: Scripting Reference, 14.2.7.1 SoundLabel Attributes”, p435.

SND15 Incompatible Run File Version

The version of the run file is incompatible with the current version of PsyScope. The run file will have to be recreated.

SND98 Internal Error

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.

18.10 Button Box Errors

BB10 ExInit: Button box disconnected

The button box driver was unable to establish communications with the button box. Check for the following causes: The button box is turned off; the button box is not connected; the button box is connected to the wrong port; the power supply for the button box is not connected, or the button box is malfunctioning.

BB11 ExInit: Timer inaccurate: ...

The button box driver’s internal sanity check for the button box timer failed. The number given is the number of button box milliseconds counted in one Macintosh second. (Note: on some Macintosh models, this check fails spuriously, giving very small deviations, e.g. 997 or 1002, subsequent attempts to run should produce correct results. System 7.1 with the hardware system upgrade is expected to remedy this problem)

BB98 Internal error.

There is a bug in PsyScope or some other program interfered with PsyScope. Please send a bug report to PsyBug@serviceberry.psy.cmu.edu.



Chapter 19. Configuring the Button Box

In order to use the button box with PsyScope, you need:

- A button box.
- A cable to connect the button box to the modem or printer port of a Macintosh™ computer. This cable is packaged with the button box.
- The PsyScope software.
- A 6V, 150mA power supply; this plugs into a 5.0-5.5 mm jack in the back of the button box. The outside of the plug is positive, and the inside is negative.

Note: The standard U.S. 120V power supplies are not shipped abroad, since they will not work there. If we did not ship you a power supply, you need to purchase an appropriate AC to DC adapter.

To set up the button box:

1. Turn your machine off.
2. Unplug any devices plugged into the modem port (or printer port).
3. Connect the button box to the modem port (or printer port) on the back of your computer with the special cable provided. (The modem port has a telephone icon above it; the printer port has a printer icon.) The little arrow on the computer end of the cable should be facing up.
4. Hook up an AC adapter to the button box and plug it into the power outlet.
5. Turn on the button box. The switch is on the back and “up” is “on”.
6. Turn on the computer.

Whenever you start up a new experimental session, you will probably want to check the button box to make sure that it is functioning properly. In order to do this, select **Test Button Box** from PsyScope’s **Experiment** menu. This will bring up the dialog shown below.

*Note: If there is no **Test Button Box** item in the menu, you can evaluate `Run("Test BBox")` in the Evaluator.*

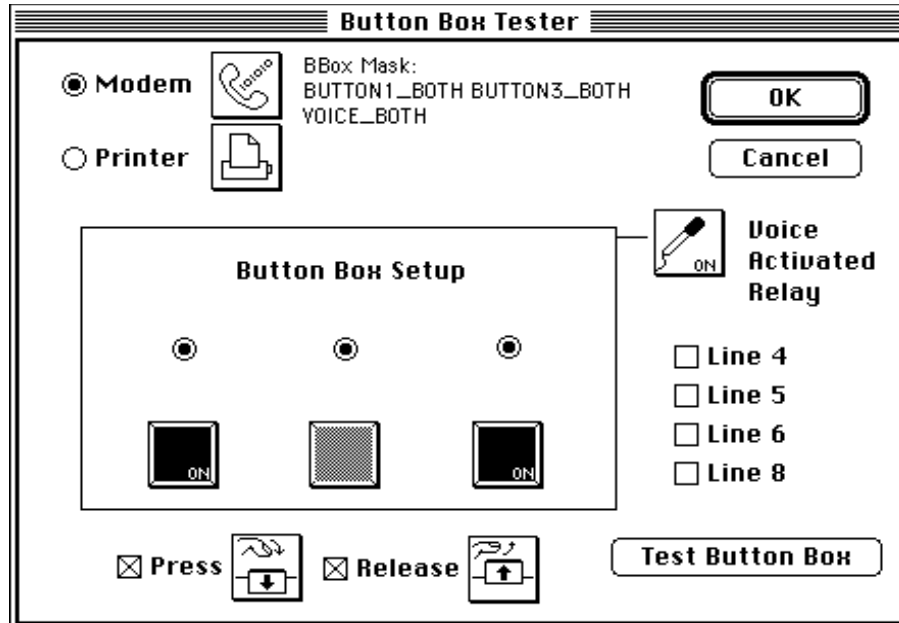


Figure 205 – The Button Box Testing dialog

Within this dialog box, there are a few choices for you to make:

Depending on where the button box cable is connected, you need to select **Modem** or **Printer**. It is recommended to use the modem port, unless there is some specific reason to use the printer port.

If you need to use the built-in voice activated relay, you should press the button labeled **Voice Activated Relay** and it will say **ON**. Otherwise this button should say **OFF**.

Click on the drawing of each button in the middle of the dialog to turn them on or off. Buttons which are on are hilited and say **ON**; buttons which are off are grayed out. (in the figure, the center button is turned off).

For testing, you can ignore the **Press** and **Release** buttons at the bottom of the dialog.

After you have finished making these settings, you should test the box. You can do this in four steps:

First you should click on **Test Button Box**. This will send out initialization signals to the button box; depending on the response it receives, the dialog will tell you if the button box is connected properly. A message dialog will notify you if the connection test fails. In this case, check all cables and make sure the switch on the back of the button box is turned on.

To test the functioning of individual buttons, press them on the real button box and their icons on the Macintosh should hilite.

You can test the functioning of the LED's above the buttons by clicking on their icons (in the middle of the dialog, above the buttons) to turn toggle them on and off.

You can test the voice activated relay by talking into it with a microphone. If it is working, the microphone icon will hilite. You may need to adjust the sensitivity of the microphone by turning the knob on the back of the button box.

Press **Return to Setup** after testing is completed.

If you want to send an electrical signal through the output port of the button box to control another device, you should look at the section of the button box manual that describes the control of the output port and you should look in this manual for material on the BBoxOut action.

Ψ

Chapter 20. Creating Picture Resources

PsyScope can present picture stimuli that are save either as standard PICT files, or as PICT resources. You can create picture stimuli with virtually any application you want; you can then save the drawing to a PICT file, or you can put a number of pictures together in a single resource file.

To save a drawing as a PICT file, you must use **Save As...** in the drawing program. Most programs will let you select the standard PICT format in the saving dialog.

This rest of the chapter will explain how to create resource files containing a number of pictures. In addition to a drawing program, you will need ResEdit™, a standard utility available from Apple®.

First, start up a drawing application and create the picture. Make sure the illustrations are about the same size and that they will fit on the screen. Then, select the picture. If it is the only thing in the document, type Command-A (or choose **Select All** from the **Edit** menu). Otherwise, use the selection tool shown in the illustration below:

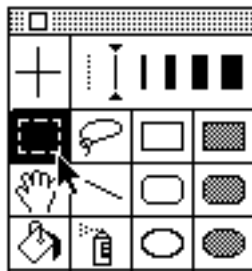


Figure 206 – A Tool palette

Type Command-C (or select **Copy** from the **Edit** menu) to copy the picture from the document.

Now, start up ResEdit™. Select **New** from the **File** menu to create a new picture resource file, or open an existing one using **Open**. Paste the picture (using Command-V) into the resource window. The word **PICT** in the window indicates that you created a picture resource.

