Rapid User Manual

Rapid User Manual

© 1997 by Emultek Ltd. All rights reserved.

Emultek Ltd.
POB 45002
Jerusalem
91450
Israel

Tel: 972-2-5870770
Fax: 972-2-5870773

Information in this manual is subject to change without notice and does not represent a commitment on the part of the vendor. The software described in this manual is furnished under a license agreement and may be used or copied only in accordance with the terms of that agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Emultek Ltd.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows is a trademark of Microsoft Corporation.
This program was developed using NeoAccess: © 1992–1996 NeoLogic Systems.

The NeoAccess software contained within this program is propriety to NeoLogic Systems and is licensed to Emultek Ltd. for distribution only for use in combination with Rapid. NeoLogic Systems makes no warranties whatsoever, expressed or implied, regarding this product, including warranties with respect to its merchantability or its fitness for any particular purpose.

Written and produced by Emultek Ltd.
Printed in Israel.

MAN-US-40

CONTENTS

About the User Manual xx
Getting Online Help xxi
Conventions Used in this Manual xxii

Part 1: Getting Started

CHAPTER 1: STARTING RAPID 1–1
Opening Rapid 1–2
Working With Rapid Applications 1–3
Creating a New Application 1–3
Opening an Existing Application 1–3
Saving an Application 1–4
Loading a Backup Application 1–4
Generating a Runtime Application 1–5
Exiting Rapid 1–5
Opening Rapid Tools 1–5
Customizing Your Application 1–6
Changing Rapid's Font 1–6
Setting the Default Scale 1–7
Tool Tips 1–9
Saving Settings on Exit 1–10

Part 2: Rapid Objects

CHAPTER 2: THE OBJECT LAYOUT 2–1
What is an Object? 2–2
Types of Objects 2–2
Sample Applications 2–3
Behavior of Objects 2–8
Hierarchy of Active Objects 2–11
Object Names 2–11
The Object Layout Window 2–12

Customizing The Object Layout 2–12
Resizing the Layout Frame 2–13
Hiding/Showing the Object Palette 2–13
Using the Grid 2–13
Showing Mouse Coordinates 2–14
Adding Graphic Objects 2–14
General Procedure 2–14
Selecting a Graphic Object to Add 2–15
Placing a Graphic Object 2–18
Modifying an Object's Parameters 2–20
Basic Object Parameters 2–21
Working With Graphic Objects 2–24
Selecting and Deselecting Graphic Objects 2–24
Moving Graphic Objects 2–27
Resizing Graphic Objects 2–28
Cutting, Copying and Pasting Graphic Objects 2–29
Duplicating Graphic Objects 2–32
Deleting Graphic Objects 2–32
Undoing Operations 2–33
Grouping Graphic Objects 2–33
Aligning, Centering and Distributing Graphic Objects 2–36
Changing Objects' Stacking Order 2–41
Hiding and Showing Graphic Objects 2–42
Editing Objects' Colors 2–43
Editing a Graphic Object in the Object Editor 2–47
Replacing Graphic Objects 2–47
Adding and Viewing Notes on Objects 2–49
Redrawing the Layout 2–49
Adding Bitmaps to Your Layout 2–50
Pasting from the Clipboard 2–50
Importing a Bitmap 2–50
Importing Bitmaps in 256 Color Mode 2–51

CHAPTER 3: PRIMITIVE OBJECTS 3–1
Frames 3–2
Circles and Ellipses 3–2
Lines 3–3
Drawing a Line Object 3–3
Label 3–4
Adding a Label Object 3–4

Modifying a Label Object 3–5
Setting the Line Width 3–6
The Line Width Box 3–6
Adjusting the Line Width for Existing or New Objects 3–7
Making Primitive Objects Active 3–8
Naming a Primitive Object 3–8
Adding Properties to Primitive Objects 3–8
Making a Primitive Object Dynamic 3–11
Making a Primitive Object Drag 'n Drop 3–11

CHAPTER 4: ACTIVE OBJECTS 4–1
Lamps 4–2
Indicators 4–3
Dials 4–3
Linear Indicators 4–7
Bar Indicators 4–11
Barrel Dials 4–12
Pointers 4–17
Sector Dial 4–22
Potentiometers 4–25
Rotary Potentiometer 4–25
Slider Potentiometers 4–26
Jogdial 4–27
Pushbuttons 4–31
Switches 4–32
Rocker Switch 4–33
Toggle Switches 4–34
Rotary Switch 4–34
Slider Switches 4–35
Cross Switch 4–35
Stepper Switch 4–36
Joystick 4–37
Displays 4–39
Text Display 4–39
Text Window 4–44
Menu 4–47
Animated Object 4–50
Changing the Animated Object's Bitmaps 4–50

Flow Objects 4–52
Pipe 4–52
Valves 4–56
Tanks 4–56

CHAPTER 5: NONGRAPHIC OBJECTS 5–1
Adding Nongraphic Objects 5–2
Managing Nongraphic Objects 5–3
Renaming A Nongraphic Object 5–4
Editing A Nongraphic Object 5–4
Copying A Nongraphic Object 5–4
Deleting A Nongraphic Object 5–5
Replacing A Nongraphic Object 5–5
Data Objects 5–6
String 5–6
Number 5–6
Integer 5–8
Random Number 5–9
Holder 5–10
Array 5–11
Data Store 5–15
Point 5–20
Time Objects 5–21
Timer 5–21
Stopwatch 5–21
Date 5–22
Time 5–23
Signal Objects 5–24
Event 5–24
Sound 5–24
Cursor 5–26
Message 5–26
System Objects 5–28
Ascii 5–28
Keyboard 5–28
Mouse 5–29
SystemCursors 5–29
SystemDate 5–29
SystemTime 5–29

CHAPTER 6: MULTIMEDIA OBJECTS 6–1
About Media Control Interface (MCI) 6–2
The Multimedia Objects 6–3
Compound vs. Simple Multimedia Objects 6–3
Adding and Defining a Multimedia Object 6–4
Adding a Multimedia Object 6–4
Defining the Multimedia Object's Properties 6–5
Managing Multimedia Objects 6–8
Using Multimedia Objects 6–8
Rapid Syntax for Multimedia Objects 6–8
Function Summary 6–10
Opening an Object 6–12
Playing an Object 6–15
Displaying an Image in a Destination Window 6–15
Time Formats 6–19
Play Parameters 6–19
Getting Information about an Object 6–20
Using the 'Notify' Parameter to Generate Events 6–20
Using the Wait Parameter 6–21
Stopping and Pausing Objects 6–22
Closing Objects 6–22

CHAPTER 7: COMMUNICATION OBJECTS 7–1
DDE Basics 7–2
DDE Server and Client 7–2
Applications, Topics and Items 7–2
DDE Objects 7–3
DDE Client 7–3
DDE Server 7–6
Examples Using DDE Objects 7–8
Example 1: Passing Data Between Rapid Applications 7–8
Example 2: Passing Data From Rapid to Excel 7–11
Example 3: Excel Performing Calculations on Data from Rapid 7–13
DLL Object 7–15
Defining DLL Parameters 7–17
DLL Object Logic 7–19
DLL Object Examples 7–20
Example 1: Changing the Cursor Position 7–20
Example 2: Performing a Calculation 7–23

Commlink Object 7–25
Using the Commlink Object 7–27
Commlink Object Logic 7–32
Error Codes 7–32
Commlink Object Example 7–35
Simulation Manager Object 7–38
The Simulation Manager: When? 7–38
Defining the Simulation Manager 7–39
How the Simulation Manager Interacts with Rapid 7–40
The Simulation Manager's Properties 7–44
Controlling the Simulation Window 7–45
Getting Information about the Simulation 7–46
Controlling the Simulation's Response 7–47
Receiving Events from the Simulation 7–47
Sending Events to the Simulation 7–48
Automatically Returning Events 7–48
Checking for a Particular Event 7–49
Setting the Simulation's Size and Position 7–50
Sending Messages to the Student 7–50

CHAPTER 8: OBJECT LIBRARIES 8–1
The Librarian 8–2
Working with Libraries 8–2
Creating a Library 8–2
Managing Library Objects 8–4
Using Library Objects 8–5
Saving an Object to a Library 8–5
Adding a Library Object to the Object Layout 8–6

CHAPTER 9: THE OBJECT EDITOR 9–1
The Object Editor Window 9–2
Creating New Objects 9–3
Saving New Objects 9–5
Editing Existing Objects 9–5
Editing Application Objects 9–5
Editing Library Objects 9–7
Saving Edited Objects 9–8
Working in the Object Editor 9–10
Changing the Object's Size 9–11
Using the Pixel Grid 9–11

Setting the Line Width 9–12
Centering New Graphic Elements 9–13
Propagating Changes 9–14
Zooming 9–14
Selecting Colors 9–15
Customizing Colors 9–15
Restoring the Bitmap/Active Areas/Pointer 9–16
Undoing 9–17
Bitmap Editing 9–17
The Editing Tools 9–18
Cutting, Copying and Pasting Sections of a Bitmap 9–21
Clearing and Refreshing the Bitmap 9–22
Manipulating Columns and Rows 9–22
Showing the Pointer 9–25
Active Areas Editing 9–26
Pointer Editing 9–28
Test Mode 9–29

Part 3: Rapid Logic

CHAPTER 10: RAPID LOGIC: THE BASICS 10–1
Modes 10–2
What Is a Mode? 10–2
Hierarchy 10–3
Types of Modes 10–4
Transitions 10–6
What Is a Transition? 10–6
Transition Types 10–7
Internal Transitions 10–10
Triggers 10–11
Conditions 10–11
Events 10–11
Multiple and Compound Triggers 10–12
Activities 10–13
What is an Activity? 10–13
Syntax of Activities 10–13
Mode Activities 10–14
Actions 10–16
Internal Actions 10–16

CHAPTER 11: THE MODE TREE 11–1
The Mode Tree Window 11–2
Developing the Mode Tree 11–2
Adding a New Mode 11–3
Editing a Mode Name 11–4
Reassigning the Default Mode 11–5
Removing the Default Designation 11–6
Hiding and Showing Modes 11–6
Cutting, Copying and Pasting Modes 11–7
Moving Modes 11–9
Inserting Modes 11–9
Removing Modes 11–11
Creating Transition Destinations 11–13
Creating A Transition Destination 11–13
Finding a Mode 11–15

CHAPTER 12: BUILDING LOGIC 12–1
The Logic Editor Window 12–2
Opening the Logic Editor 12–2
The Logic Palette 12–8
Opening the Logic Palette 12–8
Finding an Object in the Logic Palette 12–11
Accessing Find & Replace 12–11
Inspecting an Object 12–11
General Editing in the Logic Editor 12–12
Using the Edit and Popup Menus 12–12
Selecting Lines 12–12
Editing a Line's Logic 12–13
Changing a Column's Width 12–14
Creating Transition Destinations 12–14
Creating External Transitions 12–14
Creating Internal Transitions 12–16
Changing the Transition Type 12–16
Building Triggers 12–17
Building an Event Trigger 12–17
Building Condition Triggers 12–20
Building Mode Triggers 12–24
Building Compound Triggers 12–26
Building Actions and Activities 12–27
Mode Activities versus Entry/Exit Activities 12–27

Building Actions and Activities **12–28**
Changing an Activity's Type **12–29**
Order of Execution of Actions and Activities **12–30**
Manipulating Lines of Logic **12–31**
Cutting, Copying and Pasting **12–31**
Moving Lines **12–33**
Inserting and Deleting Lines **12–34**
Manually Typing the Logic **12–35**
Adding Comments to Logic **12–36**
Temporarily Removing Logic ("Remarking Out") **12–37**
Verification Test **12–38**
Running the Verification Test **12–38**
Examining Logic Errors **12–39**
Warning Messages **12–40**

CHAPTER 13: HOW TO USE OBJECT LOGIC **13–1**
Active Objects (General) **13–2**
Disabling an Object **13–2**
Testing an Object's Position in Relation to Another **13–2**
Dynamic Functions **13–4**
Changing an Object's Position or Size **13–4**
Conditions that Test Coordinates **13–6**
Changing a Primitive Object's Colors **13–7**
Dynamic Lines **13–8**
Strings **13–9**
"Padding" Strings with Spaces **13–9**
Changing Individual String Characters **13–10**
Determining the Format of a String **13–11**
Numbers and Integers **13–13**
Formatting a Number as a String **13–13**
Degrees and Radians **13–15**
Random Numbers **13–15**
Generating a Random Integer **13–16**
Data Stores **13–16**
Records, Fields and Elements **13–17**
Syntax for Addressing a Data Store **13–18**
Assigning Values to Data Store Records, Fields and Elements **13–19**
Saving and Loading Data Stores **13–21**
Removing Data Store Records **13–24**
Using Data Store Values in Conditions **13–25**

Arrays 13–26
Queries & Searches in Data Stores and Arrays 13–27
Instances Functions 13–27
Search Functions 13–28
Holders and Arrays of Objects 13–29
Holding Specific Objects 13–29
Generating Objects 13–30
"Releasing" Objects and Deleting Generated Objects 13–30
Events of Holders and Arrays of Objects 13–31
Time 13–31
Date 13–32
Event Object 13–32
As an Activity to Trigger a Remote Action 13–33
To Avoid Repeating Identical Lists of Actions 13–34
To Avoid Repeating Complex Conditions 13–35
Keyboard 13–36
Principles of Operation 13–36
Displaying Text with the Keyboard Object 13–37
Testing for the Control and Shift Keys 13–39
Mouse and Cursor 13–39
Activating/Deactivating the Mouse Object 13–40
Using the cursorEntered and cursorExited Events 13–40
Enabling/disabling the mouseMove event 13–41
Defining cursor shapes 13–41
Cursor Priorities 13–43
Setting and Changing Cursor Position 13–44
Specifying Mouse Event AutoRepeat Frequency 13–45
Barrel Dial 13–47
Starting and Stopping the Rotation 13–47
Setting the Current Value 13–48
Changing the Rotation Rate 13–48
Connecting Barrel Dials 13–49
Sector Dial 13–50
Using the reading Property 13–50
Jogdial 13–51
Rotating the Jogdial 13–52
Fast Rotation 13–53
Clicking the Jogdial 13–53
Setting the Jogdial Value 13–54

Switches 13–54
Joystick 13–57
Animated Object 13–58
Running the Animated Object 13–58
Moving the Animated Object 13–59
Putting multiple sequences in one animated object 13–59
Pipes 13–60
Activating the Pipe's Flow 13–60
Changing the Flow's Pattern 13–61
Changing the Flow's Speed 13–61
Valves 13–62
Activating a Pipe's Flow with a Valve 13–62
Tanks 13–62

CHAPTER 14: FIND AND REPLACE 14–1
Accessing Find And Replace 14–2
Using Find 14–2
Setting the Search Options 14–3
Initiating a Search 14–4
Display of Search Results 14–4
Using Replace 14–6
Replacing a String 14–6
Undo 14–8
Using Filters ("Advanced" Tab) 14–8
Other Find & Replace Features 14–9
Closing the Find & Replace Window 14–9
Stopping a Search 14–9
Jumping to a Logic Line 14–9
Deleting Lines from the Displayed List 14–9
Some Find & Replace Examples 14–10
Example 1: Change Logic 14–10
Example 2: Replace Text 14–11
Example 3: Going to an Activity 14–11

CHAPTER 15: DEBUGGER 15–1
Overview 15–2
The Debugger Window 15–2
Controlling Application Flow 15–3

Flow Display While Debugging 15–4
Debugger Status Messages 15–5
Breakpoint Buttons in Debugger Window 15–5
Active Mode in Mode Tree 15–6
Active Line in Logic Editor 15–6
Managing Breakpoints 15–7
Display of Breakpoints 15–7
Inspecting the Contents of an Object 15–8