Double-clicking on **PICT** opens a new window which should show your illustration; all that remains to be done is to give it a name that you will use in the script. Select the picture, and

type Command-I (or select **Get Info** from the **File** menu). Fill in the proper name once the following window appears:

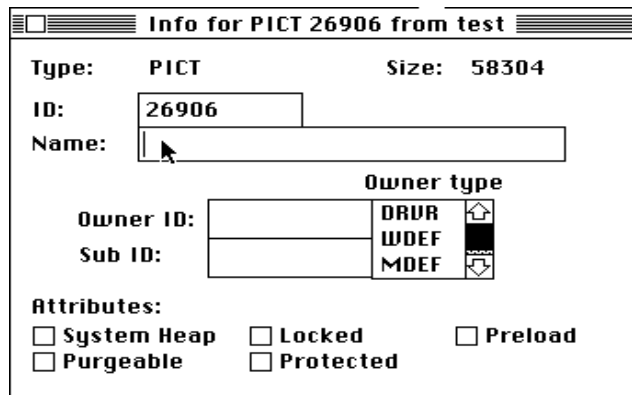


Figure 207 – The Set Info dialog in ResEdit

Close the info box and choose **Save** from the **File** menu.

You may have as many pictures in one resource file as you need. By pasting into an existing SoundEdit™ document, sound and picture resources can be combined into one file.

To use the pictures in PsyScope, you will need to tell PsyScope to open the resource file. See “Part 2: Graphic Environment Reference, 6.1.3 Resources”, p216.

You may find that it is best to keep your sounds and pictures in separate files. Unfortunately, there is no method for setting folders access inside the GUI. However, you can go inside the script and modify it using the strcat function in the following way:

```
PictureList:: dog cat
```

```
SoundList:: woof meow
```

```
PictureEvent:
```

```
Stimulus: Strcat(":stimuli:pictures:", FactorAttrib(PictureList))
```

```
SoundEvent:
```

```
Stimulus: Strcat(":stimuli:sounds:",FactorAttrib(SoundList))
```

Here there is a folder called stimuli and within that there is a folder for sounds and another for pictures.



Chapter 21. Creating SoundEdit™ Sound Files

Use the SoundEdit™ application from Farallon's MacRecorder to create sound files. Follow the instructions in the MacRecorder manual to record samples, and pay particular attention to sections concerned with how to “label” a segment or all of a sound sample. Under **Recording Options** set the sample rate to 22 kHz. Always use monaural recording with no compression.

The best way of recording stimuli for most experiments is to read all the sounds into a single SoundEdit file in one pass. Use a high quality microphone with a long cord connected to the MacRecorder. Make sure that you are in a quiet room and try to stand as far away from the Macintosh as you can. If you have a separate hard disk and can run SoundEdit without it, you could turn off the hard disk to cut down on noise. Try to set the recording level as high as you can without leading to any distortion. It is quite important to choose a high recording level, since a low recording level will yield a low single-to-noise ratio.

After you have completed the recording, use the MacRecorder playback function to locate the beginnings and ends of individual sounds. Look for quiet “zero” level spots to begin and end labelled sections of sound. You may convert the material between lexical items to silences using the **Silence** item in the **Effects** menu. Make sure that at least about 1 msec— about 22 samples — of silence is present at the beginning and end of each lexical item, so that there will not be any rapid transitions leading to bursts when the words are strung together.

After you have surrounded a lexical item with a bit of silence, hilite just this section of sound including the minimum 22 samples of silence at each end; give it a label — which will be used as the stimulus in PsyScope — by simply typing the name. Be careful to write the label name exactly (capitalization is not important). Avoid using spaces and punctuation; use the _ instead of a space if you need to.

A label is limited to 30 characters. If you need to edit a label name, click on the label and choose **Edit Label** from the **SoundEdit** menu to make your corrections. Be sure to use **Save As** option from **File** menu to save your editing of labels.

Note: Apparently, there is a bug in SoundEdit such that it does not consider label editing as something that has changed its file; thus, editing a few labels and quitting will lose all your labelling work without the usual “Do you want to save?” warning.

You can use the Slice n' Splice application written by Darius Clynes to break up the large file into many little files. Do this by setting up a folder with only the large file in it. The Slicer will remove extra silence during the process of creating the little files. You can then

correct particular lexical items or add new ones. Once you have a finished set of item prepared, put them back together into one large file using the Splicer in Slice n' Splice. You can also use the Slicer to break up large files you need to copy to diskettes.

Symbols

- ! (PsyScript) 351
 - ! (trial variable operator) 418
 - != (PsyScript) 351, 352
 - # (PsyScript) 324
 - #> ExperimentDefinitions, etc. markers 282
 - #include modifier 325
 - #inherit modifier 325
 - #NoIncludeStdLib modifier 326
 - #noinherit modifier 325
 - #NonLiteral 131, 343
 - #PsyScope 281
 - #PsyScope modifier 324
 - #winclude modifier 325
 - \$ (PsyScript) 352
 - \$- (PsyScript) 352
 - \$+ (PsyScript) 352
 - % (PsyScript) 351
 - % (trial variable operator) 418
 - %= (PsyScript) 351
 - && (PsyScript) 351
 - && (trial variable operator) 418
 - (), see *parentheses*
 - * (PsyScript) 351
 - * (trial variable operator) 418
 - *! (PsyScript) 354
 - ** (PsyScript) 354
 - *, see *asterisk*
 - *= (PsyScript) 351
 - + (PsyScript) 351
 - + (trial variable operator) 418
 - += (PsyScript) 351
 - (PsyScript) 351
 - (trial variable operator) 418
 - \$ (PsyScript) 352
 - = (PsyScript) 351
 - > (PsyScript) 323, 350
 - > (trial variable operator) 418
 - .(PsyScript) 350
 - .(trial variable operator) 418
 - . operator 330
 - .. (PsyScript) 354
 - / (PsyScript) 351
 - // (PsyScript) 351
 - /= (PsyScript) 351
 - :: pop-up menu 261
 - < (PsyScript) 351, 352
 - < (trial variable operator) 418
 - <= (PsyScript) 351, 352
 - <= (trial variable operator) 418
 - = (PsyScript) 354
 - = (trial variable operator) 418
 - == (PsyScript) 351, 352
 - == (trial variable operator) 418
 - => (PsyScript) 409
 - > (PsyScript) 351, 352
 - > (trial variable operator) 418
 - >= (PsyScript) 351, 352
 - >= (trial variable operator) 418
 - >> (PsyScript) 323
 - ? (PsyScript) 352, 353
 - ??? 126
 - @ (PsyScript) 350
 - @.:, see *references* in *attribute blocks*
 - [], see *square bracket*
 - ^ (PsyScript) 350
 - “_LatinSets” sub-attribute 390
 - ` (PsyScript) 342
 - {}, see *curly braces*
 - | (PsyScript) 353
 - | (trial variable operator) 418
 - || (PsyScript) 351
 - || (trial variable operator) 418
 - ~ (PsyScript) 354
 - ◇ 41
- ## A
- AbortEvent[] action 198
 - scripting 419
 - access types 133, 141
 - Blocked Random 141
 - By Factor 141
 - Cycle Random 141
 - Least-Used Random 141
 - scripting 391
 - access types (PsyScript lists) 331
 - Access() function 330, 352
 - AccessAll() function 352
 - AccessSome() function 353
 - “AccessType” attribute 331, 392
 - “AccessTypes” sub-attribute 391
 - Action bar (of the editor) 261
 - actions 30, 192
 - active 192
 - active until 33, 192
 - scripting 410
 - available 198
 - based on event/trial start or end 197
 - cancelling 199
 - instances 33, 192
 - scripting 410
 - maximum instances, see *instances*
 - of events, see *events*
 - of trials, see *trials*
 - posted 192
 - running 444
 - run-time information 243
 - scripting 290, 409
 - for trials 295
 - setting beep for key press 31
 - Actions event attribute, see *attributes* under *events*
 - Actions List dialog 198
 - Actions submenu 260
 - Actions trial attribute, see *attributes* under *templates*
 - active until, see *actions*
 - “ActiveUntil” attribute 410
 - AddAttrib() function 355, 358
 - AddToList[] action 198, 417
 - scripting 422
 - AllExcept() function 353

- And() function 351
- AppendTok() function 355, 358
- Arctan() function 351
- asterisk, see *columns with non-literal values in files under lists*
- “Async” attribute 470
- attribute blocks (PsyScript) 276, 321
 - referencing 329
- attributes 149
 - block, see *blocks*
 - custom 149, 156
 - creating 73
 - default 149
 - Default Stimulus/Event/Trial Attribs 150, 159
 - Event 169, 171, 173
 - Stimulus 169, 171, 173
 - Trial 169, 171
 - default value 149
 - event 17, 149
 - event, see *events*
 - experiment, see *experiments*
 - factor tables and 150
 - group, see *groups*
 - hierarchy 149
 - inheritance 150
 - inheritance (PsyScript) 296, 325, 344
 - Factor format 403
 - inline entries and, see *inline entries*
 - StimList/EventList format 372
 - linking to variables, see *variables*
 - PsyScript 276, 321
 - @, see *references in attribute blocks*
 - naming 321
 - references 323
 - renaming 157
 - sets 149
 - setting 16, 151, 152
 - settings
 - Default 150, 152
 - Script 154
 - Access 156
 - Current 156
 - Linked 155
 - Multiple 156
 - Other 156
 - Set To 152
 - Vary By 152
 - Block 72, 154
 - Group 83, 154
 - List 152, 209
 - PsyScript analogues 384, 385
 - Run Mode 154
 - Template 154
 - standard 149
 - stimulus 17, 149
 - stimulus, see *stimuli*
 - structural vs. non-structural 402
 - Trial Chooser and 152
 - trial, see *trials*
- automatic group selection, see *groups*
- Automatic Grouping dialog 234
- B**
- “BackColor” attribute 429
- BackColor experiment attribute, see *attributes under experiments*
- background color, see *colors*
- Balance menu item 259
- “BaseCellWeights” sub-attribute 393
- BBox dialog extension 477
- BBox event type, see *types under events*
- BBox Init experiment attribute, see *attributes under experiments*
- BBox, see *button box*
- BBox[] condition, see *button box*
- “BBoxInitialState” attribute 437
- BboxOut 199, 421
- “BBoxPort” attribute 437
- Beep[] action 199
 - scripting 422
- beeping 199
- Between crossing type, see *crossing types*
- between-subjects design 57, 89, 133
 - selecting 57
- blanks (PsyScript) 277, 322
- Block Attributes dialog 172
- Block dialog 68, 122
- block entries 380
 - attributes
 - factoring and linking 381
- BlockAttrib() 309
- BlockAttrib() function 381, 385
- “BlockDuration” attribute 402
- Blocked Random access type, see *crossing types*
- Blocked Random level order, see *level orders in levels*
- Blocked Sequential level order, see *level orders<> in levels*
- blocks 65, 120
 - attributes 170
 - custom 171
 - configuring 121
 - creating 66
 - durations 120, 247
 - icon 66
 - linking to 120
 - opening 121
 - ordering, see *sequencing*
 - previewing 95, 209
 - quitting 201
 - renaming 122
 - scaling factor 211
 - setting 120
 - scripting 309
 - (see also *block entries*)
 - sequencing 70, 118, 120
 - (see also *ordering under lists*)
 - trial count 120
 - varying by, see *Vary By in attributes*
 - with blocks 121
 - with events (without templates) 121
 - with lists 121
- “Blocks” attribute 378, 379, 381
- Boolean values (PsyScript) 327
- breaking (an experiment run), see *experiments*
- Build Run File... menu item 254
- “BuilderData” entry 284
- “BuildStart” entry 460
- button box
 - as input device, see also *input devices*
 - bad cable hack 166
 - controlling lights 436
 - data recording, see *fields, input device states in data*

- file*
 - in conditions 194
 - scripting 439
 - setting parameters 195
 - initial light state
 - (see also *BBox Init* under *experiment attributes*)
 - initial state, see *BBox Init* under *experiments attributes*
 - initial light state 437
 - removing use via PsyScript 463
 - serial port 437
 - setup and configuration 513
 - testing 196
 - dialog via PsyScript 463
 - Button Box Parameter dialog 195
 - buttons (in dialogs) 106
 - Buttons dialog extension 472
 - By Factor access type, see *access types*
- C**
- CancelAction[] action 199
 - scripting 422
 - carriage returns (PsyScript), see *blanks*
 - Cell Chooser menu item 257
 - cell weight, see *cells* under *factor tables*
 - cells, see *factor tables* and *factor sets* under *factors*
 - ChanceEvent[] action 200
 - scripting 420
 - Change Data File... menu item 255
 - Change Log File... menu item 255
 - “Check” attribute 452
 - Check function extension 476
 - Check Links menu item 256
 - checkboxes (in dialogs) 106
 - CheckBoxes dialog extension 472
 - checking trials, see *trials*
 - Choose Crossing dialog 143
 - Clean Up menu item 113, 256
 - Clear menu item 254, 258
 - Clearing event attribute, see *attributes* under *events*
 - clearing the screen 200
 - ClearPort[] action 200
 - scripting 431
 - ClearScreen[] action 200
 - scripting 430
 - ClearStim[] action 200, 443
 - scripting 420
 - “ClearType” attribute 368, 443
 - Close menu item 253, 257
 - “CloseAlert” attribute 453, 468
 - “CloseAlertMsg” attribute 453, 468
 - “Color” attribute 428
 - colors
 - background 204
 - setting, see *attributes* under *experiments*
 - drawing, see *foreground*
 - foreground 204
 - setting, see *attributes* under *experiments*
 - Column() function 335, 353
 - columns (in a list file), see *files* under *lists*
 - commas (PsyScript), see *blanks*
 - Comment Lines menu item 260
 - comments (PsyScript) 278, 319, 324
 - compact factors, see *factors*
 - condition (of a trial), see *trials*
 - condition name pattern 175
 - Condition Name trial attribute, see *attributes* under *templates*
 - condition-action pairs 193
 - scripting 409
 - “ConditionName” attribute 366
 - conditions 30, 192
 - (see also *input devices* and *actions*)
 - PsyScript evaluation 198
 - scripting 290, 409
 - terminating, see *events*
 - using variables 198
 - Conditions and Actions dialog 193
 - Conditions dialog 194
 - Conditions submenu 260
 - Connect List dialog 148
 - Console 3, 257
 - configuring through PsyScript 455, 467
 - “Console” entry 455
 - Console menu item 256
 - “Constant” attribute 384, 387
 - content (of an entry), see *entries*
 - context (from a factor table), see *factor tables*
 - conventions 2
 - Copy menu item 254, 258
 - CopyContent() function 356
 - Cos() function 351
 - Count Finds menu item 259
 - counterbalancing, see *stimuli*
 - “Criteria” attribute 486
 - Cross() function 347, 354
 - crossing types 132, 140
 - Between 140
 - Fixed 141
 - index 140, 143
 - (see also *subject numbers*)
 - Default 144
 - Group 144
 - Run 144
 - Run of Subject 144
 - Run within Group 144
 - scripting 390
 - Subject 144
 - Subject within Group 143
 - Latin Squares 63, 140
 - Latin square partition 63
 - Latin square partitions 134
 - setting 143
 - scripting 390
 - subtables 140
 - List Access 140
 - scripting 389
 - Use Access 140
 - Use/Reset Access 141
 - Within 140
 - “CrossingValues” attribute 395
 - curly braces (PsyScript) 326
 - “Current” attribute 282
 - current group, see *groups*
 - current script, see *scripts* in *projects*
 - “CurrentExperiment” entry 461

CurrentIndex() function 353
 cursor (hiding) 166
 custom attributes, see *attributes*
 Custom dialog extension 473
 Cut menu item 254
 Cycle Random access type, see *access types*
 Cycle Random level order, see *level orders* in *levels*
 “Cycles” attribute 401