CHAPTER 16: RAPID LOGIC: HINTS AND TIPS 16–1
Summary of Rapid Syntax 16–2
General 16–2
Conditions 16–2
Events 16–3
User-Defined Functions 16–3
Working with Arithmetic Functions 16–4
A Selector: Parent-to-child Transitions 16–5
Sequence of Operations During Transitions 16–7
How Does Rapid Decide which Modes to Activate? 16–7
Transition to an Exclusive Mode 16–9
Entering a Subtree of Exclusive Modes 16–13
Transitions within a Subtree 16–14
Exiting a Subtree of Exclusive Modes 16–15
Transitions To and From Concurrent Mode Subtrees 16–16
Optimizing Rapid Applications 16–17

CHAPTER 17: THE STATE CHART 17–1
The State Chart Window 17–2
Representation of Logic Elements 17–3
Logic Information Pane 17–5
Optimizing the State Chart 17–5
Setting the Font 17–6
Setting the Length of the Mode Names 17–6
Setting the Number of Displayed Levels 17–7
Displaying Transitions 17–8
Numbering Transitions 17–8
Highlighting Transitions 17–9
Navigating Through the State Chart 17–9
Choosing a View 17–9
Zooming the State Chart 17–16

Redrawing the State Chart 17–17
Logic Operations 17–17
Selecting a Mode 17–17
Selecting a Transition 17–18
Creating Transition Destinations 17–18
Creating Activities, Actions and Triggers 17–19

Part 4: Rapid Reusability

CHAPTER 18: USER-DEFINED FUNCTIONS 18–1
What is a User-defined Function? 18–2
User-Defined Activities 18–2
User-Defined Conditions 18–2
User-Defined Functions with Arguments 18–4
Building User-defined Functions 18–4
Building User-Defined Functions in the Logic Editor 18–4
Building User-Defined Functions in the Function Editor 18–7
Editing, Renaming and Deleting User-Defined Functions 18–9
Exporting User-Defined Functions 18–10
Assigning Help ID Numbers 18–12
User-defined Functions With Arguments 18–13
Examples of User-Defined Functions with Arguments 18–13
Converting all of the Objects to Arguments—"Generating Arguments" 18–14
Selecting Certain Objects to Convert to Arguments 18–16
Directly Typing a User-Defined Function's Arguments 18–17
Typing Arguments Without Adding Objects 18–18
Renaming a User-Defined Function with Arguments 18–19
Changing the Number of Arguments 18–20

CHAPTER 19: USER-DEFINED OBJECTS 19–1
What is a User-Defined Object? 19–2
When to Use a User-Defined Object? 19–2
How Does a User-Defined Object Behave in a Parent Application? 19–2
Building a User-Defined Object 19–4
Adding Objects to a User-Defined Object 19–4
Saving an Application as a User-Defined Object 19–5
Opening a User-Defined Object 19–5
Runtime User-Defined Objects 19–6
Nongraphic user-defined objects 19–6

Adding User Properties, Events and Functions 19–6
Adding User Properties to a User-Defined Object 19–7
Adding Events to a User-Defined Object 19–9
Adding Functions to a User-Defined Object 19–10
Defining a User-defined Object's Interface 19–10
Defining the Object's Buttons in the Object Palette 19–11
Assigning a Help File to the User-Defined Object 19–13
Adding and Registering a User-Defined Object 19–14
Adding a User-Defined Object (without Registering it) 19–15
Registering a User-Defined Object 19–16
Defining a User-Defined Object 19–17
Dynamic and Drag 'n Drop User-Defined Objects 19–17
Choosing Which Properties, Events and Functions to Include in the Application 19–18
Changing a User-Defined Object that is Part of an Application 19–21
More About User-Defined Objects 19–22
Position of Objects within User-Defined Objects 19–23
Objects Generated by User-Defined Objects 19–25
"Visibility" of Objects within a User-Defined Object 19–25
Arrays of Objects and Holders in User-Defined Objects 19–26
The Keyboard and User-Defined Objects 19–27
Recording the Events of User-Defined Objects 19–29
Tutorial: Building A User-Defined Object 19–30
Part 1: Description of the User-Defined Object 19–30
Adding the Objects and Saving the Application as an Object 19–31
Building the Basic Logic 19–31
Adding an Integer Property 19–32
Adding an Event 19–32
Creating the User-Defined Functions 19–33
Defining the User-Defined Object's Class and Button 19–35
Registering and Adding the User-Defined Object 19–36
Using the User-Defined Object's Logic 19–37
Testing the User-Defined Object 19–38
Part 2: Making Changes to the User-Defined Object 19–38

Part 5: Rapid Results

CHAPTER 20: THE PROTOTYPER **20–1**
The Prototyper Window **20–2**
Running the Application **20–2**
Starting the Application **20–2**
Testing the Application **20–3**
Updating the Palette **20–4**
Tracing the Logic **20–4**
Pausing/Resuming the Application **20–5**
Stopping the Application **20–6**
Recording in the Prototyper **20–6**
Recording a Sequence **20–7**
Playing the Recorded Sequence **20–7**
Managing Recorder Files **20–8**
Playing a Saved Recording **20–9**
Editing the Recording File **20–10**
Recording "Special Cases" **20–13**
Externally Controlling the Prototyper **20–15**
Setting up the CBT Channel **20–15**
Rapid in Student Mode **20–16**
Controlling the Prototyper's State **20–17**
Saving the Prototyper's State **20–17**
Loading a State File **20–18**
The State File Format **20–18**
Compressed State File **20–21**

CHAPTER 21: THE REVIEWER **21–1**
Opening and Closing the Reviewer **21–2**
Working with the Application **21–4**
The Reviewer Window **21–4**
Running the Application **21–8**
Restarting an Application **21–11**
Opening a Different Application **21–11**
Recording **21–11**
Controlling an Application's State **21–11**
Using the Reviewer's Command Line Options **21–12**
Adding Options to the Command Line **21–12**
The Command Line Options **21–14**

CHAPTER 22: THE PACKAGER 22–1
Overview 22–2
Getting Started 22–2
Installation 22–2
Preparing the Application for Packaging 22–3
Launching the Packager 22–3
Creating a Package 22–4
Selecting the Application 22–5
Selecting an Icon 22–6
Adding External Files 22–7
Adding Readme Files 22–8
Adding Fonts 22–9
Specifying the Default Directory 22–10
Specifying the Folder Name 22–10
Specifying the Package Build Directory 22–11
Verifying Summary List 22–12
Editing the Package 22–12
Editing Via the .pak File. 22–12
Editing Via the setup.ini File 22–13
Modifying the Reviewer File List 22–14
Instructing the End-user 22–14

CHAPTER 23: PLUG-IN FOR NETSCAPE 23–1
Preparing the Rapid Application 23–2
Packaging the Application 23–2
Preparing the Html Page 23–4
Adding Control Buttons (Optional) 23–6
Controlling the Application Window in all Modes 23–6
Controlling the Application Window in Standalone Mode Only 23–7
Putting the Html Page and Package on the Web Server 23–8
What Happens When the Page is Opened? 23–8

CHAPTER 24: GENERATING REPORTS 24–1
Object Reports 24–2
Object Layout Report 24–2
Object Tree Report 24–3
Object Data Report 24–4
Prototyper Layout Report 24–6
Mode Reports 24–7
State Chart Plot 24–7

Mode Tree Report 24–8
Mode Data Report 24–10
Editing Reports 24–12
Choosing the Report Editor 24–13
Editing Text Reports 24–13

APPENDIX A: GRAPHIC PERFORMANCE IN RAPID A–1
Rapid Bitmap Objects A–1
Display Modes in Windows A–1
Bitmap Formats A–2
Palette Mode vs. True Color Mode A–3
Aspects of Rapid's Graphic Performance A–4

APPENDIX B: CBT INTERFACE MESSAGE SYSTEM B–1
Packet Format B–2
Delimiter B–2
Syntax B–3
Invalid Messages B–3
Message Handler Addresses B–4
Channel Manager B–4
Reviewer/Prototyper Commands B–5
State Manager Commands B–7
Recorder Commands B–7
Event Packets B–8
Event Rerouting B–8
Event Monitoring and External Events B–9
Event Message Structure B–9
Announcements B–11
Response Level B–11

APPENDIX C: WRITING DLLS FOR RAPID C–1

APPENDIX D: RAPID LANGUAGE SYNTAX D–1
Token Specifications D–1
Grammar Specifications D–2
Token Specification for English (1) D–3
Token Specification for Japanese (81) D–4
Token Specification for Korean (82) D–5
Syntax Definition D–6

GLOSSARY G–1

INDEX I–1

ABOUT RAPID'S DOCUMENTATION

There are two editions of Rapid SIMULATION: basic and professional. Both editions include the primary application-development tools required to design and test a Rapid prototype. Rapid SIMULATION has other features, however, that enhance and extend the prototype development platform. The following table summarizes the features included in each edition:

FEATURE	BASIC	PROFESSIONAL
Debugger	+	+
Find and Replace	+	+
State Charts		+
Verification Test		+
Reports		+
Add External Objects		+
Save as UDO		+
Packager		+
Plug-in Packager		+
Object Development Kit (ODK)		+

This User Manual(described below) covers all of the features included in the professional edition of Rapid SIMULATION, except for the ODK, which is documented in a separate manual.

About the User Manual

The User Manual provides the information you need to build Rapid applications. It is divided into five parts:

Getting Started

This part gives information on:

- Opening and saving applications.
- Generating runtime only applications.

- Customizing your Rapid environment.

It also provides general information on Rapid's interface.

Rapid Objects

This part is divided into several chapters; each chapter explains a different aspect of working with Rapid objects:

- Adding and defining objects in the Object Layout.
- Primitive, active and nongraphic objects.
- Customizing objects in the Object Editor and storing them in object libraries.

Rapid Logic

The chapters in this part describe different aspects of building Rapid logic:

- Rapid logic basics; modes, transitions, triggers, actions and activities.
- Building the logic using the Mode Browser, Logic Editor and Logic Chart.
- Advanced logic topics and detailed explanations on objects' properties and functions.

Rapid Reusability

The chapters in this part describe how to build user-defined functions and user-defined objects. Once built, these entities can be reused many times during the application-building process.

Rapid Results

- Running the application in the Prototyper; recording, saving and playing back application sequences.
- Generating reports on the application that give detailed data on its logic and objects.
- Linking your application to an external program, such as an authoring tool.

Getting Online Help

Rapid offers you online Help on procedures, menu commands, dialog boxes and object properties and functions. You can open Help by choosing Help|Rapid Help Topics in any of Rapid's windows.



All of Rapid's dialog boxes have a Help button allowing you to get online explanations on how to use the dialog box.

You can also get context-sensitive Help on a particular item (button, menu command, etc.,) by clicking the Help button in the tool bar and then clicking the item.

❖ *NOTE: To get Help on a menu command or on a function in the Logic Palette, you can also highlight the command or function (with the mouse or using the keyboard arrow keys) and press F1.*

Conventions Used in this Manual

Rapid's documentation uses the following conventions:

Menu conventions

- "Choose File|Save Application" means to select the Save Application command from the File menu.

Mouse conventions

- Rapid makes use of the left and right mouse buttons. "Click" means to click the **left** mouse button; "right-click" means to click the **right** mouse button.
- "Shift-click" means to hold down the Shift key and click the left mouse button. Similarly, "Ctrl-click" and "Alt-click."

Keyboard conventions

- "Ctrl+C" means to hold down the Ctrl key, while pressing C. Similarly, "Ctrl+O," "Ctrl+V," etc.

Typographic conventions

- Names of properties, functions and events are italicized. For example, *connect* and *blinkPeriod*.

Complete phrases of Rapid logic are presented in bold, sans serif text:

& Switch2.position3 is connected

CHAPTER 1

Starting Rapid

This chapter explains a few basic procedures for getting started with Rapid and managing your applications. The topics covered are:

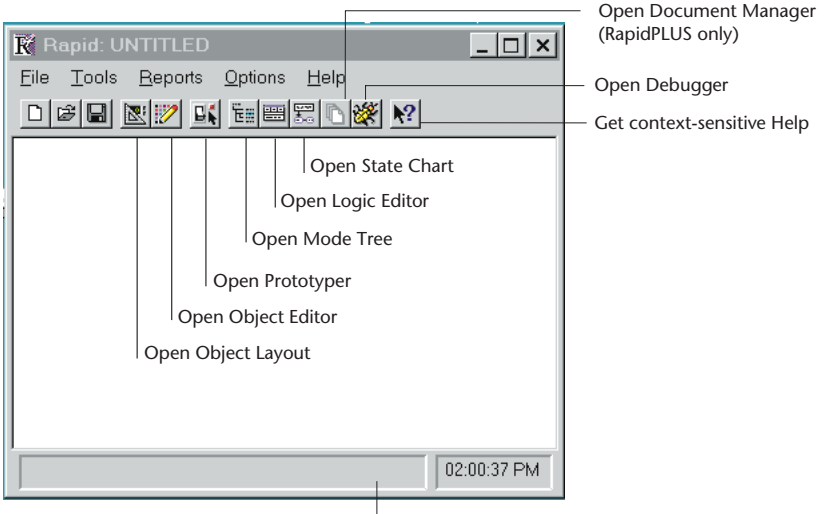
- Opening Rapid.
- The Application Manager window.
- Creating, opening and saving applications.
- Backup applications.
- Generating a runtime application.
- Opening Rapid's application-building tools (Object Layout, Mode Tree, etc.).
- Customizing your Rapid environment.

OPENING RAPID

To open Rapid:

- Double-click the Rapid application icon.

While Rapid loads, the Rapid logo is displayed on the screen. Rapid's Application Manager window opens as shown below.



Application Manager window

The Application Manager serves several functions, including:

- Managing Rapid files (opening and saving Rapid applications, etc.).
- Opening Rapid's application-building tools (Object Layout, Object Editor, Mode Tree, Logic Editor, State Chart and Prototyper).
- Setting general parameters (fonts, scale, etc.)
- Generating printed or screen reports on various aspects of the application. See Chapter 24: "Generating Reports" for information on Rapid reports.

WORKING WITH RAPID APPLICATIONS

The File menu allows you to perform, among other options, the following file management functions:

- Open an existing or new application.
- Save the current application with either the current name or a new name.
- Load the backup of any application.
- Save the application as a runtime application.
- Exit Rapid.

Creating a New Application

To create a new application:



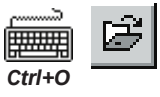
- Choose File|New, or click the New Application button.

  The current application is cleared, the various Rapid windows remain at their current configuration, and the window titles change to: “Rapid: Untitled.”

  If the current application contains unsaved changes, Rapid reminds you to save any changes before starting a new application.

Opening an Existing Application

To open an existing application:



- Choose File|Open, or click the Open Application button.

  Rapid clears the current application, and replaces it with the existing application. If the current application contains unsaved changes, Rapid prompts you to save.

Rapid .tmp Files

Whenever you open a new or existing Rapid application, Rapid generates a temporary file. This file is placed in the “temp” folder as defined in your system. If you open a different application, Rapid replaces that *.tmp* file with another; when you exit Rapid, the file is deleted.

If, for some reason, you do not exit Rapid smoothly, the *.tmp* file will **not** be deleted. Since the *.tmp* file can be very large (depending on the size of the

Rapid application that you opened), this might use up a lot of disk space for no obvious reason; delete the *.tmp* file manually from your "temp" directory.

Saving an Application

To save an application:



- Choose File|Save, or click the Save Application button.

  When Rapid saves the application, it saves the previous version as a backup which can be recovered using the Load Backup command (see below). Backup applications have the extension *.$pd*.

  If the application is untitled, this option performs identically to the Save As command (see below).

  It is recommended that you do not save applications directly to a floppy disk, as temporary files created during the save may fill it. Rather, first save the application to the hard disk and then copy to the floppy disk.

  If you delete objects, bitmaps or logic from an application, its file size is not affected when you perform a normal save. To reduce the file size, you must use the Save As command.

To save an application with a new name:



- Choose File|Save As.

  This command is identical to Save, except that Rapid prompts you to supply a file name for the new copy of the application. If an application with the supplied name already exists, you are prompted to either cancel the save command or overwrite the previous application.

  When you save an application under a new name, the current application name changes to that new name.

Loading a Backup Application

To load a backup application:

- Choose File|Load Backup.

  This command loads the penultimate saved version of the application. (You load the last, most recently saved version of the application with the File|Open command.)

Generating a Runtime Application

Rapid allows you to generate a runtime version of your application which can only be opened in the Rapid Reviewer. This application, with the extension, *.rvr*, is a "read-only" file whose layout and logic **cannot** be changed.

❖ *NOTE: The Rapid Reviewer can open both .rvr and .rpd files.*

To generate a runtime application:

- Choose File|Generate Runtime. Rapid saves the application under its current name, with the extension .rvr.

  If you have not previously saved the application, the Save Application As dialog box opens. Rapid saves the application as both a runtime file (i.e., *.rvr*) and a regular file (i.e., *.rpd*).

Exiting Rapid

To exit Rapid:

- Choose File|Exit Rapid.

  If you have changed the current application since last saving it, or if the application is still untitled, a dialog box opens which prompts you to save unsaved changes.

OPENING RAPID TOOLS

You open Rapid's application-building modules from the Application Manager. Choose Tools|<tool name> to open that window.

TOOL	BUTTON (SHORTCUT KEY)	DESCRIPTION
Object Layout	 (Ctrl+A)	Use the Object Layout to add and define the objects that make up the application's interface (e.g., switches), as well as any nongraphic objects that you may require (e.g., timers).
Mode Tree	 (Ctrl+T)	Use the Mode Tree to define the application's tree of modes and the transitions between the modes.

TOOL	BUTTON (SHORTCUT KEY)	DESCRIPTION
Logic Editor	 (Ctrl+L)	Use the Logic Editor to create or modify transitions between modes, activities, triggers, and actions.
State Chart	 (Ctrl+C)	Use the State Chart to display a bird's-eye view of the application modes and transitions.
Object Editor	 (Ctrl+B)	Use the Object Editor to create and edit certain kinds of objects.
Prototyper	 (Ctrl+R)	Use the Prototyper to run and test your application's behavior.
Debugger	 (Ctrl+G)	Use the Debugger to pause the application at specified breakpoints and/or execute the logic step by step.

CUSTOMIZING YOUR APPLICATION

There are several options that allow you to customize features of Rapid's working environment:

- Changing Rapid's system font.
- Setting the text editor used for editing reports. (This option is described in Chapter 24: "Generating Reports.").
- Setting the scale of the next new application.
- Showing or hiding tool tips.
- Saving settings.

Changing Rapid's Font

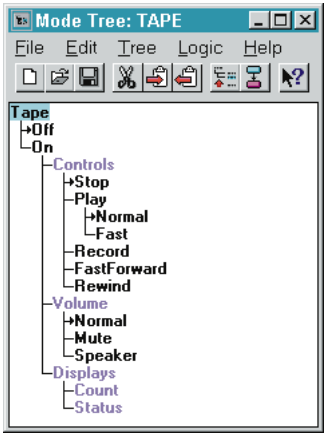
Rapid allows you to change the font for the system text which appears in the following windows:

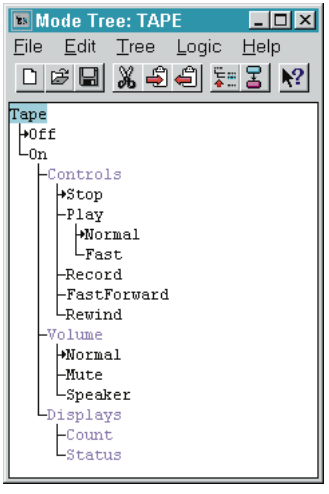
- Mode Tree
- Logic Editor
- State Chart's Logic Information Pane

❖ *NOTE: The text in the menus and dialog boxes are not affected by changes made through this dialog box.*

To change Rapid's font:

1. Choose Options|Font to open the Font dialog box.
2. Set the font, size and style, then click OK. Rapid redraws any open windows. An example of a font change is shown below:





HINT:
To apply the user-defined font (as it appears in the mode tree, tool tips, etc.) to edit boxes (such as the Logic Editor edit line), hold down the Ctrl key while you choose File|Exit Rapid.
To re-apply the Rapid default font (MS Sans Serif, 8 points, bold) to edit boxes, press Ctrl+Shift+File|Exit Rapid.

Setting the Default Scale

Before you start a new application, you can set its scale. For example, if you are designing an aircraft's control panel, you might set your scale to 3 feet. Alternatively, if you are designing a telephone, you might set your scale to be

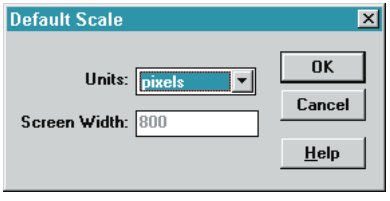
20 centimeters. This feature allows you to include the physical specifications of an application at the initial design.

An application's scale affects all aspects of the objects' sizes and positions. For example, in an application with a scale of centimeters, the objects' sizes are given in centimeters; the activity *Circle changeXBy: 1* will change Circle's x position by 1 cm., etc.

❖ *NOTE: You cannot change the scale of an existing application (even if the application is still untitled). You can only set the scale units for your* ***next*** *new application.*

To set the default scale:

1. Before starting a new application, choose Options|Default Scale; the following dialog box opens (possibly with a different workspace width, depending on your monitor resolution):



2. Click the Units list box arrow and select a unit. You can choose from:
   - Centimeters (cm)
   - Feet (ft)
   - Inches (inch)
   - Kilometers (km)
   - Meters (m)
   - Miles (ml)
   - Millimeters (mm)
   - Pixels (pxl)
   - Yards (yd)

   The characters in parentheses after each unit show how that unit is represented in the Object Layout's parameter panes.
3. Type a measurement in the Screen Width text box.

❖ *NOTE: You must select the units before entering a width measurement.*

Once you have set the default scale, it applies to all future applications that you create until you reset it, or until you exit Rapid.

To retain this default scale for your next Rapid session, choose Options|Save Settings on Exit (see below). If you check the Save Settings option, the default scale for your next Rapid session will be the scale you have just set.

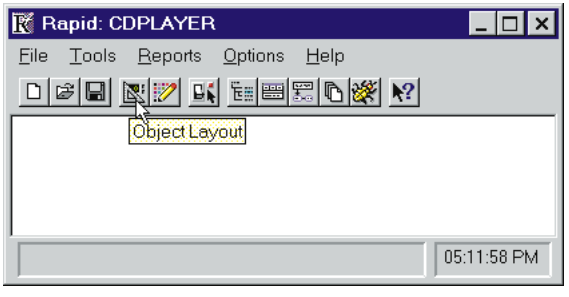
If you do not check the Save Settings option, the default scaling for your next Rapid session will be the same scale as at the start of your current session.

❖ *NOTE: When you install Rapid, the default scale is in pixels.*

The default scale setting is saved in the RAPIDxx.INI file along with other Rapid settings. You can adjust the setting directly by editing RAPIDxx.INI accordingly.

Tool Tips

A tool tip is a description of a button or other control that appears alongside the cursor when the mouse is positioned over it. Tool tips give you a convenient, quick reminder of the function of each tool button, list box, etc.



To view tool tips:

- Choose Options|Tool Tips.

  By default, Rapid shows tool tips.

Saving Settings on Exit

To save settings on exit:

- Check Options|Save Settings on Exit.

When this command is checked, various system settings are saved in the RAPID40.INI file and become system defaults every time you open Rapid. These settings include window sizes and positions, fonts, scale units, etc. For a full listing of the saved parameters, open your Windows Explorer (or File Manager) and double-click the RAPID40.INI file in the Win95 (or Windows) directory.

HINT:
If you hold down the Shift key when you choose File|Exit Rapid, Rapid saves the current settings **without** exiting.

Developing your own Rapid desktop

As you work with Rapid, you will find that it is often convenient to have windows always at a certain size and position. The Mode Tree, for example, is usually used together with the Logic Editor. As you develop your own preferences, save the settings of individual windows. (Remember the Shift-Exit method!).

CHAPTER 2

The Object Layout

The Object Layout is a comprehensive tool for creating a visual representation of an application's layout. You use the Object Layout to add and define the graphic and nongraphic elements that comprise the application that you are developing. In Rapid terminology, these elements are called "objects." You can modify the objects as often as necessary in the Object Layout until the layout meets the application specifications.

In this chapter you will read about:

- The different categories of Rapid objects.
- The behavior of objects.
- Customizing the Object Layout.
- Adding graphic objects to the layout.
- Modifying an object's basic parameters.
- Manipulating objects on the layout.

For information on how to add and work with nongraphic objects, see Chapter 5: "Nongraphic Objects."

For information concerning specific graphic objects, see Chapter 3: "Primitive Objects" and Chapter 4: "Active Objects."

WHAT IS AN OBJECT?

Objects are the basic building-blocks of all Rapid applications. Objects are either graphic (visual) or nongraphic ("behind the scenes") elements that you add to an application in the Object Layout. Rapid objects imitate their real-life counterparts.

Rapid objects have pre-defined sets of properties and functions that describe how they behave in real time. For example, a real lamp can either be on or off, so a Rapid lamp has the two functions, *on* and *off* (it has other functions as well which are explained in the *Object Reference* manual).

Types of Objects

There are three types of objects:

- **Active objects**

  Graphic elements that play an "active" role in an application's behavior. Active objects can be manipulated in the Prototyper. For example, the knob on a Rapid rotary switch can be turned with the click of the mouse.

- **Primitive objects**

  Graphic elements such as lines, frames, text labels and imported bitmaps that enhance a layout's appearance. Primitive objects are usually passive, that is, they cannot be manipulated in the Prototyper.

  Primitive objects do not have properties and functions, and therefore cannot interact with other objects. You can, however, make a primitive object "active" so that it behaves like an active object.

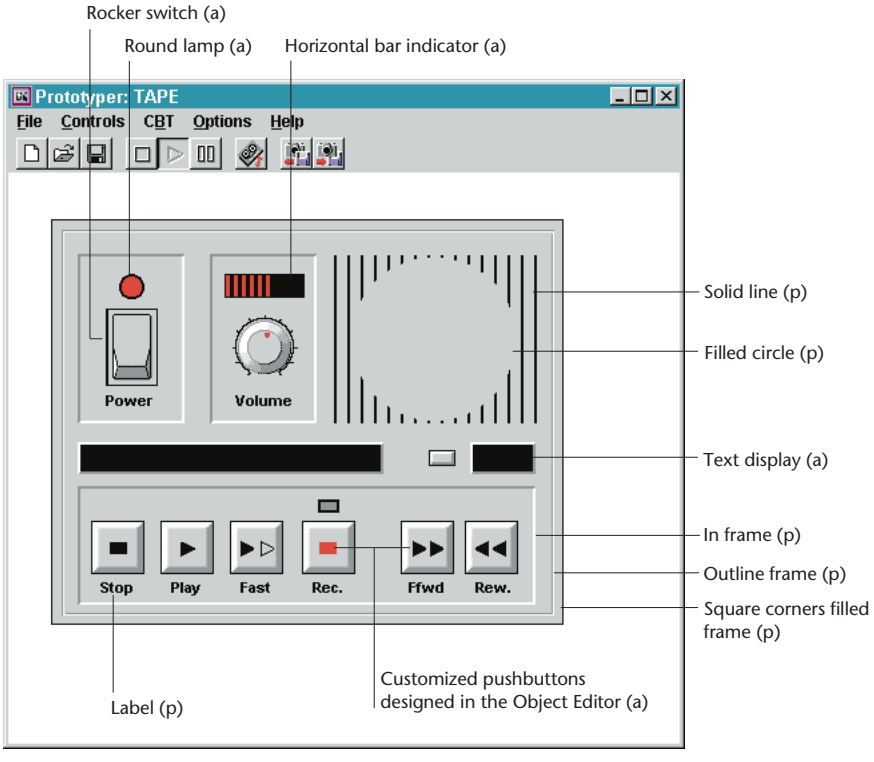
- **Nongraphic objects**

  Non-visual elements that play important "behind the scenes" roles in an application's behavior. For example, a timer does not appear on the layout, but it can be used to cause a lamp to turn on.

  You create interaction between nongraphic and active objects by constructing logic phrases in the Logic Editor.

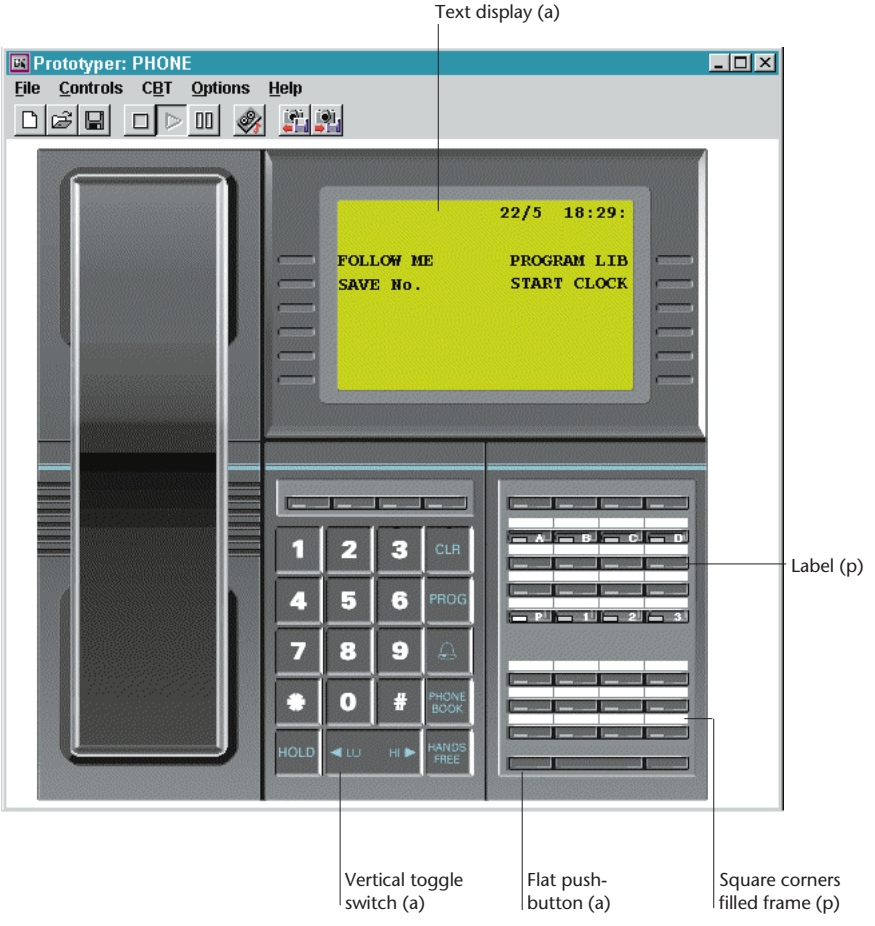
Sample Applications

Below are examples of applications created in Rapid. Some of the objects are pointed out in order to show you how a layout can be developed. The (a) indicates an active object; the (p) indicates a primitive object. The non-graphic objects are listed below the applications.