D

data file 79, 217
 automatic naming 80, 236
 fields 161, 218
 default 219
 input device states 220
 optional 219
 showing, see *attributes* under *experiments*
 format 222
 header 217
 response data, see *fields* in *data file*
 setting, see *attributes* under *experiments*
 timing statistics 221
 value on every data line 362
 variables
 decimal places, see *attributes* under *experiments*
 showing, see *attributes* under *experiments*
 viewing 255
 when written 166
 Data File dialog 80, 236
 Data File experiment attribute, see *attributes* under *experiments*
 Data Info experiment attribute, see *attributes* under *experiments*
 Data Variables experiment attribute, see *attributes* under *experiments*
 “DataFields” attribute 362
 “DataFieldsMenuItem” entry 462
 “DataFile” attribute 362
 “DataHeader” attribute 362
 “DataRecordSeparator” attribute 365
 “DataVariables” attribute 365
 Date() function 355
 “DCOD” attribute 467
 DCODs
 (see also *PsyScope Extensions*)
 “Debrief” attribute 364
 Debriefing File experiment attribute, see *attributes* under *experiments*
 debriefing, see *experiments*
 debugging, see *experiments*
 Decimal Places experiment attribute, see *attributes* under *experiments*
 “Default” attribute 470
 Default attribute setting, see *settings* under *attributes*
 default attributes, see *attributes*
 Default Stimulus/Event/Trial Attribs attributes, see *attributes*
 “DefaultColor” attribute 429
 definitions 2
 “Degradation” attribute 429
 Deiconify menu item 257

DeleteAllToks() function 355, 358
 DeleteTok() function 355, 358
 “Depth” attribute 434
 Design menu 256
 absence, see *user mode*
 Design menu item 256
 design types 1
 Design window 11, 107
 automatically opening 268
 Notes button 114
 opening 107
 Show Events checkbox 114
 Show Lists checkbox 114
 Subject Info icon 113
 tools 109
 (see also *tools*)
 trash can, see *trash*
 Variables icon 114
 Work area 107
 cleaning up 113
 hierarchical arrangement 268
 Diag() function 353
 “Dialog” attribute 450, 467
 dialog entry 467
 dialog extensions 451, 467
 standard 469
 dialogs 105, 106
 scripting 450
 (see also *dialog extensions*)
 setting the title 454
 diamond 41
 distributivity (of a PsyScript operator), see *operators*
 Div() function 351
 “DLOG” attribute 471
 “DLOGx” attribute 471
 “DLOGy” attribute 471
 Do Run File... menu item 254
 Document event type, see *types* under *events*
 DrawAllPortBorders[] action 200
 scripting 430
 drawing color, see *foreground* under *colors*
 DrawPortBorder[] action 200
 scripting 430
 “Duration” attribute 287, 367, 411
 Duration dialog 180
 Duration event attribute, see *attributes* under *events*
 duration, see *events*

E

Edit menu 254, 258
 Edit This Script menu item 253
 editor 258
 (see also *text tools*)
 auto-tab 267
 interactive mode 261, 284
 keyboard shortcuts 260
 searching and replacing 262
 wrapping text 259, 268
 Editor dialog extension 477
 emacs keys 260
 Empty Trash menu item 115, 256
 “Enabled” attribute 453
 “EnableParts” attribute 469

- End[] condition 417, 443
 - scripting 437
- EndEvent[] action 201, 443
 - scripting 419
- EndWatchCursor() function 356
- entries 275, 319, 320
 - content 276, 320
 - defining 321
 - executing 467, 468
 - finding in the script 261
 - from a file reference, see *file entries*
 - global 321
 - inline entries and, see *global entries and under inline*
 - inline 335
 - attribute inheritance and 346
 - attributes 337
 - difference from regular entries 336
 - global entries and 338
 - lists and 338
 - THIS keyword and 341
 - token reference 339
 - using () 339
 - using [] 335
 - linking to from graphic environment 154
 - names 276
 - reserved 459
 - references 279, 323
 - special 459
 - syntax 321
- EntryList dialog extension 477
- EntryName() function 354
- error messages 489
 - repeating 262
 - suppressing 266
- Evaluate Again menu item 255
- Evaluate menu item 255
- Evaluate() function 354
- evaluation (of a PsyScript expression) 320
- Evaluator 263, 356
- Evaluator menu item 257
- Event Attributes dialog 16, 176
- event attributes, see *events and attributes*
- event bar 16
- event entries 383
 - attributes
 - factoring 384
 - linking to run mode 385
 - linking to templates, blocks, and groups 385
- Event Monitor 98, 241
 - step mode 245
- Event Name area (of the Template window), see *Template window*
- Event palette 14, 109
- event types 14, 177
 - (see also *events*)
- “EventActions” attribute 368, 409
- EventList format 368
 - (see also *StimList format*)
- “EventName” attribute 367
- events
 - aborting 198
 - actions 193
 - scripting 409
 - setting, see *attributes under events*
 - attributes 176
 - Actions 180
 - Clearing 180
 - Duration 19, 179
 - Load Time 180, 251
 - standard (PsyScript) 367
 - Tag 180
 - clearing 166
 - (see also *clearing in stimuli*)
 - setting clear type, see *attributes under events*
 - constant (Factor format) 387
 - creating 15, 112
 - creating (PsyScript) 285
 - default name in StimList/EventList format 373
 - duration 19, 127, 411
 - calculating 443, 444
 - Sound events 447
 - changing 127
 - setting to until mouse click 20
 - setting, see *attributes under events*
 - duration (PsyScript), see “Duration” attribute
 - ending 201, 443
 - event bar 126
 - load time 180
 - setting, see *attributes under events*
 - loading, see *loading under stimuli*
 - masking 201
 - masking, see *clearing*
 - offset 127
 - ports, see *ports*
 - renaming 126
 - running 200, 203, 204, 442
 - running multiple times 444
 - scheduling 127, 204
 - scheduling dependency 127
 - scripting 285
 - (see also *event entries*)
 - sequencing 22, 411
 - sequencing (PsyScript) 287
 - showing 204
 - starting 442
 - statuses (run-time) 243
 - tags
 - setting, see *attributes under events*
 - terminating condition, see *duration*
 - timeline 126
 - moving 126
 - types 177
 - Button Box
 - attributes
 - scripting 436
 - Document 177
 - attributes 182
 - scripting 431
 - icons 112
 - Key Sequence 178
 - attributes 184
 - scripting 432
 - Paragraph 178
 - attributes 183
 - scripting 432
 - icon 67
 - Pasteboard 178
 - attributes 183
 - scripting 434
 - PICT 177
 - attributes 182
 - scripting 433
 - Sound 179
 - attributes 184
 - scripting 435
 - Sounds

- creating sounds 519
 - Text 177
 - attributes 181
 - scripting 424
 - icon 15
 - position, see *positions* under *ports*
 - setting constant stimulus 18
 - Time 177
 - attributes 181
 - icon 23
 - unscheduled 127
 - unscheduling 204
 - “Events” attribute 378, 379, 381, 382
 - Events palette 112
 - “EventTag” attribute 367
 - “EventType” attribute 367
 - execution entries 460, 467
 - Exp Keywords submenu 260
 - Exp() function 351
 - expanded list, see *lists*
 - Experiment Attributes dialog 160
 - experiment attributes, see *experiments*
 - Experiment dialog 117
 - experiment entries 282, 359
 - attributes
 - factoring and linking 379
 - finding current 461
 - syntax 378
 - Experiment menu 256
 - Experiment Notes menu item 255
 - “ExperimentClose” entry 460
 - experiments 116
 - attributes 90, 158
 - Backcolor 168
 - BBox Init 168
 - Data File 80, 161, 217, 236, 255
 - Data Info 161
 - dialog via PsyScript 462
 - Data Variables 163
 - Debriefing File 165
 - Decimal Places 168
 - Forecolor 168
 - Input Devices 164
 - dialog via PsyScript 462
 - Instructions File 165
 - Num Rests 165
 - Optimization 167
 - dialog via PsyScript 463
 - Precompile 167
 - Resources 167, 216
 - Rest Duration 165
 - scripting 299
 - Special 165
 - dialog via PsyScript 462
 - standard 161
 - standard (PsyScript) 360
 - Timer 164
 - dialog via PsyScript 463
 - Trials per Rest 165
 - breaking 92
 - comments, see *notes*
 - compiling 246
 - (see also *precompiling*)
 - algorithm 357
 - Factor format 404
 - quickly 246
 - statistics 241
 - suppressing time bar count, see *time bar*
 - configuring 116
 - creating 10
 - current 257
 - PsyScript 282
 - debriefing 91
 - setting, see *attributes* under *experiments*
 - debugging 238, 241
 - hierarchy, see *hierarchyexperiment hierarchy, see hierarchy*
 - instructions 91
 - creating 91
 - setting, see *attributes* under *experiments*
 - suppressing 165
 - via blocks 66
 - notes 114, 255
 - number of trials, see *trial counting*
 - opening 116
 - renaming 117
 - running 4, 19, 92, 238, 441
 - scripting 281
 - (see also *experiment entries*)
 - special flags
 - setting, see *attributes* under *experiments*
 - title for data logging 360
 - verifying the structure 256
 - with groups 116
 - with lists 116
 - without groups 116
 - “Experiments” entry 282, 359, 459
 - “ExperimentStart” entry 460
 - expressions (PsyScript) 320
 - evaluating 263
 - function calls, see *function calls (PsyScript)*
 - literals, see *literals (PsyScript)*
 - expressions (variable), see *variables*
 - “ExpTypes” attribute 415
 - “ExpVariables” attribute 365, 413
 - extensions, see *PsyScope extensions* and *dialog extensions*
 - “ExtResult” attribute 340
- ## F
- f^ (PsyScript) 350
 - “Face” attribute 427
 - Factor format 376
 - compilation details 404
 - optimization 387, 404
 - setting, see *attributes* under *experiments*
 - script summary 405
 - technical details 402
 - Factor Set dialog 147
 - factor set, see *factor set* under *factors*
 - Factor Table floating window 144
 - factor table set entries 395
 - factor table sets 58, 130
 - Factor Table window 36, 138
 - factor tables 36, 129
 - access types, see *access types*
 - accessing cells
 - sequential 51
 - and lists 132
 - built-in 130
 - cell navigation 41
 - cell selection 41
 - cells
 - current 130, 135
 - ordering 139

- (see also *access types*)
 - (see also *ordering under lists*)
 - selecting 139
 - weight 54, 133
 - setting 54
 - context 139, 144, 150
 - scripting values 396
 - creating 37
 - creating new within a factor table set 138
 - crossing type, see *crossing types*
 - deleting from a factor table set 138
 - icon 4
 - ordering cells 49
 - renaming 138
 - scripting 309
 - (see also *factor table set entries*)
 - transforming 111
- FactorAttrib() function 384
- factors 36, 129
 - (see also *factor tables and lists*)
 - compact 303
 - creating in factor table 37
 - crossing 36, 37, 135
 - crossing factors 129
 - factor set 129
 - cells 129
 - scripting weights 393
 - scope 136
 - scripting 400
 - scripting 306, 388
 - set names 389
 - free
 - scripting 301
 - in a factor table
 - creating 138
 - nested, see *nested under factors*
 - ordering 139
 - renaming 138
 - scripting 395
 - in a table
 - ordering 52
 - nested 132
 - in a factor table
 - creating 138
 - scripting 396
 - owning level 132
 - scripting 307, 393
 - ordering 143
 - scripting 299
 - referencing 384
- “Factors” attribute 378, 379, 381, 382
- “FactorTable” attribute 395
- “FactTypes” sub-attribute 395
- “Feature” attribute 428, 433
- fields (of a list), see *lists*
- Fields dialog extension, see *Standard dialog extension*
- file entries 343
 - optimizing 349
 - with expressions 343
 - with literals only 343
- File menu 253
- file name paths, see *paths*
- file references (PsyScript), see *file entries*
- FileLists dialog extension 456, 480
- FileRef() function 350
- files
 - getting names from PsyScript 456
 - resource, see *resources*
 - saving (safe saves) 224
 - using as a list (PsyScript), see *file entries*
 - using as a list, see *files under lists* 131
- Find Again menu item 259
- Find dialog 262
- Find... menu item 258
- Fixed crossing type, see *crossing types*
- “FixedCycles” attribute 401
- “Flags” attribute 361
- “Flip” attribute 429
- folder icon 217
- “Font” attribute 427
- “ForeColor” attribute 429
- Forecolor experiment attribute, see *attributes under experiments*
- foreground color, see *colors*
- “Format” attribute 360
- free factors, see *factors*
- “Function” attribute 451, 467
- function calls (PsyScript) 280, 320, 327
 - exceptions to rules 327
- function extensions 451, 467
 - (see also *dialog extensions*)
- functions (PsyScript) 326
 - (see also *function calls*)
 - built-in 327
 - creating 340
 - listing 349
- ## G
- GCD() function 351
- Get [Object]... menu item 115, 254
- GetCurrent() function 334, 352
- GetPrevCurrent() function 353
- GetSome() function 353
- GetSubjNum function extension 483
- GetTag() function 386
- GetToks() function 350
- Graeco-Latin square 133
- graphic environment 1, 9
- “Grip” attribute 332
- Group Attributes dialog 170
- Group Criteria dialog 235
- group criteria, see *grouping criteria under groups*
- Group dialog 119
- group entries 379
 - attributes
 - factoring and linking 380
- GroupAttrib() 312
- GroupAttrib() function 380, 385
- groups 118
 - attributes 169
 - custom 169
 - configuring 119
 - current 88, 116, 379
 - setting manually 88
 - grouping by subject info values 82
 - grouping criteria 82, 233
 - linking to 84
 - modulo comparison 234
 - icon 83
 - linking to 118

- opening 119
- renaming 119
- scripting 311
 - (see also *group entries*)
- selecting for a run 118
 - automatically 233, 485
- varying by, see *Vary By in attributes*
- with blocks 118
- with lists 118
- without blocks 118

“Groups” attribute 378

H

Head() function 354

“Height” attribute 470

help 264

- searching for, see *Help Search dialog*

Help Search dialog 264

Help... menu item 257

hierarchy 107

- attributes and, see *hierarchy* under *attributes*
- lists and, see *hierarchy* under *lists*
- skipping levels 108

hot spot (of a port), see *ports*

hyperdiagonal 134

I

Iconify menu item 257

IDEVs

- (see also *PsyScope Extensions*)

If() function 354

Import List... menu item 115, 132

index (for a factor set), see *crossing types*

index (user manual), see *index (user manual)*

“Indices” sub-attribute 390

inheritance (of attributes), see *attributes*

Inherited() function 354

“Init” attribute 412

inline entries, see *entries*

input devices

- (see also *button box*, *mouse*, and *keyboard*)
- activating, see *attributes* under *experiments*
- available 1
- in conditions 194

Input Devices experiment attribute, see *attributes* under *experiments*

“InputDevices” attributes 360

“InputDevicesMenuItem” entry 462

“Instances” attribute 410

instances, see *actions*

“Instructions” attribute 364

Instructions File experiment attribute, see *attributes* under *experiments*

instructions, see *experiments*

Interactive Mode menu item 259

interactive mode, see *editor*

interrupt 4

intertrial interval 96

- (see also *ITI*)