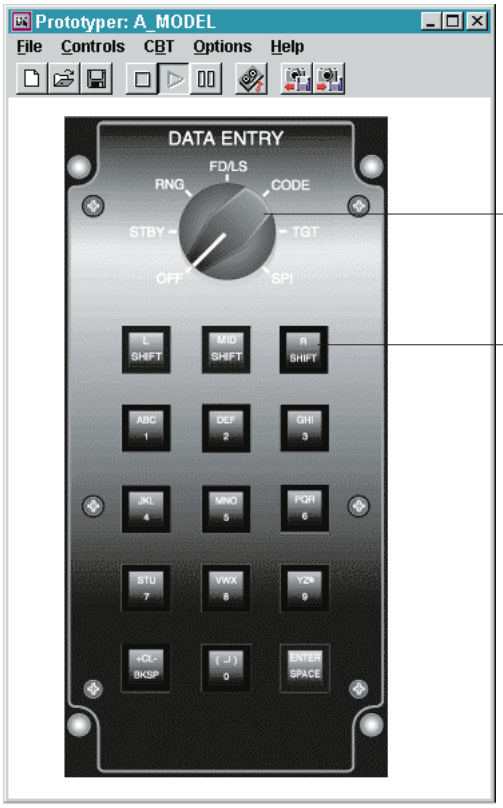


Nongraphic objects: Stopwatch, String, Sound

In the following three applications, bitmaps were imported into Rapid. The imported bitmaps are primitive objects that are not used in the logic. The active objects, customized in the Object Editor, were then placed on top of the bitmaps.



Nongraphic objects: Data store, Number, Stopwatch, String, Sound, Timer

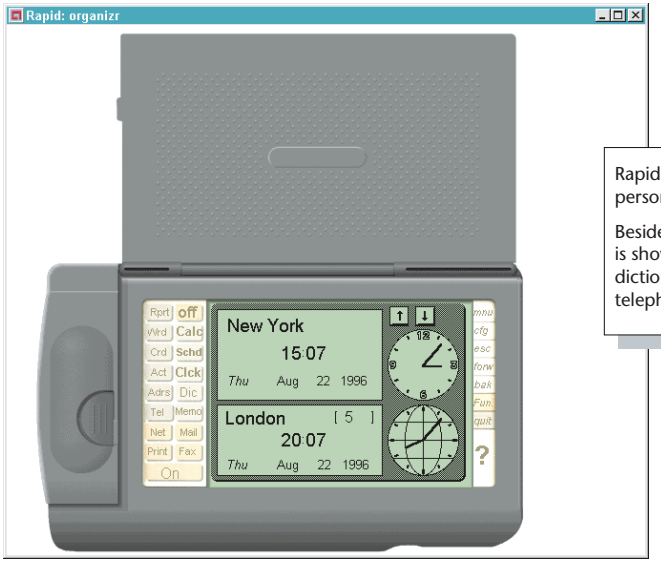


Rotary switch customized in the Object Editor (a)

Flat pushbutton (a)

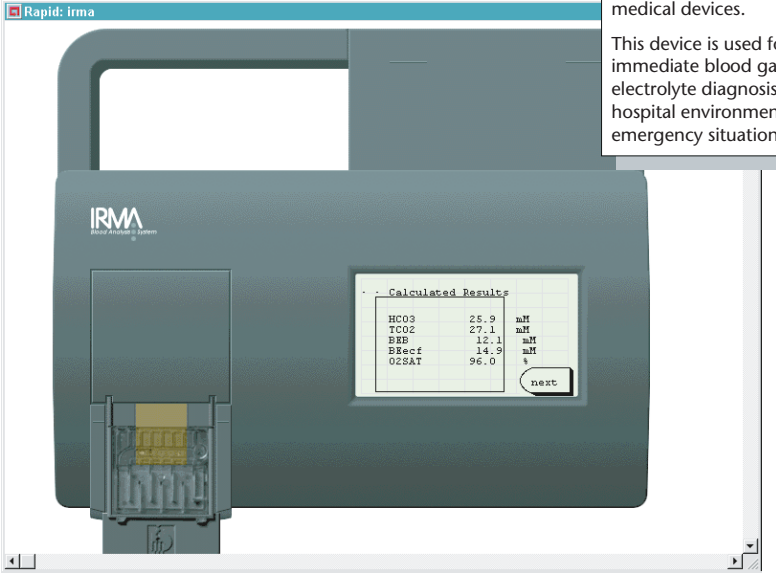
Nongraphic objects: Sound, String, Time

The applications shown below demonstrate Rapid's versatility:



Rapid was used to create a personal organizer.

Besides the world clock that is shown, it has a calculator, dictionary, holds addresses, telephone numbers, etc.



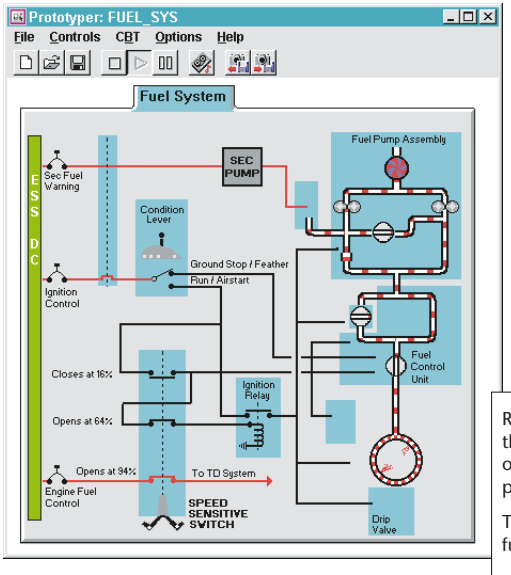
Rapid can be used to create medical devices.

This device is used for immediate blood gas and electrolyte diagnosis in hospital environments and emergency situations.



Rapid can be used for computer-based training.

This simulation allows you to recreate the navigating and targeting abilities of a fighter jet.



Rapid allows you to simulate the storage and movement of a substance along a set path.

This example illustrates the fuel system of an aircraft.

Behavior of Objects

Each object has its own characteristics that allow you to control its behavior. In this section you will read about:

- Groups, or classes, of objects
- Object properties and functions
- Dynamic and Drag ‘n Drop functions

Classes

All objects are grouped into classes according to their similarities. For example, Timer, Stopwatch, Date and Time nongraphic objects are grouped into the Time class.

Properties

Every active and nongraphic object has a defined set of properties that describes how it behaves in real-time. Each property controls a specific aspect of the object’s behavior. For example, the lever of a real 2-position, horizontal toggle switch can be in either the left or right positions; the equivalent Rapid switch has the properties, *left* and *right*.

Objects also have a *self* property which refers to the object as a whole.

Functions

Every property has functions. Functions are commands that either adjust a property’s value or adjust the property itself.

Functions that adjust a property’s value

Most properties have a value that is either a number or a string. For example, a Rapid lamp can be designed to blink at regular intervals. It has the property, *blinkPeriod*, which sets the period of time the lamp blinks.

The value of *blinkPeriod* might be 500 msec., i.e., the lamp will blink at 500 millisecond intervals. *blinkPeriod* has many arithmetic functions (such as *changeBy:*) which allow you to increase or decrease the lamp’s blink period.

Functions that adjust the property itself

The toggle switch’s *left* and *right* properties each have the function, *connect*. When the *left* property receives the *connect* function, the lever moves to the

left position; when the *right* property receives the *connect* function, the lever moves to the right position.



(Switch1. left connect)



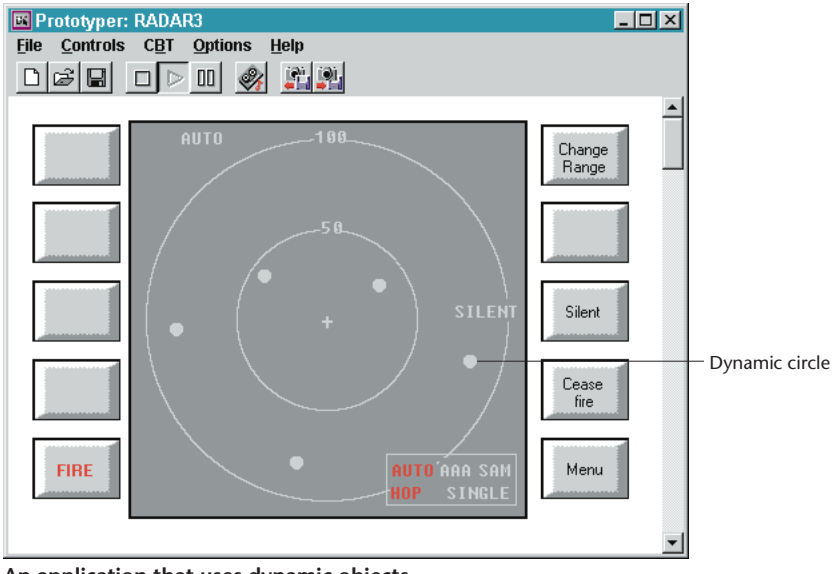
(Switch1.right connect)

Dynamic objects

Dynamic objects let you introduce motion into applications. A dynamic object is a graphic object whose position can be moved in the Prototyper in a pre-determined path. You determine this path in the application's logic.

Making a graphic object dynamic adds properties and functions to its logic. A dynamic primitive object can also be resized and have its color changed in the logic, as well as generate an event when clicked in the Prototyper.

The illustration below uses dynamic circles to simulate moving targets across a radar screen:



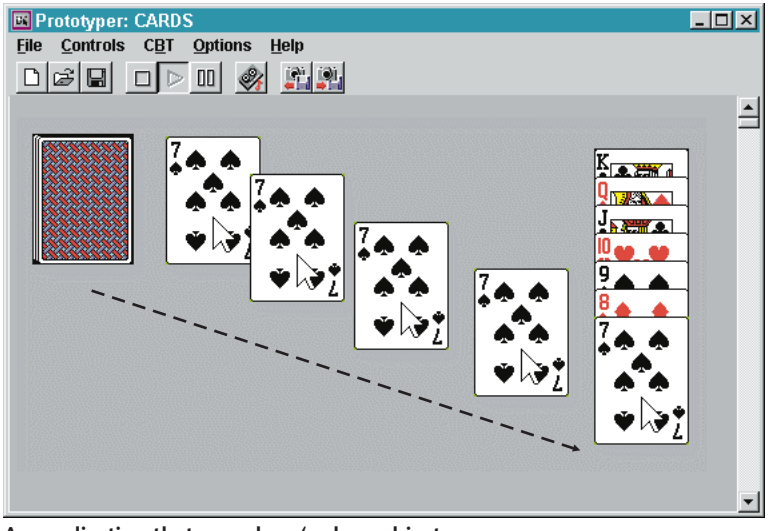
An application that uses dynamic objects

To make an active object dynamic, see p. 2–20. To make a primitive object dynamic, see p. 3–11.

Drag ‘n drop objects

A drag ‘n drop object is a graphic object that can be dragged in the Prototyper using the mouse. Making an object drag ‘n drop adds events to the object’s logic.

The illustration below shows a (bitmap) card as it is being dragged and dropped onto another card in the Prototyper.



An application that uses drag ‘n drop objects

To make an active object drag ‘n drop, see p. 2–20. To make a primitive object drag ‘n drop, see p. 3–11.

Hierarchy of Active Objects

The active objects in an application are organized in a hierarchical manner; therefore, every object that you create must be attached to an existing object. The new object is referred to as the "child" while the existing object is its "parent." This hierarchical organization has a direct effect on the application's behavior. In the Prototyper, for example, if a parent moves or is hidden or displayed, its children follow suit.

The following rules apply in the Object Layout hierarchy:

- Parents can have more than one child, but children have one parent only.
- The default parent is the application itself which is created as an object when you begin a new application. The application is referred to as the "root object."
- You can change an object's parent, but the combination of object name and parent name must be unique, i.e., Rapid does not accept siblings with the same name.
- If you edit the name of a parent object which has children, Rapid changes the childrens' "Parent" parameter to reflect this change.

Object Names

Rapid names active and nongraphic objects when you place them on the layout. Each name consists of the object class and a sequential number. The first object of a class that you add is numbered one, the second object of that class, two, etc.

For example, if you place objects on the layout in the following order: a vertical toggle switch, a data store and a horizontal toggle switch, the objects are named "Switch1," "Data_Store1" and "Switch2," respectively.

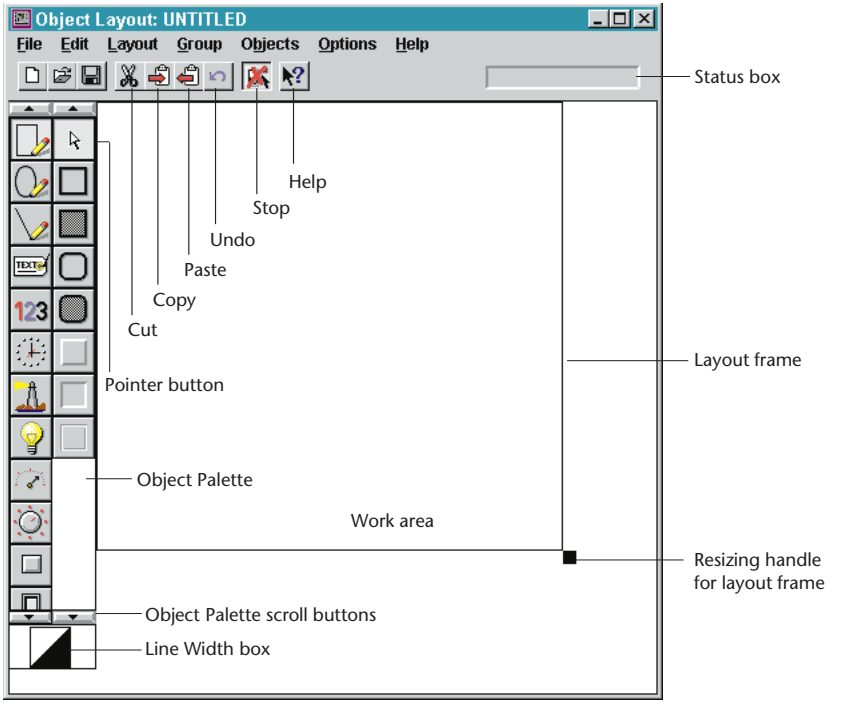
Deleted objects do not decrease the numbering. For example, if you delete the first toggle switch, the data store and the second toggle switch keep their original numbers, and the next switch added to the layout will be assigned the number, three.

THE OBJECT LAYOUT WINDOW

To open the Object Layout:



- Choose Tools|Object Layout from the Application Manager or click the Object Layout button. Its window is set up as shown below:



CUSTOMIZING THE OBJECT LAYOUT

The Object Layout options let you customize the working environment to suit your preferences and requirements. The following options are available:

- Resizing the layout frame.
- Hiding or displaying the Object Palette.
- Enabling a snap-to-position grid.
- Hiding or displaying the mouse coordinates in the status box.

Resizing the Layout Frame

The layout frame defines the boundaries of the application under development. You can resize the layout frame in order to enlarge or reduce the size of the work area.

To resize the layout frame:

1. If necessary, first enlarge or maximize the Object Layout window.
2. Drag the frame's resizing handle until the frame is the size you want.

Hiding/Showing the Object Palette

When you first open the Object Layout, the Object Palette is displayed (and the Options|Show Object Palette command is checked). If you hide the Object Palette, the layout frame shifts to the left of the Object Layout window, thereby increasing the work area.

To hide the Object Palette:

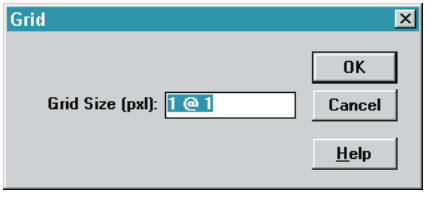
- Choose Options|Show Object Palette. The menu command is not checked when the Object Palette is hidden.

Using the Grid

The Object Layout has an invisible grid that allows you to "snap" objects to it when you add, move or size them. The grid is useful for neatly laying out objects without having to constantly align them. The default grid size is 1@1 pixels (or the equivalent measurement in your default scale units), which means that effectively no grid is enabled.

To change the grid size:

1. Choose Options|Grid to open the Grid dialog box:



The current size is shown in the Grid Size text box, in the following format: "x@y" where x is the grid size in the horizontal direction and y in the vertical direction. The default scale units are shown in parentheses.

2. Overtype the grid size, then click OK.

❖ *NOTES: If you only provide a single number in the text box, it will be used for both the x and y grid sizes. For example, typing "20" is the same as typing "20@20".*

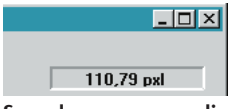
The grid also affects resizing the layout frame.

Showing Mouse Coordinates

When the Options|Show Mouse Coordinates command is checked, the mouse position coordinates are displayed in the status box on the right side of the toolbar. When the command is not checked, the status box is empty.

To display mouse coordinates:

- Choose Options|Show Mouse Coordinates; a check mark appears next to the command and the mouse coordinates are displayed:



Sample mouse coordinates

ADDING GRAPHIC OBJECTS

General Procedure

There are three steps in adding a graphic object to the layout:

a. Selecting the object.

b. Placing the object on the layout.

c. Modifying the default values of the object's parameters in the object's parameter pane (optional).

These steps are described below.

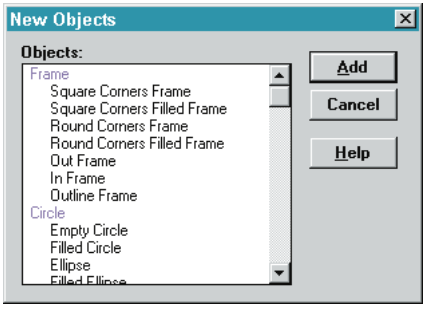
❖ *NOTE: To add a label object, see "Adding a Label Object" in Chapter 3.*

Selecting a Graphic Object to Add

In both the New Objects dialog box and the Object Palette, the object types are organized into classes according to their functional or graphic characteristics. For example, all of the switches are grouped together in one class, all the frames are grouped together in another class, etc.

To select an object from the New Objects dialog box:

1. Choose Objects|Add to see a listing of the objects by class:



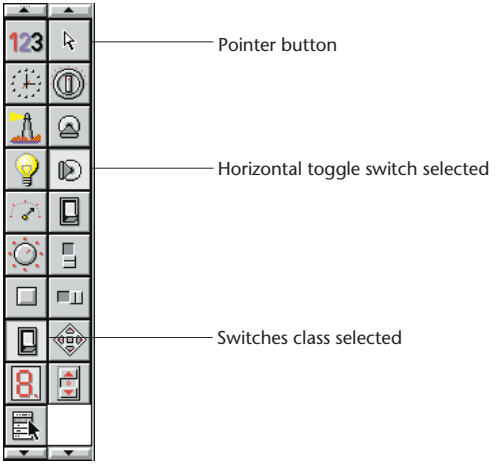
2. Select an object and click the Add button.

To select an object from the Object Palette:

The object class buttons are located in the left column of the Object Palette; the object type buttons are located in the right column. The button of the currently active object class is pressed in and the object types that make up this class are displayed.

Use the scroll buttons at the top and bottom of the Object Palette to access object classes or types that are not visible.

1. To select an object class, click its button in the left column. The class's objects are displayed in the right column, and the first object is selected.
2. To select an object, click its button in the right column (see illustration below).



To deselect the object class:

- Select a different class button.

To deselect the object type:

Either:

- Select a different object.

Or:

- Deselect the class.

To cancel the add object operation:

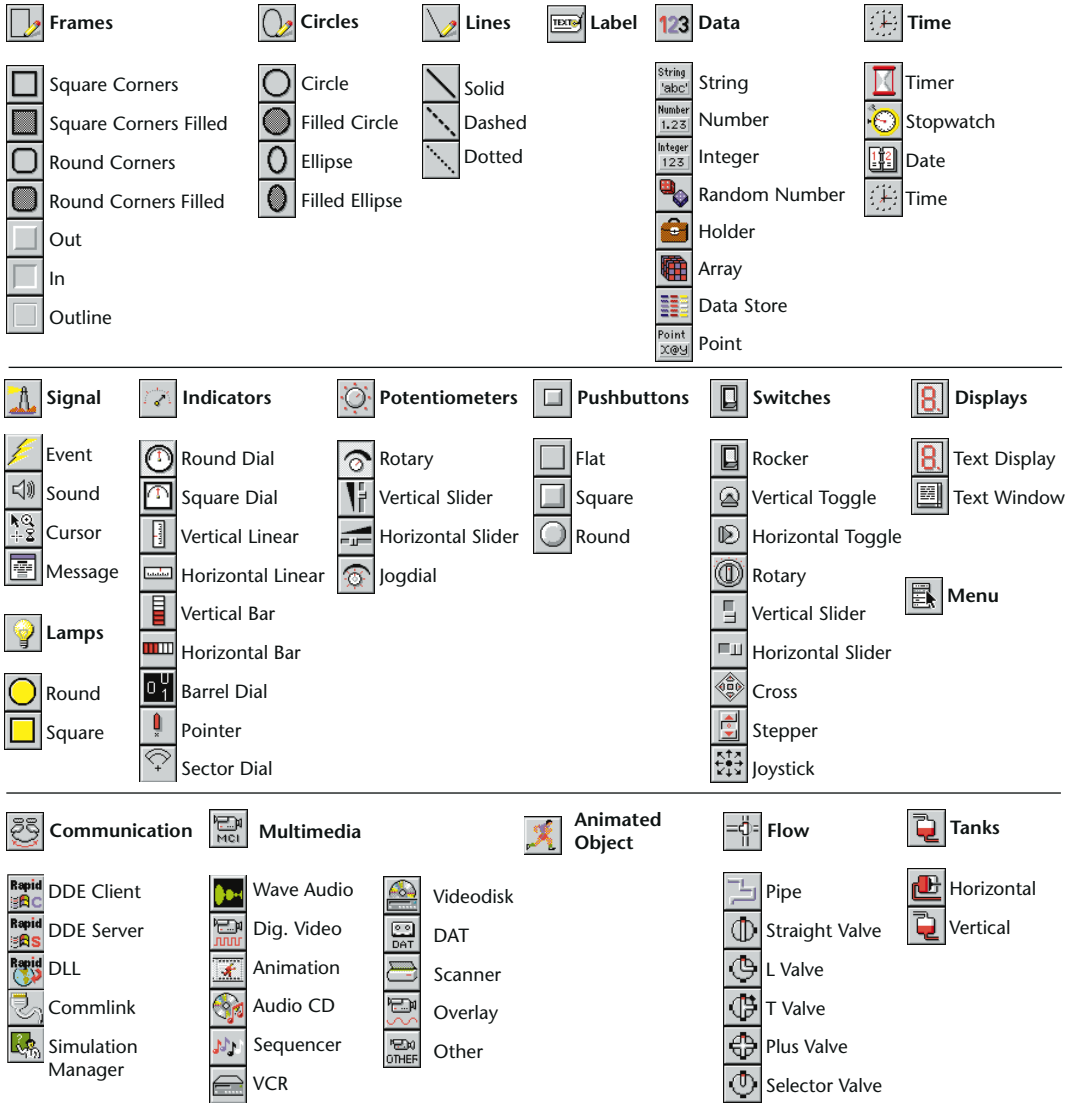
Either:

- Click the Pointer at the top of the Object Palette's right column.

Or:

- Choose Objects|Cancel Adding. This command is particularly useful when the Object Palette is hidden.

The full range of class and object icons is illustrated on the next page:



The class and object icons

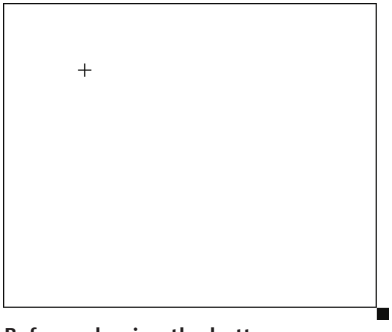
Placing a Graphic Object

After you select an object, the cursor changes to a crosshairs when moved to the work area. The next step in adding the object is to place it on the layout.

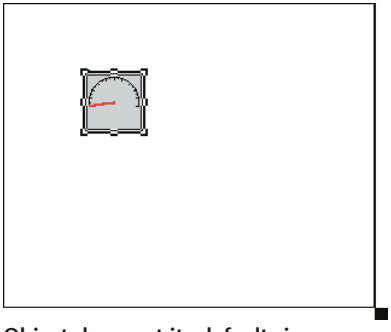
You can either place the object at its default size or resize it as you place it.

To place a graphic object at its default size:

1. Place the crosshairs where you would like the object to be drawn on the layout. The crosshairs marks the upper left corner of the object's enclosing rectangle (within which the object will be drawn).
2. Click the left mouse button. The object is drawn on the layout.



Before releasing the button

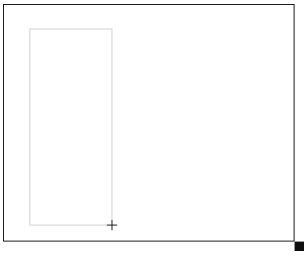


Object drawn at its default size

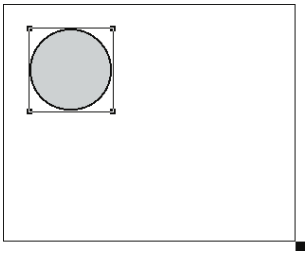
To place a graphic object and resize it simultaneously:

1. Place the crosshairs at the position where you would like to start drawing the object. The crosshairs marks one corner of the object's enclosing rectangle (within which the object will be drawn).
2. Click and drag the enclosing rectangle that appears on the screen. When you are satisfied with the size of the rectangle, release the mouse button.

❖ *NOTE: A circular object's size is constrained to the height or width of the enclosing rectangle, whatever is the smaller. An elliptic object (ellipse, filled ellipse) is tangential to all sides of the enclosing rectangle.*



Before releasing button



Drawn circle after releasing the button

Continuously adding the same type of object

The Objects|Continuous Adding command allows you to continuously add the same type of object without having to reselect the type each time you add an object. When this command is disabled, the object type is deselected every time you place an object on the layout.

To continuously add objects:

1. Make sure that Objects|Continuous Adding is checked.
2. Select the desired object class and type.
3. Place the object on the layout (see “Placing a Graphic Object” above).
4. After placing the object, the cursor remains as a crosshairs, allowing you to place another object of the same type.

To stop adding objects of the same type:

- Select a different object class or object type; or
- Click the Pointer button in the Object Palette; or
- Choose Objects|Cancel Adding.



Ctrl+K

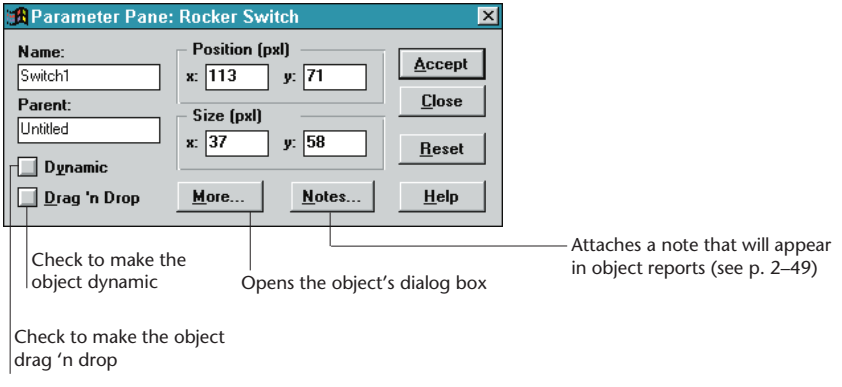
MODIFYING AN OBJECT'S PARAMETERS

Once you have added an object to the layout (as described in "Adding Graphic Objects" on p. 2–14), you can use it in the application's logic. However, you might want to customize its name, change its behavior, etc. To do so you must use the object's parameter pane.

Rapid provides a parameter pane that is used by all graphic objects. As objects are selected on the layout, the information in the parameter pane changes to reflect the parameters of the selected object. You can open object-specific dialog boxes from the parameter pane.

To modify an object's parameters:

1. Double-click the object on the layout to open its parameter pane (or click it once if the parameter pane is already open).



2. Modify the parameters as necessary. Be sure to click Accept when you have finished, otherwise the changes will be lost.

   For convenience, you can leave the parameter pane open as you modify objects, being careful to click Accept after each modification.

BUTTON	DESCRIPTION
Accept	Confirms changes to the object's parameters. The object is redrawn on the layout (the parameter pane remains open).
Reset	Abandons the latest modifications (if they have not been accepted). The parameter pane remains open.

Basic Object Parameters

Every graphic object has the following four parameters:

- Name (optional for primitive objects)
- Parent
- Position/Center/Start
- Size/Radius/End

The parameters are described below. For a detailed discussion of which parameters apply to each object and their default values, see Chapter 3: "Primitive Objects" and Chapter 4: "Active Objects."

Name

The object's name is used to identify it in the Object Layout and Logic Editor. By default, the object name consists of the object class and a sequential number, e.g., Potentiometer1, Switch2, etc. You might want to give objects their own, individual names in place of the default names. For example, instead of "Switch1," you might want to call it "OnSwitch."

Primitive objects do not require names, and therefore do not receive default names when they are added to the layout. (See Chapter 3: "Primitive Objects" regarding the behavior of primitive objects.)

To add or change an object's name:

- Overtype or revise the name in the Name text box, then click Accept.

When you edit object names, Rapid checks for illegal names and displays error messages, if necessary.

Rules for naming objects

- Object names are limited to 64 characters.
- Object names must begin with a letter.
- Illegal characters for an object name are: @ & * ( ) - + = { } [ ] ; : , < > . ? / and [space]. All other characters are legal. If Rapid detects an illegal character, it will replace each illegal character with an underscore (_).
- Rapid distinguishes between capital and lowercase letters, so you can have one object called "Switch" and another called "switch".
- You cannot give any object the same name as the root object.

- Objects can have the same name as long as they have different parents.
- The following words are "reserved," that is, you cannot use them as names of objects: *action*, *and*, *begin*, *clear*, *end*, *entry*, *exit*, *has*, *internal*, *is*, *mode*, *not*, *or*, *resetValue*, *self*, *subroutine*, *x*, *y*.

Parent

By default, an object's parent is the application itself. However, any other graphic object on the layout can be made the object's parent, as long as it has a name. An unnamed primitive object, therefore, cannot be a parent.

To assign an object as the parent:

1. Type the object's name directly into the Parent text box, taking care to duplicate upper and lower case.

   SHORTCUT
   Alt-click the object in the layout that you want to be the parent. A frame appears around the object, and its name appears in the Parent text box.

2. Click Accept.

Position/Center/Start

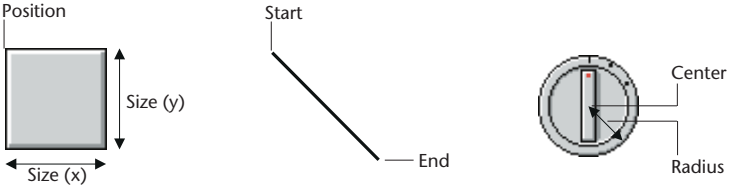
For non-circular objects, the object's **position** is presented in x and y values, where x and y are the coordinates of the upper left corner of the object's enclosing rectangle. Ellipses are non-circular objects, and therefore have the position parameter.

For circular objects, the x and y coordinates pinpoint the object's **center**. For lines, the x and y values represent the object's **starting position**.

The position/center/start values are given in the application's scale units.

To change an object's position/center/start:

- Overtype the numbers in the x and y text boxes, then click Accept.



Size/Radius/End

For non-circular objects, the object's **size** is presented in x and y values, where x is the width of the object's enclosing rectangle and y is its height. Ellipses are non-circular objects, and therefore have the size parameter.

For circular objects, the size is presented by the **radius** value. For lines, the x and y values represent the object's **end position**.

The size/radius/end values are given in the application's scale units.

To change an object's size/radius/end:

- Overtype the numbers in the x and y text boxes, then click Accept.