IsItemInList() function 355

“IsList” attribute 394

item weight, see *items* under *lists*

ItemList dialog extension 474

“ItemName” attribute 453

items (for subject info), see *subject info*

items (in lists), see *lists*

Iterate() function 355

ITI 96, 246, 250

“ITI” attribute 366

ITI trial attribute, see *attributes* under *templates*

K

Key Parameter dialog 196

Key Sequence event type, see *types* under *events*

Key[] condition, see *keyboard*

keyboard

- as input device, see also *input devices*
- data recording, see *fields*, *input device states* in *data file*
- in conditions 194
 - scripting 438
 - setting parameters 196
- Key icon 28

keyboard shortcuts, see *shortcuts*

KeyState dialog extension 478

keywords 2, 114

L

Latin squares (general) 133

Latin Squares crossing type, see *crossing types*

Latin Squares dialog 142

LCM() function 351

Least-Used Random access type, see *access types*

Least-Used Random level order, see *level orders* in *levels*

Level dialog 147

level weight, see *levels*

levels 36, 129

- adding to factor table 38
- crossing, see *crossing* under *factors*
- free 132
- in a factor table
 - creating 138
 - renaming 138
- moving in a table 51
- ordering 132
 - (see also *ordering* under *lists*)
- level orders 141
 - Blocked Random 141
 - Blocked Sequential 141
 - Cycle Random 141
 - Least-Used Random 142
- owning nested factors 132
- weight 55
 - setting 56

“Levels” attribute 387

linear list, see *lists*

Link to Entry dialog 155

link tool, see *tools*

linking, see *objects*

List Access crossing type, see *crossing types*

list boxes (in dialogs) 106

List dialog 145

- tick marks 146

List File dialog 146

list file, see *files* under *lists*

- List Script Changes menu item 255
 - lists 36, 59, 130
 - accessing 60
 - creating 59
 - expanded 132
 - fields 130
 - creating 146
 - renaming 157
 - files 131
 - columns 131
 - marked with an asterisk, see *with non-literal values*
 - with non-literal values 131, 147
 - graphic environment vs. PsyScript 329
 - hierarchy and 135
 - in a factor table 132
 - items 60, 130
 - adding to list 61
 - weight
 - setting 61, 146
 - linear 131
 - ordering 210
 - renaming 145
 - using 130
 - lists (PsyScript) 279, 329
 - access types, see *access types (PsyScript lists)*
 - accessing 330
 - checklist 330
 - crossing 347
 - by map 348
 - checklist storage 349
 - current item 330
 - previous 334
 - inline entries and, see *inline entries*
 - items 330
 - linking 332, 339
 - offset 333
 - resetting 330, 366
 - size 330, 333
 - sublists 334
 - weights 332
 - ListSize() function 352
 - ListSum() function 353
 - ListWeight() function 352
 - literals (PsyScript) 320, 326
 - Load Time event attribute, see *attributes under events*
 - “LoadTime” attribute 443
 - Log Comment menu item 255
 - log file 79, 222, 458
 - format 223, 458
 - logging a comment 255
 - logging data from PsyScript 458
 - setting 222, 255
 - subject info and, see *subject info*
 - viewing 223, 255
 - “Log File” entry 458
 - Log function extension 458
 - Log() function 351
 - LogInfo function extension 482
 - Lowercase menu item 260
- M**
- magnet icon 109, 115
 - MakeFileName function extension 481
 - Map() function 335, 353
 - MappedCross() function 348, 354
 - mapping list (PsyScript) 335
 - “Margin” attribute 470
 - “Mask” attribute 428
 - MaskStim[] action 201
 - scripting 420
 - Match() function 355
 - Max() function 351
 - maximum instances, see *actions*
 - memory (conserving) 246, 251
 - (see also *screen saving*)
 - “MenuName” attribute 453
 - menus 253
 - customization through PsyScript, see *script-specific*
 - script-specific 256, 449, 467
 - checkmarked items 452
 - disabling menus/items 453
 - standard 461
 - submenus 455
 - “Menu” entry 449
 - “Mode” attribute 427
 - modifiers (PsyScript) 319, 324
 - Monitor menu item 254, 257
 - “MonitorOrder” attribute 430
 - mouse
 - as input device, see also *input devices*
 - data recording, see *fields, input device states in data file*
 - icon 21
 - in conditions 194
 - scripting 439
 - setting parameters 197
 - Mouse Parameter dialog 197
 - Mouse[] condition, see *mouse*
 - MouseState dialog extension 477
 - “Msg” attribute 468
 - “Multiple” attribute 332
 - muon field, see *PsyBug*
- N**
- New Attribute dialog 157
 - New Experiment menu item 256
 - New Object menu item 254
 - New Object Name dialog 114
 - New Port dialog 189
 - New Positions dialog 189
 - New Project... menu item 253
 - New Text File menu item 253
 - New/Reconfigure Info Item dialog 228
 - New/Rename Table Factor dialog 142
 - New/Rename Table Level dialog 142
 - newlines (PsyScript) see *blanks*
 - NewListItem[] action 201, 417
 - scripting 423
 - Next() function 352
 - NextCrossing[] action 201
 - scripting 424
 - “NoDirty” attribute 469
 - notes (for an experiment), see *experiments*
 - Notes button (in Design window), see *Design window*
 - “Notes” entry 284
 - NthChar() function 352

Null() function 354
Num Rests experiment attribute, see *attributes* under *experiments*
“NumRestPeriods” attribute 364
“NumTrialsPerRest” attribute 364

O

Object List dialog 115
objects 107

- arranging 268
- creating 112
- linking 108, 110
 - creating multiple links 111
- names 13, 114
 - prompting for 268
- owner 107, 111
- recovering from trash, see *trash*
- renaming 151
- selecting 105, 110

Objects in Script menu item 256
Objects in Script submenu 115
Objects palette 109, 112
ODEVs
(see also *PsyScope Extensions*)
“Offset” attribute 333
Open Selection menu item 253
Open Text File... menu item 253
Open... menu item 253
“OpenAlert” attribute 453, 468
“OpenAlertMsg” attribute 453, 468
operation sentence (PsyScript) 320, 328
operators (PsyScript) 326, 328

- binary 328
- distributivity 328
- listing 349
- precedence 356
- unary 328

operators (trial variable) 417
optimization (PsyScript)
see *optimizing* under *scripts*
“Optimization” attribute 378
Optimization experiment attribute, see *attributes* under *experiments*
optimization, see *Factor format* or *StimList format*
“OptimizeMenuItem” entry 463
options 265

- custom 269
 - defining 456
- Design 268
- editor 267
- general 266
- projects and 213
- run 266
- script-based
see *custom* under *options*

“Options” entry 456
Options submenu 254
Or() function 351
“Origin” attribute 429
output devices

- available 1

owner (of an object), see *objects*
OWNER keyword (PsyScript) 324, 342, 350

P

palettes 109
Palettes menu item 115, 257
Paragraph event type, see *types* under *events*
parameter tags (PsyScript) 342
parameters (of actions) 32
Parameters dialog 205
parentheses (PsyScript) 320, 339
parentheses, balancing 259
“PartIDs” attribute 471
pass (through a list of blocks) 380
Paste menu item 254, 258
Pasteboard event type, see *types* under *events*
paths 215

- projects and 213
- relative 215
 - projects and 215
 - setting start 266
- reverse notation 215

“PBoardDegradation” attribute 434
“PBoardDepth” attribute 434
“PBoardMode” attribute 434
“PBoardPort” attribute 434
PICT event type, see *types* under *events*
Picture dialog extension 482
“Placement” attribute 470
PopUp dialog extension 475
pop-up menus (in dialogs) 106
“Port” attribute 424
Port Info dialog 190
ports 187