Building an application's layout

There are two general methods for building an application's layout:

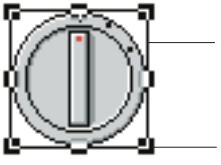
- Add all the objects (accepting Rapid's automatically-assigned names), quickly building the application's layout. At a later stage, go back, if necessary, to rename objects with suitable names and redefine object parameters.
- Build the application's layout methodically, naming and defining each object as you add it.

WORKING WITH GRAPHIC OBJECTS

This section provides step-by-step instructions for working with objects after you have placed them on the layout.

Selecting and Deselecting Graphic Objects

Many of the Object Layout operations (deleting, copying, moving, sizing objects, etc.) are performed on "selected" objects. A typical selected object is illustrated below:



To select a single object:

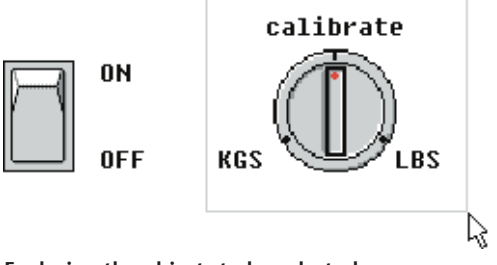
- Click inside the object's enclosing rectangle. Any objects which were previously selected are deselected.

To select several objects:

- Shift-click the additional objects.

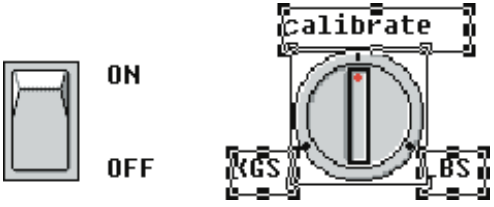
To select multiple objects in one operation:

1. Click-hold at one of the corners of an imaginary rectangle which would enclose the objects you want to select; all other objects are deselected.
2. Drag the mouse to the diagonally opposite corner, until the gray rectangle being drawn on the screen completely encloses the desired objects.



Enclosing the objects to be selected

3. Release the button. All the objects which are entirely within the rectangle are selected.



Objects selected

❖ *NOTE: To avoid deselecting objects that are already selected on the layout, press the Shift key before carrying out this operation.*

To select an obscured object:

1. Place the cursor over the top, obscuring object and press the Tab key. The obscuring object is selected.
2. Each time you press Tab (without moving the mouse) the next lower object is selected—until the top object is selected again.
3. When an obscured object is selected, you can choose Layout|Move to Front, to bring it to the top, or you can use the x- and/or y-position parameters in the parameter pane to move the object.

To move an obscured object:

1. Click and hold the top object.
2. Press Tab until the obscured object is selected, without releasing the mouse button.
3. Drag the object.

❖ *NOTE: If more than one object is obscured, performing this procedure with the parameter pane open helps you to differentiate between the objects.*

Deselecting objects

To deselect one object when several objects are selected:

- Shift-click the object.

To deselect all objects:

- Click an empty space in the layout.

Selecting all active objects

To select all active objects on the layout:


Ctrl+Y

- Choose Edit|Select all Active. This command deselects any non-active primitive objects which were selected.

Automatically selecting an object's children

When you select an object, only that specific object is normally selected. There are some occasions, however, when you will want to automatically select an object's children whenever you select the parent object.

Example:

You select a parent object and its children using the Shift-click method. If you then copy them, Rapid copies the individual objects, without retaining the family hierarchy.

If you want to retain the hierarchy when you copy, you should tell Rapid to automatically select the children when you select the parent.

To automatically select an object's children:

1. Choose Options|Select Children; a check mark appears next to the command.
2. Select the parent object; the parent object and all of its children are selected.

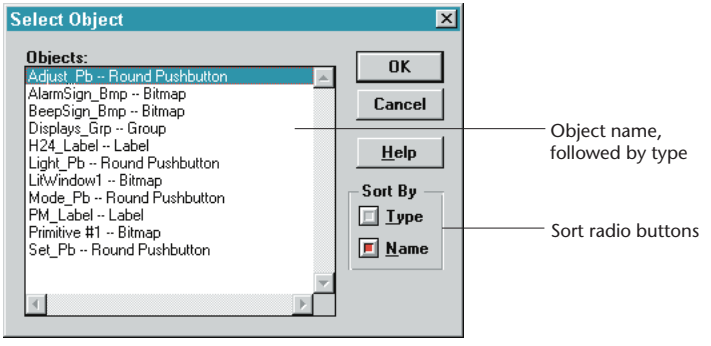
Until you uncheck this command, every time you select an object, all of its children will be selected.

Selecting objects from a list

As your applications become more complex, containing large numbers of objects, you might find it easier to select objects from Rapid's Object list. This list displays all of the graphic objects contained in the application.

To open the Object list:

- Choose Edit|Select from List to open the Object list.



Rapid sorts the list alphabetically, in one of two ways:

- By object type, followed by object name. This sort is useful if you only know the type of object you want to select.
- By object name, followed by object type (default). This sort is useful when you know the name of the object you want to select.

Moving Graphic Objects

You can move single or multiple objects by:

- Dragging them with the mouse.
- Using the arrow keys.
- (For a single object only) Entering new values in the Position boxes.

❖ *NOTE: Only a move operation using the mouse can be undone by the Undo command.*

Moving objects using the mouse

1. Select the objects.
2. Drag the objects to the new location.

Moving objects in grid-size jumps using the arrow keys

1. Select the objects.
2. Press Shift + the appropriate arrow key. The first press of the arrow brings the objects into alignment with the grid, and subsequent presses move the objects in grid-size jumps.

Moving an object by precise amounts using the parameter pane

1. Select the object and open its parameter pane.
2. Overtype the current position coordinates (refer to p. 2–22 for an explanation of these parameters), then click Accept.

❖ *NOTE: The grid size does not affect this method of moving.*

More on xy coordinates

Here is a summary of how you would move an object by changing its xy values in the parameter pane:

- To move left: reduce the x value.
- To move right: increase the x value.
- To move up: reduce the y value.
- To move down: increase the y value.

Resizing Graphic Objects

You can resize a single object either by changing the size parameter in its parameter pane or by using the mouse. You can resize multiple objects by grouping them, and then using the mouse.

To resize a single object through its parameter pane:

1. Select the object and open its parameter pane.
2. Overtype the current size coordinates (refer to p. 2–23 for an explanation of these parameters), then click Accept.

❖ *NOTE: The grid does not affect this method of resizing.*

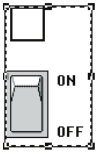
To resize a single object with the mouse:

1. Select the object.
2. Click and hold one of the object's resizing handles; the cursor changes into a two-directional arrow shape.
3. Drag the mouse and release the mouse button when the object reaches the desired size.

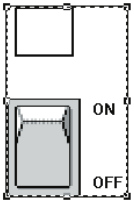
❖ *NOTE: This method of resizing is affected by the grid.*

To resize multiple objects simultaneously:

1. Select the objects.
2. Group them (see "Grouping Objects" on p. 2–33).
3. Use the group's resizing handles as for a single object. When you release the mouse button, all objects within the group's enclosing rectangle are resized in proportion.



a) Objects grouped



b) After resizing

Cutting, Copying and Pasting Graphic Objects

You can cut or copy graphic objects from the layout to Rapid's internal paste buffer, and then paste them to add identical objects on the layout. The Cut and Copy commands also copy the objects' bitmaps to the Windows Clipboard. The bitmaps can be pasted into documents generated in other Windows programs.

Cutting graphic objects

Cutting an object removes it from the layout, places it in Rapid's internal paste buffer and copies its bitmap to the Clipboard. When you cut objects which are used in the application's logic, Rapid warns you that cutting them also deletes the related logic and asks you to confirm the cut operation.

To cut a graphic object(s):

1. Select the object(s).
2. Choose Edit|Cut, or click the Cut button.


Ctrl+X

Cutting an object erases any objects previously placed in the paste buffer.

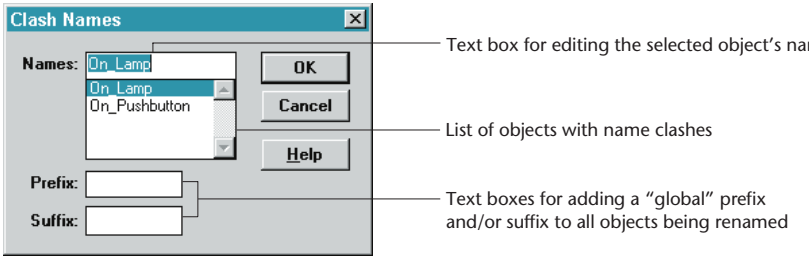
❖ *NOTES: Until you overwrite the internal paste buffer contents, you can recover the cut objects with the Paste Object command.*

If you want to paste a hierarchy of objects, you must check the Options|Select Children option before cutting.

Cutting an object which has children

If you cut (delete or replace) an object without its children, these children objects become "orphans." Orphans are made children of their closest surviving ancestor (e.g., grandparent, if it was not also cut).

If this reallocation would result in sibling objects with the same names, you need to rename the orphaned objects. You do this through the Clash Names dialog box which automatically opens before the cut operation is carried out.



After making the appropriate changes, click OK. Rapid rechecks for invalid names and only closes the dialog box (and completes the Cut/Delete/Replace operation) when none are found.

❖ *NOTES: The children of orphans are unaffected by the renaming procedure, apart from any changes in their parent names.*

Even if you add a prefix and/or a suffix to all orphaned objects, Rapid does not assign names to orphaned primitive objects that did not previously have names.

Copying graphic objects

Copying an object places it in Rapid's internal paste buffer without removing it from the layout, and copies its bitmap to the Windows Clipboard. Copying an object clears the internal paste buffer of its previous contents.

To copy a graphic object(s):


Ctrl+C

1. Select the object(s) you want to copy.
2. Choose Edit|Copy, or click the Copy button.

❖ *NOTE: If you want to paste a hierarchy of objects, you must check the Options|Select Children option before copying.*

Pasting graphic objects

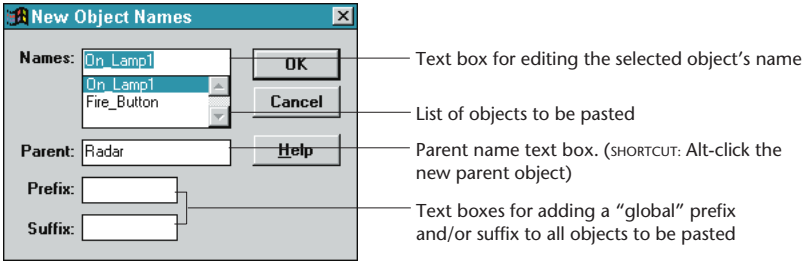
You can paste an object from Rapid's internal paste buffer and place it at any location in the work area. Pasted objects are always made children of the root object. If you want to paste a hierarchy of objects, you must check the Options|Select Children option before copying/cutting.

❖ *NOTE: Because the Paste Object operation does not clear the paste buffer, you can add copies of the same object(s) at more than one location on the layout.*

To paste a graphic object:


Ctrl+V

1. Choose Edit|Paste Object, or click the Paste button; the cursor changes to a crosshairs.
2. Place the crosshairs where you would like the object to be pasted and click. (The crosshairs marks the upper left corner of the object's enclosing rectangle.) If the paste operation does not cause any name clashes, the pasted object is drawn on the layout and is selected. (Skip steps 3–4.)
3. If the Paste operation creates name clashes, the New Object Names dialog box opens:



❖ *NOTE: If you edit the object's name, the prefix and/or suffix (if any) are appended to the changed name.*

4. When you have completed editing the object names, click OK.

   Rapid automatically checks the names and parentage that you have assigned the pasted objects. If there are still errors, such as a sibling name clash, the dialog box remains open to either correct the problem or to cancel the Paste operation.

   When the paste operation is successfully completed, Rapid draws the pasted objects on the layout and selects them.

❖ *NOTE: The above procedure also applies to all pasted active primitive objects. Unnamed primitive objects do not appear in the New Object Names dialog box.*

Duplicating Graphic Objects

You can duplicate selected objects using the mouse. A duplicated object does not overwrite any object previously cut or copied to the internal paste buffer.

To duplicate an object:

1. Ctrl-click the object you want to duplicate.
2. Drag the object frame that appears to a new location.
3. Release the mouse button. The duplicated object is drawn on the layout and receives a default name (see "Naming Objects" on p. 2–11).



a) Ctrl-click **b) Drag** **c) Release**

❖ *NOTES: Do not release the Ctrl key before the mouse button or the object will not be duplicated, but will be moved to the new location.*

If you duplicate an object which is part of a group (in group-edit mode), the new object will not be part of the group. To make it a part of the group, use the Group|Add Members command (see p. 2–34).

To duplicate multiple objects:

1. Select the objects.
2. Ctrl-click **one** of the selected objects and drag.

Deleting Graphic Objects

Deleting an object removes it entirely from the application; the object is not placed in the internal paste buffer. When you delete object(s) that are used in the application's logic, Rapid asks you to confirm the delete operation.

❖ *NOTE: Deleting the object also deletes the related logic.*

To delete graphic objects:



- Select the objects you want to delete, then choose Edit|Delete; the objects are removed from the layout.

  If you delete an object without its children, the children become "orphans." (See "Cutting an object that has children" on p. 2–30.)

Undoing Operations

If you have made a mistake, such as incorrectly resizing an object, don't worry! You can return to the situation prior to your mistake by using the Undo command. Undo cancels the previous operation if the operation was:

Resize object	Delete	Align
Move	Distribute	Center
Change line width		

Redo reperforms the canceled operation.

To undo your previous action:



- Choose Edit|Undo, or click the Undo button.

❖ *NOTE: If you click anywhere after your action, the Undo option is unavailable.*

Grouping Graphic Objects

When you want to resize or move several objects simultaneously, you can group them and use the Group menu commands. With these commands, you can also add and remove objects from existing groups, and edit individual objects within a group, without having to ungroup them.

Rapid treates a group as an individual object in the Logic Editor. The group's member objects are children of the group.

Grouping separate objects

You can group any graphic objects, regardless of their proximity.

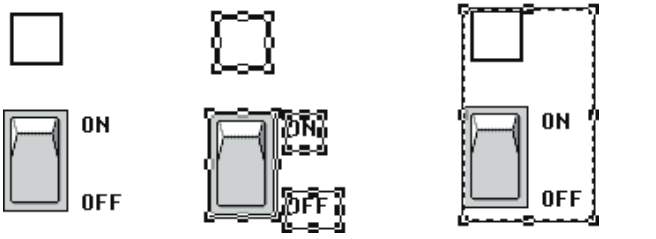
To group separate objects:

1. Select the objects.
2. Choose Group|Create; an enclosing, dashed rectangle surrounds the selected objects, and Rapid assigns a name to the group.

   If any of the objects has children which were not selected to be part of the group, a dialog box appears asking if you want to include the children in the group. If you click Yes, the children are also included in the enclosing rectangle. If you click No, no group is created.



❖ *NOTE: A drag 'n drop object which is part of a group can be dragged, unless the group itself is defined as drag 'n drop (see "Behavior of groups" on p. 2–35).*



a) Several Objects b) Objects selected c) Objects grouped

❖ *NOTE: You can group two or more groups together.*

Ungrouping objects

To ungroup objects:

1. Select the group.
2. Choose Group|Break; the enclosing rectangle disappears and the individual objects are all selected.


Ctrl+U

Adding objects to a group

To add objects to a group:

1. Select the group and the objects you want to add to the group.
2. Choose Group|Add Members; the enclosing rectangle surrounds the added objects.

Editing objects within a group

In edit mode, you can edit individual objects within a group, add objects to a group and even create groups within a group.

To edit objects within a group:

1. Select the group.
2. Choose Group|Edit. All of the individual objects are selected.
3. Click anywhere on the layout to deselect the objects, then select the object you want to edit.
4. When you have completed editing the objects, choose Group|End Edit.