- clearing 200, 202
- default 187, 434
- drawing 166, 200
- hot spot 187
- positions 187
 - creating 46
 - setting 45

Ports/Positions dialog 187
PosItemInList() function 355
“Position” attribute 427
Position Info dialog 191
Positions dialog extension 478
positions, see *ports*
Power() function 351
practice 93
(see also *run mode*)
Practice menu item 254
“PracticeBlockDuration” attribute 402
“PracticeBreak” entry 460
“PracticeCycles” attribute 401
“PracticeEnd” attribute 460
“PracticeFixedCycles” attribute 401
“PracticeScaleBlocks” attribute 401
“PracticeStart” entry 460
“Precompile” attribute 365
Precompile experiment attribute, see *attributes* under *experiments*
precompiling 96, 166

- problems 247
- setting fixed count, see *attributes* under *experiments*

- preferences, see *options*
 PrevCurrentIndex() function 353
 previewing, see *trials* and *blocks*
 privileges 461
 processes 237
 scheduling 231, 237, 460
 Product() function 351
 progress bar, see *time bar*
 Project Scripts menu item 254
 projects 213
 autoloading 266
 creating 213, 253
 options and, see *options*
 paths and, see *paths*
 scripts
 adding and removing 214
 current 214
 start-up 214
 subject info and 228
 “Prompts” attribute 470
 PsyScope Extensions 216
 PsyScope Extensions folder 216
 PsyScopeStdLib 461
 PsyScript 9, 103
 basic syntax rules 321
 defining an experiment 357
 environment configuration 312, 449
 introduction 275
 reference manual 319
 tutorial 281
 PSYXs, see *PsyScope Extensions*
 Put in View menu item 256
- ## Q
- Quit menu item 253
 QuitBlock[] action 201
 scripting 423
 QuitTrial[] action 202, 445
 scripting 419
 quotes (PsyScript) 278, 326
- ## R
- radio buttons (in dialogs) 106
 Random() function 355
 “Range” attribute 452, 468
 “RangeFailMsg” attribute 453, 468
 references (PsyScript), see *entries*
 references (to PsyScript attribute blocks), see *attribute blocks*
 Reinitialize Script menu item 255
 RemoveDups() function 355
 RemoveFromList[] action 202, 417
 scripting 423
 RemoveMatching() function 355
 RemovePortBorder[] action 202
 scripting 430
 Rename/Retype Attribute dialog 157
 Replace All menu item 259
 Replace and Find Again menu item 259
 Replace menu item 259
 RerunTrial[] action 202
 scripting 423
 “Reset” attribute 366
 Reset() function 330
 ResetAll() function 330, 353
 ResetList() function 353
 resources 216
 files
 creating PICTs 517
 opening
 (see also *attributes* under *experiments*)
 “Resources” attribute 365
 “Resources” entry 216, 461
 Resources experiment attribute, see *attributes* under *experiments*
 responses
 in data file, see *data file*
 recording 26, 203
 (see also *RT[] action*)
 Rest Duration experiment attribute, see *attributes* under *experiments*
 rest period 92, 167
 setting, see *attributes* under *experiments*
 “RestPeriod” attribute 364
 “Result” attribute 340
 reverse notation, see *paths*
 reverse video 92, 203
 dialog for setting via PsyScript 462
 ReverseVideo[] action 203
 scripting 431
 “ReverseVideoMenuItem” entry 462
 Revert File menu item 254
 Revert Script menu item 253
 Row() function 335, 353
 RT[] action 26, 203, 218, 417
 scripting 421
 run files 266
 Run menu 254
 Run menu item 254
 run mode 93
 (see also “*Run Mode*” attribute)
 linking attributes to, see *attributes* under *event entries*
 Run() function 355, 467
 “RunAfter” attribute 469
 “RunBefore” attribute 468
 “RunBreak” entry 460
 “RunEnd” entry 460
 RunEvent[] action 203, 442
 scripting 419
 vs. ScheduleEvent[] 444
 “RunFailMsg” attribute 468
 “RunLabel” attribute 362
 “RunMode” attribute 158, 365
 RunModeAttrib() function 378, 385
 “RunStart” attribute 460
- ## S
- Save a Copy As... menu item 253, 254
 Save File As... menu item 254
 Save File menu item 254
 Save Script As... menu item 253
 Save Script menu item 253

- enabled due to PsyScript operations 469
- “SaveCurrents” attribute 334
- “ScaleBlocks” attribute 401
- scaling factor (of a block), see *blocks*
- ScheduleEvent[] action 204
 - scripting 420
 - vs. RunEvent[] 444
- scissors tool, see *tools*
- screen
 - clearing 200
 - displaying stimuli 445
 - (see also *types* under *events*)
 - timing issues 446
 - multiple 430
 - timing issues 446
 - non-standard 166
 - saving 165
 - setting co-ordinates origin 429
- Script attribute setting, see *settings* under *attributes*
- Script Enabled menu item 259
- Script Functions submenu 260
- Script Keywords submenu 260
- ScriptEval[] action 204
 - scripting 422
- scripting, see *PsyScript*
- scripts 9, 275
 - autoloading 266
 - auto-scrolling to background changes 259
 - creating (for PsyScript editing) 281
 - debugging, see *debugging* under *experiments*
 - editing, see *editor*
 - format of 358, 360
 - grouping, see *projects*
 - including in another script 325
 - opening 253, 257
 - optimizing 349
 - projects and, see *projects*
 - reinitializing 255
 - saving automatically 266
 - self-modifying 358
 - standard format 281
 - suspending interpretation 259
- ScriptWhen Parameter dialog 198
- ScriptWhen[] condition
 - scripting 437
- Scroll to Changes menu item 259
- section markers (PsyScript) 319, 326
- Select All menu item 258
- selection devices (in dialogs) 106
- Selection to Search menu item 259
- selection tool, see *tools*
- Set To attribute setting, see *settings* under *attributes*
- Set[] action 203, 417
 - scripting 422
- SetBackColor[] action 204
 - scripting 431
- SetCurrent() function 353
- SetDefaultColor[] action 204
 - scripting 431
- “SetNames” sub-attribute 389
- “Sets” sub-attribute 388
- “SetScopes” sub-attribute 400
- SetTag() function 386
- “SettingsMenuItem” entry 462
- “SetUp” attribute 471
- Shift Left menu item 259
- Shift Right menu item 259
- shortcuts 111, 157, 224, 260
- Show Events checkbox (in Design window), see *Design window*
- Show Lists checkbox (in Design window), see *Design window*
- Show Trash menu item 256
- ShowStim[] action 204, 443
 - scripting 420
- Shut Up button 262
- “Shutdown” entry 460
- side-effect operations (PsyScript) 358
- Sin() function 351
- “Size” attribute 427
- snooping around 9
- Sound event type, see *types* under *events*
- SoundEdit™ 519
- SoundLabel event type, see *Sound event type*
- spaces (PsyScript), see *blanks*
- Special experiment attribute, see *attributes* under *experiments*
- square brackets (PsyScript) 335
- Standard Attributes dialog 151
- standard attributes, see *attributes*
- Standard dialog extension 475
- START event 442
- start references 411
 - (see also “StartRef” attribute and *events sequencing*)
- Start/End Parameter dialog 197
- Start[] condition 417, 443
 - scripting 437
- “StartRef” attribute 289, 368, 411
 - default 166
- “StartUp” entry 460
- start-up script, see *scripts in projects*
- StartWatchCursor() function 356
- Statistics menu item 259
- StimList format 368
 - attribute summary 374
 - block mode 370
 - compilation details 375
 - optimization 373
 - trial attributes 370
- stimuli 185
 - attributes 181
 - clearing 200, 445
 - (see also *clearing* under *events*)
 - counterbalanced 63, 134, 135
 - keeping in memory 251
 - loading 247
 - controlling 249
 - preloading 166, 250
 - presentation 443
 - scripting sub-stimuli, see *sub-stimulus entries*
 - showing 204, 445
 - storing values and attributes 386
- “Stimuli” attribute 384, 434
- Stimuli dialog 185
- “Stimulus” attribute 384, 424, 431, 432, 433, 436
- stimulus attributes, see *stimuli and attributes*
- Stimulus dialog 185