❖ *NOTES: You can edit more than one group at a time. When you end an edit, the group which was last selected for editing is closed first.*

To add objects to a group while in edit mode:

1. Perform steps 1 through 3 in "To edit objects within a group" above.
2. Select a member of the group and also the objects you wish to add to the group.
3. Choose Group|Add Members; the group enclosing rectangle reappears surrounding the original group and the added objects. Rapid automatically ends the edit-group mode.

Removing objects from a group

To remove objects from a group:

1. The group must be in edit mode before you can remove objects from it (perform steps 1 through 3 in "To edit objects within a group" above).
2. Select the objects you wish to remove from the group.
3. Choose Group|Remove Members. The group enclosing rectangle no longer surrounds the removed objects, and the group is selected. Rapid automatically ends the edit-group mode.

Behavior of groups

In general, a group behaves like a single object in the Object Layout. You can move and resize it, make it dynamic and drag 'n drop, etc.

In addition, a group has its own set of functions in the Logic Editor that allow you to hide, show and disable it. (Disabling a group does not disable its member objects).

If the group is dynamic, you can change its position in the logic.

A group can be drag 'n drop. Click anywhere within the group (including on a primitive object or on an empty space) to drag it.

❖ *NOTE: A group's empty space (between objects) is* **not** *transparent. If a clickable object is located* **behind** *a group, you will not be able to click it during runtime. In the Object Layout, make sure to either send the group to the back or bring the clickable object to the front.*

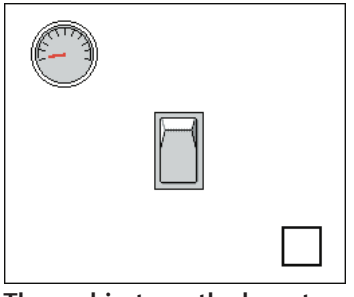
Aligning, Centering and Distributing Graphic Objects

The Align, Center and Distribute commands in the Layout menu enable you to improve the layout's appearance. You must select at least two objects for these commands to be active.

❖ *NOTE: The align operations are not affected by the grid size (see p. 2–13).*

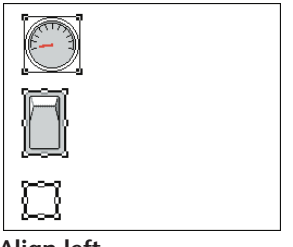
Aligning objects

Use the Layout|Align submenu commands to align the selected objects in a vertical or horizontal row. Rapid aligns objects according to their enclosing rectangles. For example, align commands are useful for straightening rows and columns of keypad keys.

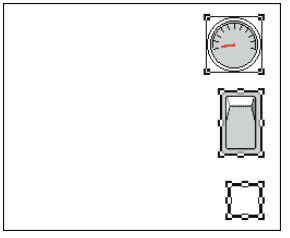
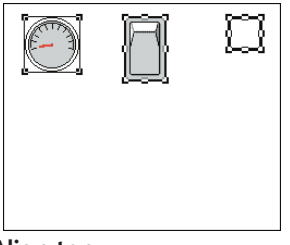
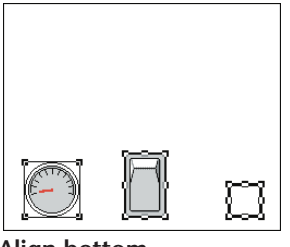


Three objects on the layout

USE THIS COMMAND	TO
Left *(Ctrl+L)*	Align the left sides of the objects with the left side of the left-most object.

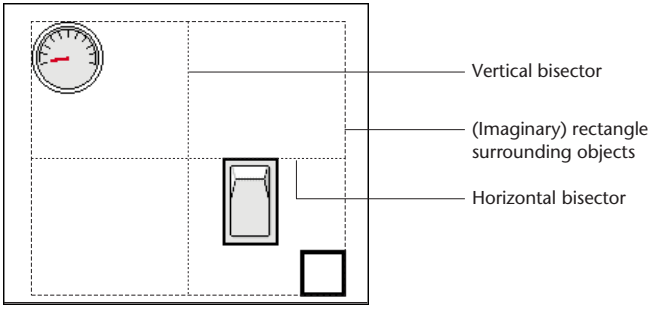


Align left

USE THIS COMMAND	TO
Right *(Ctrl+R)*	Align the right sides of the objects with the right side of the right-most object.  **Align right**
Top *(Ctrl+T)*	Align the tops of the objects with the top of the highest object.  **Align top**
(Bottom *Ctrl+B)*	Align the bottom sides of the objects with the bottom side of the lowest object.  **Align bottom**

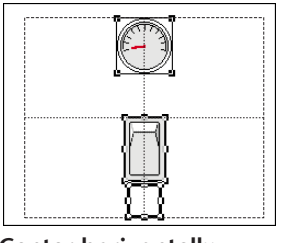
Centering objects

Use the Layout|Center submenu commands to line up the centers of the selected objects horizontally or vertically. For example, centering is useful to center a label in relation to its object.



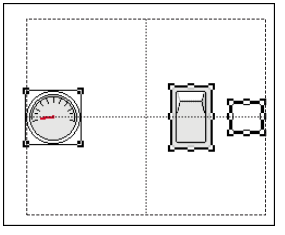
Three objects on the layout

USE THIS COMMAND	TO
Horizontal (Ctrl+Shift+H)	Move the objects horizontally, so that the center of each object lines up with the vertical bisector of the (imaginary) rectangle that surrounds the selected objects.



Center horizontally

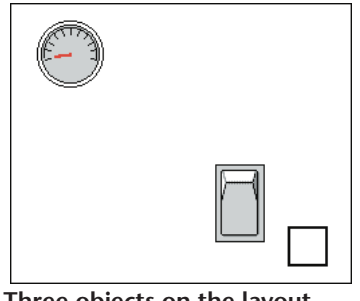
USE THIS COMMAND	TO
Vertical (Ctrl+Shift+V)	Move the objects vertically, so that the center of each object lines up with the horizontal bisector of the (imaginary) rectangle that surrounds the selected objects.



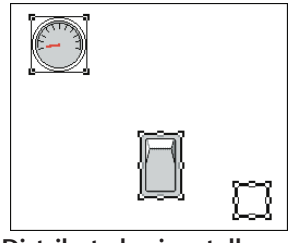
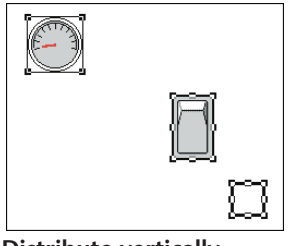
Center vertically

Distributing objects

Use the Layout|Distribute submenu commands to create equal spaces between selected objects. For example, distributing is useful to create equal distances between keypad keys that have already been aligned in a row or a column with the Align commands.



Three objects on the layout

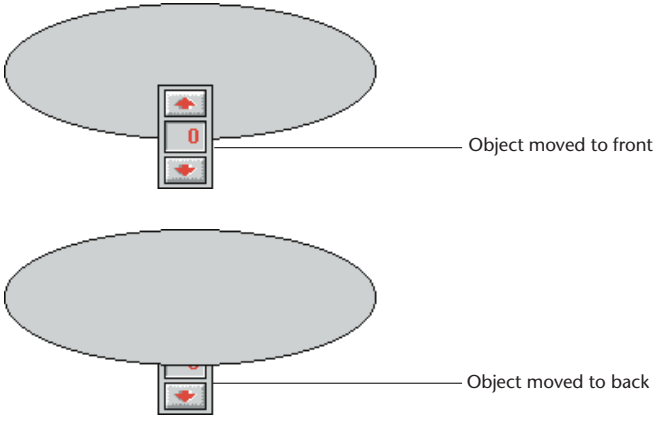
USE THIS COMMAND	TO
Horizontal (Ctrl+Shift+Z)	Move the objects horizontally so that there is a uniform horizontal gap (or overlap) between them. The left-most and right-most shapes do not move. This operation does not affect the left-to-right order of the selected objects.  **Distribute horizontally**
Vertical (Ctrl+Shift+E)	Move the objects vertically so that there is a uniform vertical gap (or overlap) between them. The highest and lowest shapes do not move. This operation does not affect the top-to-bottom order of the selected objects.  **Distribute vertically**

Changing Objects' Stacking Order

Often you will want to place one graphic object on another, e.g., a switch on a frame or a label on a button. The Layout menu offers you two commands which control how Rapid displays overlapping objects:

- Move to Front
- Move to Back

These commands force the selected objects to the foreground or background of the screen. As illustrated below, objects moved to the front will "hide" any objects that are behind them, and objects moved to the back will be "hidden" by any objects that are in front of them. A typical use for the Move to Front or Move to Back commands would be to provide a sense of depth by superimposing an active object on a bitmap or by sending a bitmap behind an active object.



To move an object to the front:


Ctrl+F

- Choose Layout|Move to Front.

To move an object to the back:


Ctrl+B

- Choose Layout|Move to Back.

❖ *NOTES: In the Prototyper, a text window will always appear in the foreground of the screen even if you moved it to the back.*
If you place a primitive object in front of an object that responds to a mouse click (e.g., a pushbutton), the object will come to the front when you click it in the Prototyper. To get around this, make a primitive that is in front of an object a child of that object.

Hiding and Showing Graphic Objects

In some circumstances, you may want to hide temporarily objects on your layout. For example, if the layout is very "crowded," hiding objects can help you concentrate on the particular objects with which you are currently working.

Hiding objects

The Layout|Hide Object command allows you to hide the selected objects.

❖ *NOTE: You cannot select a hidden object.*

To hide objects:

1. Select the objects you want to hide.
2. Choose Layout|Hide Object; the objects are hidden on the layout.


Ctrl+H

Showing objects

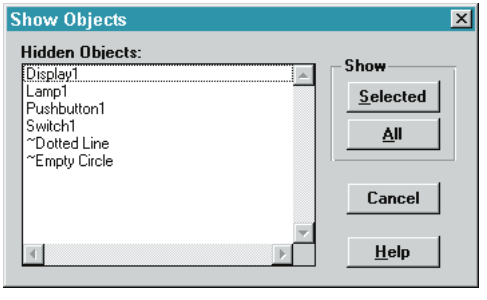
The Layout|Show Objects command allows you to redisplay hidden objects.

To show one or more hidden objects:

1. Choose Layout|Show Objects to see a list of objects hidden on the layout.


Ctrl+W

❖ *NOTE: If you have unnamed, hidden primitives, the object type name appears at the end of the list preceded by a tilde (~), e.g., "~Dotted Line."*



2. Select the hidden objects you want to redisplay.
2. Click the Selected button. The objects are displayed on the layout.

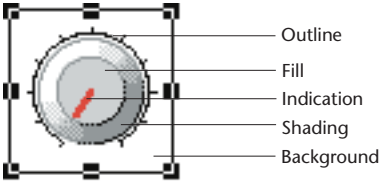
To show all hidden objects:

- In the Show Objects dialog box, click the All button.

Editing Objects' Colors

Rapid's graphic objects contain several different color elements, each of which is assigned a different color. When you add an object to the layout, Rapid colors it using its default color scheme. The different color elements and their default colors are:

COLOR ELEMENT	DESCRIPTION
Line	The color of the thin line that outlines the various surfaces of the object, as well as any text and external calibration markers (default: black).
Shading	The color of the highlighted areas that give the object its 3-dimensional appearance (default: dark gray).
Fill	The color of the object's flat surfaces (default: light gray).
Background	Any area within the object's enclosing rectangle that is not occupied by the object, as well as the color of lighter areas (default: white).
Indication	The color of the marker that indicates the object's current position, a lit lamp and the text of a stepper switch (default: red).

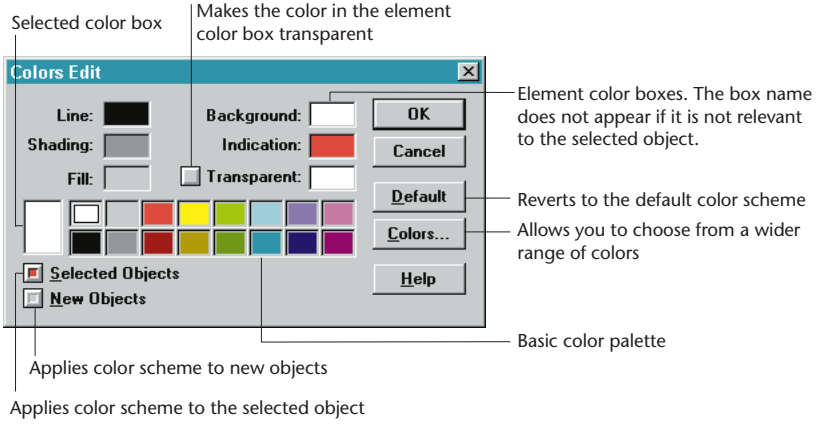


You can change the colors of objects using the Edit|Colors command. With this command, you can either change the colors of the currently selected object or the colors of objects which will be added to the layout.

To edit an object's colors:

1. Select the object.
2. Choose Edit|Colors to open the Colors Edit dialog box. Note that the Selected Objects radio button is selected.

Ctrl+L



   Only the color element names relevant to the selected objects appear. The example above is for a rotary potentiometer, so all of the color elements appear. (If a filled frame were selected, only the Fill and Line names would appear.)

3. Click a color in the color palette (or click the Colors button, see "Choosing from a wider range of colors" below); the color in the Selected color box (to the left of the color palette) changes to the selected color.
4. Click the Element color box for the color element you want to change.
5. Repeat Steps (3) and (4) until you are satisfied with the color scheme.

To change the color scheme for all new objects:

1. Click an empty space in the layout so that no object is selected.
2. Choose Edit|Colors to open the Colors Edit dialog box. Note that the New Objects radio button is selected and that all of the color element names appear.
3. Perform steps 3 through 5 above to change the color scheme.

To define a color as transparent:

1. Click the Transparent check box.
2. Select a color, then click the Transparent color box. All color elements with the same color in their Element color boxes will be transparent.

To revert to the default Rapid color scheme:

- Click the Default button.

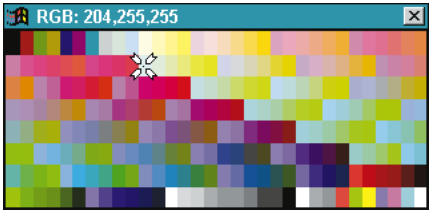
Choosing from a wider range of colors

If your display supports 256 colors or more, you can can choose from a wider range of colors than the basic color palette.

To choose from a wider range of colors:

- Click the Colors button; the dialog box that Rapid opens depends on your display's color mode:

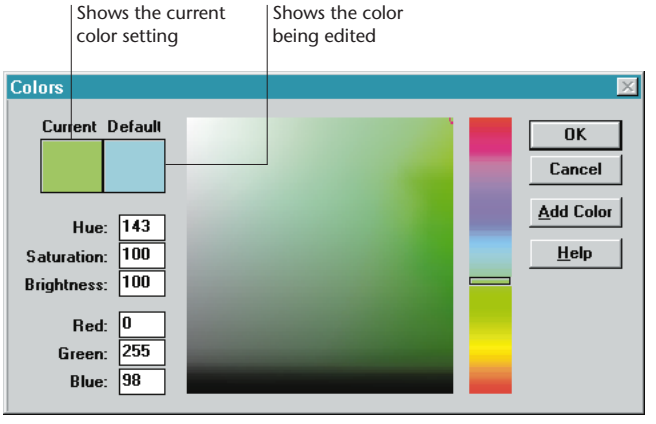
256 color mode



1. Click a color in the palette to select it. (Note that the color's RGB numbers appear in the title bar.) The color appears in the Selected color box in the Edit Colors dialog box.
2. Apply this color to the color elements as described above.