- Strcat() function 352
 - StripEndChars() function 352
 - StripFrontChars() function 352
 - Strlen() function 352
 - “Style” attribute 428
 - Style dialog 185
 - Style dialog extension 479
 - sub-attributes (PsyScript) 277
 - “Subject” entry 464
 - subject info 74, 224
 - grouping by 82
 - items 225
 - creating 76, 228
 - modifying 228
 - scheduling prompts, see *processes*
 - special 225
 - log file and 230, 231, 232
 - old PsyScript utilities 464
 - Subject Info dialog 226
 - Subject Info icon (in Design window), see *Design window*
 - Subject Info menu item 257
 - Subject Info Schedule dialog 237
 - subject numbers 75, 230
 - calculation 232, 483
 - scheduling computation, see *processes*
 - “Subject_Age” entry 465
 - “Subject_DataFile” entry 465
 - “Subject_Group” entry 465
 - “Subject_Name” entry 465
 - “Subject_Number” entry 465
 - “Subject_Run” entry 465
 - “Subject_Sex” entry 465
 - “SubjectInfoDialogItems” entry 466
 - SubjectInfoLib 464
 - “SubjectMenuItems” entry 466
 - SubjectNumAndGroup function extension 485
 - Sublist dialog 155
 - Sublist() function 334, 353
 - sublists (of PsyScript lists), see *lists (PsyScript)*
 - “Submenus” attribute 455
 - sub-stimuli, see *stimuli*
 - sub-stimulus entries 383
 - (see also *event entries*)
 - Sum() function 351
 - Superblock dialog 123
 - superblocks 65, 121
 - icon 66
 - supergroups 379
 - Switch Experiment menu item 255
 - Switch... menu item 253
 - symbols 114
 - system requirements 1
- T**
- Table Info dialog 49, 140
 - tables, see *factor tables*
 - tabs (PsyScript), see *blanks*
 - Tag event attribute, see *attributes* under *events*
 - tags (Factor format) 386
 - Tail() function 354
 - “TakeDown” attribute 471
 - Tan() function 351
 - template entries 381
 - attributes
 - factoring and linking 382
 - template weight, see *template*
 - Template window 124
 - Event Name area 126
 - Timeline area 126
 - templates 13, 124
 - (see also *trials*)
 - attributes 172
 - Actions 175
 - Condition Name 175
 - custom 173
 - in StimList format 370
 - ITI 175
 - setting, see *attributes* under *templates*
 - Minimum ITI 250
 - standard (PsyScript) 366
 - built-in 130
 - creating 13, 122
 - icon 13
 - ordering 120, 122
 - (see also *ordering* under *lists*)
 - renaming 125
 - scripting 292
 - (see also *template entries*)
 - transforming 111
 - weight 120
 - setting 122
 - “Templates” attribute 378, 379, 381
 - “Test BBox” entry 463
 - text editor, see *editor*
 - Text event type, see *types* under *events*
 - text fields (in dialogs) 106
 - text tools
 - (see also *editor*)
 - balancing parentheses 259
 - changing case 260
 - commenting out lines 260
 - counting characters, words, and lines 259
 - counting occurrences 259
 - searching and replacing 262
 - shifting left/right 259
 - Text Tools submenu 259
 - THIS keyword (PsyScript) 324, 350
 - inline entries and, see *inline entries*
 - time bar 266
 - suppressing trial count 267
 - Time event type, see *types* under *events*
 - Time() function 354
 - Timeline area (of the Template window), see *Template window*
 - timeline, see *events*
 - “Timer” attribute 360
 - Timer experiment attribute, see *attributes* under *experiments*
 - “TimerMenuItem” entry 463
 - timing
 - precision 2
 - setting source, see *attributes* under *experiments*
 - TIMRs
 - (see also *PsyScope Extensions*)
 - “Title” attribute 360, 454

tokens (PsyScript) 320

tools

- auto-changing 268
- in floating window 114
- link 13, 110
- scissors 83, 111
- selection 110
- transform 111

Tools palette 109

Tools pop-up menu 261

transform tool, see *tools*

trash 113, 115

trial 12, 441

Trial Chooser floating window 94, 152, 209

Trial Chooser menu item 257

trial counting 211

- (see also *blocks*)
- scripting 401

Trial Manager 441

Trial Manager Variables dialog 208

Trial Manager variables, see *variables*

Trial Monitor 238

- reported trial count 212

Trial Template window 13

trial templates, see *templates*

trial variables, see *variables*

“TrialActions” attribute 367, 409

TrialAttrib() 297

TrialAttrib() function 382, 385

trials 124

- (see also *templates*)
- actions 193
 - scripting 409
 - setting, see *attributes* under *templates*
- attributes, see *attributes* under *templates*
- blocking 66
- checking 98
- compiling, see *experiments*
- condition name
 - setting, see *attributes* under *templates*
- number run in experiment, see *trial counting*
- previewing 94, 125, 209
- quitting 202, 445
- rerunning 202
- running, see *running* under *experiments*
- run-time information 242
- variables, see *variables*

Trials Monitor 95

Trials per Rest experiment attribute, see *attributes* under *experiments*

triggering actions 30

TruthVal() function 351

“TurnOffBBox” entry 463

“Type” attribute 412, 451

“Types” sub-attribute 389

U

Uncomment Lines menu item 260

Undo menu item 254, 258

UnscheduleEvent[] action 204

- scripting 420

“Update” attribute 412

Uppercase menu item 260

Use Access crossing type, see *crossing types*

Use/Reset Access crossing type, see *crossing types*

user mode 266

“UserLevelMenuItem” entry 461

Utilities menu 254

V

values (of a PsyScript expression) 320

Variable Monitor 98, 245

variables 205, 412

- arrays
 - adding items 198, 201
 - removing items 202
- built-in 207, 417
- defining
 - (see also *Trail Manager Variables dialog*)
- expressions 206, 417
- in actions, see *using variables* under *actions*
- linking attributes to 207, 247
- operational overview 206
- run-time information, see *Variable Monitor*
- scripting 412
 - arrays, see *composite types*
 - composite types 413
 - built-in 416
 - initializing 414
 - declaring 412
 - records, see *composite types*
 - type declarations 415
 - types 412
- setting values 203
- types 205

Variables icon (in Design window), see *Design window*

Vary By attribute setting, see *settings* under *attributes*

Vary by Block dialog 154

Vary by Group dialog 154

Vary by List dialog 153

Vary by Template dialog 154

View Data File menu item 255

View Log File menu item 255

View Trash dialog 115

View Trash menu item 115

W

warnings, see *error messages*

weights 210

- cell, see *cell weight*
- item, see *item weight*
- level, see *level weight*
- template, see *template weight*

“Weights” attribute 332

When Parameter dialog 198

When[] condition 417

- scripting 437

whitespace, see *blanks*

“Width” attribute 470

windows 105

- iconifying 269
 - (see also *Iconify menu item*)
- main, see *Console*
- remembering positions 269
- startup
 - animating 269

Within crossing type, see *crossing types*

Work area, see *Design window*
Wraparound menu item 259

X

“X” attribute 425
x-ed out folder icon 217
XRESs 340

Y

“Y” attribute 426

Z

Zoom menu item 257