❖ *NOTE: If you color an object with colors from the 256-color palette, you may have problems if you later import or paste a bitmap (see "Importing bitmaps in 256 color mode" on p. 2–51).*

"HiColor" (16 bit) and "TrueColor" (24 bit) modes



1. Drag the slider until the approximate color that you want appears in the Current color box.
2. Click and drag in the color display area until the Current color box shows the precise color that you want.

   If you know the HSB (hue, saturation, brightness) or RGB (red, green, blue) values of the color that you want, enter them into the appropriate edit boxes.
3. Click Add Color to confirm your selection (and leave the dialog box open), or OK to close the dialog box.

RGB and HSB

These are two different models for specifiying a color.

Red, Green, Blue (RGB)

In RGB color, various brightness values of red, green, and blue light combine to form the colors on the screen. The range of colors in the visible spectrum is represented by controlling the intensities of the individual RGB components.

Values for the RGB components range from 0 to 255.

Hue, Saturation, Brightness (HSB)

HSB color is defined by hue, saturation, and brightness:

Hue is the main attribute of a color that designates its position in the visible spectrum. Values for the hue component range from 0 to 359, where red is zero.

Saturation is defined as the amount of gray in a color. The higher the saturation, the less gray there is, or more purity of color. Values for the saturation component range from 0 to 100.

Brightness describes the intensity of light reflected from an image. Values for the brightness component range from 0 to 100. Brightness is also known as "lightness" or "luminosity" (hence HLS).

Editing a Graphic Object in the Object Editor

You can take an object from the Object Layout into the Object Editor to edit:

- Physical appearance (bitmap).
- Position and size of the active areas —if any.
- Position, size and shape of the pointer—if any.

To edit an object from the layout in the Object Editor:

1. Select the object on the layout.
2. Choose Objects|Edit Object to open the Object Editor with the selected object's bitmap displayed in the edit area.



For details on working with the Object Editor, see Chapter 9.

Replacing Graphic Objects

You can replace a graphic object with another object of the same class. For example, you can replace a rocker switch with a rotary switch or a rotary potentiometer with a linear slider potentiometer, etc.

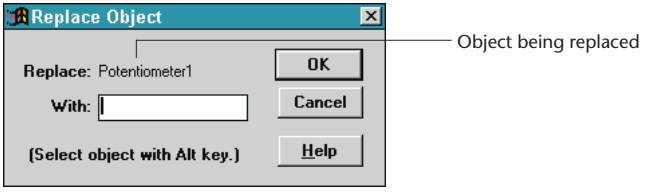
This feature is especially useful if you have already built logic around the object that you want to replace. If you were unable to replace the object, you would have to delete the object, together with its logic, add a new object and build the logic from scratch.

To replace an object:

1. Add the replacement object to the layout if it does not already appear.



2. Choose Edit|Replace Object to open the Replace Object dialog box.



3. Alt-click the replacement object on the layout. A frame appears around the replacement object and its name appears in the With text box.

Rules for replacing objects

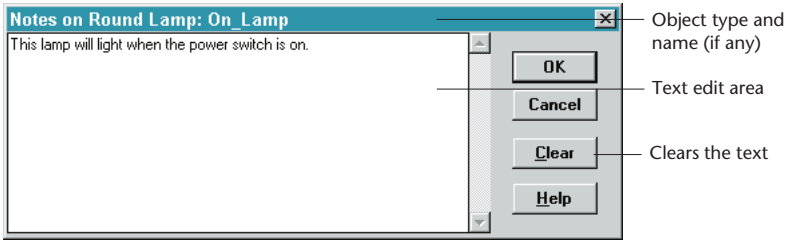
- You cannot replace a group with another group.
- You cannot replace an object with another object if the first object has more properties than the second one. For example, if the first object is a 3-position toggle switch, you cannot replace it with a 2-position toggle switch.
- The only switch type that can replace a stepper switch is another stepper switch.
- You cannot replace objects of the same class if the first object is dynamic and the second one is not dynamic. However, you can replace a non-dynamic object with a dynamic one.
- You cannot replace a dynamic round lamp with a dynamic square lamp (and vice versa) because a dynamic round lamp has the *radius* property and the dynamic square lamp has the *position* property.
- You can replace an active primitive object with an active primitive object of a different class. For example, an active circle can replace an active frame.
- You cannot replace a dynamic primitive object with a dynamic primitive object of a different class. Exception—a dynamic frame can replace a dynamic label (and vice versa) because they both have the *position* property.

Adding and Viewing Notes on Objects

You can attach a note to a graphic object which will appear on object reports. You can view and modify the note at any time. If an object has a note, an asterisk (*) appears on the Notes button.

To add or view a note:

1. Open the object's parameter pane and click the Notes button. The following dialog box (shown with sample text) appears:



2. Type in (or modify) the note in the text edit area. The usual rules for keyboard entry apply.

Redrawing the Layout

Use the Layout|Refresh Display command to rebuild and redraw all the objects in the Object Layout. This command clears any graphic smears that may have accumulated while you were moving or resizing objects. It is especially useful when your layout contains font-related objects (label, text display, text window and menu objects). These objects may be distorted in the following situations:

- When you run an application under a different screen resolution than the one on which it was designed.
- When you run an application under a different video driver than the one on which it was designed.
- When you run an application on a Windows system which does not include the font originally defined for the object.

To redraw the layout:


Ctrl+D

- Choose Layout|Refresh Display.

ADDING BITMAPS TO YOUR LAYOUT

To make your application even more realistic, Rapid allows you to add bitmaps to the layout. These bitmaps become imported bitmap objects in Rapid and are treated as primitive objects, that is, they can be sized, moved, cut, copied, made active, etc. To add bitmaps to the layout, you can either:

- Paste a bitmap from the Clipboard using the Edit|Paste Bitmap command.
- Import a bitmap file using the File|Import Bitmap command. Rapid allows you to import BMP, DIB and RLE bitmaps.

Both of these methods are described below.

❖ *NOTE: Although you cannot change an imported bitmap object's colors in the Colors Edit dialog box, you can use this dialog box to select one of the object's colors and make it transparent.*

Pasting from the Clipboard

To paste the Windows Clipboard contents:



1. Choose Edit|Paste Bitmap. If the Clipboard is empty, or if it does not contain a bitmap, Rapid displays an appropriate message. Otherwise, the cursor changes to a crosshairs.
2. Place the bitmap on the layout as you would place any graphic object (see p. 2–18 for instructions). The contents of the Clipboard are inserted as a single, primitive object.

   If you are working in 256 color mode, refer to "Importing bitmaps in 256 color mode" below.

   Because the Paste Bitmap operation does not clear the Clipboard, you can add the same bitmap at more than one location on the layout.

Importing a Bitmap

To import a bitmap from file:

1. Choose File|Import Bitmap to open the Open Bitmap File dialog box.
2. Select a bitmap file.

   If you are working in 256 color mode, refer to "Importing bitmaps in 256 color mode" below.

3. The cursor appears as a crosshairs. Place the bitmap on the layout as you would place any graphic object (see p. 2–18 for instructions).

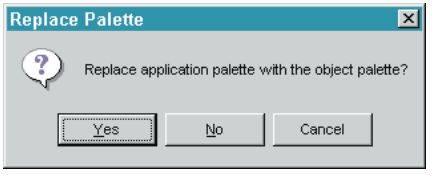
❖ *NOTE: In general, RLE files are smaller than the equivalent BMP bitmap, so your application will be smaller. RLE bitmaps, though, are limited to 256 colors.*

Importing Bitmaps in 256 Color Mode

If you work with Rapid in 256 color mode, Rapid uses its own 256-color palette. You can change the colors of your objects (see "Editing Graphic Object Colors" on p. 2–45) using the colors of this palette.

❖ *NOTE: For a more detailed description of how different display color modes affect Rapid, see Appendix A: "Graphic Performance in Rapid."*

When you attempt to paste or import a bitmap that is based on a different 256-color palette, Rapid displays the following dialog box:



Retaining Rapid's palette

If you choose "No," the new bitmap will be displayed in Rapid's 256-color palette. The colors of the objects already appearing on the layout are not affected, though the new bitmap's colors might change.

Replacing Rapid's palette

If you choose "Yes," and your layout contains objects that use colors from Rapid's original 256-color palette, their colors might change according to the new palette. The new bitmap, though, will be displayed in its correct colors.

To avoid "palette blues," it is preferable to work with a 16 bit ("HiColor") display or better. If that is not practical, you should follow these guidelines:

- Objects whose colors are restricted to the 16 fixed colors, as they appear in the Colors Edit dialog box, will not be affected by a new palette. These 16 colors are standard to all palettes.

- If your layout is to include a bitmap based on a 256-color palette, add this bitmap **before** changing the colors of any other objects. You can then change colors using the new palette.
- It is not possible to import two bitmaps, from two different sources, based on two different palettes, without damaging at least one of the bitmaps!

CHAPTER 3

Primitive Objects

Primitive objects are graphic elements used to enhance an application's physical appearance. They can also add to an application's functionality. For example, in an application that requires a calculator, adding numbers (using label objects) to the pushbuttons not only makes the Rapid calculator look more realistic, but assists in the testing of the calculator in the Prototyper.

Primitive objects are passive, that is, they cannot be manipulated in the Prototyper. They do not have properties and functions, and so cannot interact with active and nongraphic objects. You can, however, incorporate primitive objects into an application's logic by making them active and giving them properties.

❖ *NOTE: Imported bitmaps behave like primitive objects (see p. 2–50 for instructions on adding bitmaps to an application).*

In this chapter you will read about:

- Types of primitive objects. The primitive objects are presented in the order in which they appear in the Object Palette.
- Changing primitive objects' line width.
- Incorporating primitive objects into an application's logic.
- Giving primitive objects additional properties.



FRAMES

The following types of frame objects are available:

TYPE		DESCRIPTION
	*Square Corners*	A rectangular frame that has square corners and a transparent interior.
	*Round Corners*	A rectangular frame that has rounded corners and a transparent interior.
	*Square Corners Filled*	A rectangular frame that has square corners and a filled interior.
	*Round Corners Filled*	A rectangular frame that has rounded corners and a filled interior.
	*Outline*	A rectangular frame that has a border which appears “engraven” or sunken.
	*In*	A rectangular frame with a protruding 3-dimensional border framing a recessed interior.
	*Out*	A rectangular frame with a recessed 3-dimensional border framing a protruding interior.

For instructions on adding a frame, see p. 2-15,“Adding Graphic Objects”.



CIRCLES AND ELLIPSES

The following types of circle/ellipse objects are available:

TYPE		DESCRIPTION
	*Circle*	A circle with a transparent interior.
	*Circle Filled*	A circle with a filled interior.

TYPE		DESCRIPTION
	*Ellipse*	An ellipse with a transparent interior.
	*Ellipse Filled*	An ellipse with a filled interior.

For instructions on adding a circle or ellipse, see p. 2-15, “Adding Graphic Objects”.



LINES

The following types of line objects are available:

TYPE		DESCRIPTION
	*Solid Line*	A solid straight line.
	*Dashed Line*	A dashed straight line.
	*Dotted Line*	A dotted straight line.

Drawing a Line Object

To draw a line and resize it simultaneously:

1. Select a line object using either the Objects|Add command or the Object Palette (see “Selecting a Graphic Object to Add” on p. 2–15).
2. Place the crosshairs at the position where you would like the line’s starting point to appear, then click and drag the line that appears on the screen.

   When you are satisfied with the length and angle of the line, release the mouse button. The line is drawn on the layout.



LABEL

The label object is used for adding fixed text to the layout (e.g., "On" and "Off" labels next to a switch's on and off positions). The initial size of a label object is defined by four factors:

- Font
- Font size
- Font style (bold, italic, etc.)
- Number of characters

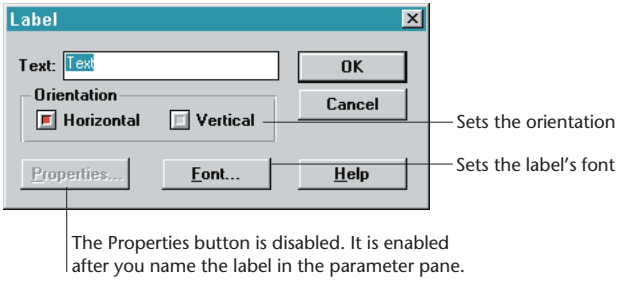
❖ *NOTE: The selected line width does not affect labels.*

Once you have added a label to the layout, you can change its size like any other graphic object (see "Resizing Graphic Objects" on p. 2–28).

Adding a Label Object

To add a label:

1. Select the Label object using either the Objects|Add command or the Object Palette (see "Selecting a Graphic Object to Add" on p. 2–15).
2. Place the crosshairs where you would like the first letter to appear and click; the Label dialog box opens:



3. Overtype "Text" with the appropriate text.
4. The default orientation is horizontal. If want to change the orientation, click the Vertical radio button.
5. To change the font, click the Font button.
6. Click OK; the title text is added to the layout.

Label object fonts

When you resize a label that contains proportional fonts (e.g., True Type), the font size changes according to the size of the enclosing rectangle. With fixed fonts, the font size changes in "fixed" increments as you enlarge or reduce the enclosing rectangle's size.

Whenever you change the font, font size and/or style for the selected label, your font selection becomes the default for new labels added to the current application. If Options|Save Settings on Exit is enabled, your font selection will also be the default for the next new application.

Modifying a Label Object

You can move, resize, and change the color of a label object like any other graphic object.

❖ *HINT: To resize a number of labels, group them, then resize the group.*

To modify a label:

1. If you want to change the label's name, parent, size or position, make the necessary changes in the parameter pane, then click Accept.
2. If you want to change the label's text, click the More button. Make the necessary changes in the Label dialog box, then click OK. The dialog box closes and the label is redrawn on the layout.
3. To add properties to the label, click the Properties button in the Label dialog box. The Properties button is enabled only if you named the label (see p. 3–8 regarding properties).

To make the label fill opaque:


Ctrl+L

1. Select the label, then choose Edit|Colors.
2. In the Colors Edit dialog box, uncheck the Transparent check box. Click OK to accept the fill color; otherwise change the fill color, then click OK.

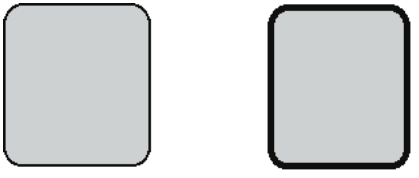
Assigning parents to label objects

When you add a label on top of an active object (e.g., a number on a pushbutton), you should make it a child of the active object. If the label is not a child of the active object, it will be hidden during runtime when the active object is manipulated (e.g., the pushbutton is pressed).

You assign a parent in the label's parameter pane.

SETTING THE LINE WIDTH

You can set the line width for frame, circle, ellipse and line (solid only) objects. The line width determines the thickness of the frame, circle or ellipse outlines, and the thickness of solid lines. You cannot set the line width for dotted lines, dashed lines, labels and imported bitmaps.



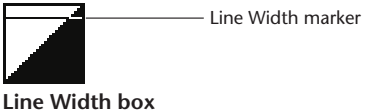
Round cornered frames with different line widths

Line width can be changed for the currently-selected objects or set for new objects which will be added to the layout. You can set the line width:

- Using the Options|Line Width command.
- Using the Line Width box.

The Line Width Box

The Line Width box is located in the bottom left corner of the Object Layout window. The black triangle represents line width thickness ranging from 1–30 pixels. When you click inside the box, a horizontal marker appears at that position. The selected line width is determined by the thickness of the black triangle at the point where the marker appears, as illustrated below:



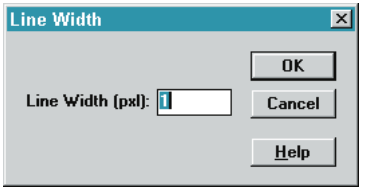
Line Width box

You can click and drag the marker up and down the box. When the mouse button is released, the line width is selected and the marker disappears.

Adjusting the Line Width for Existing or New Objects

To change the line width of an existing object:

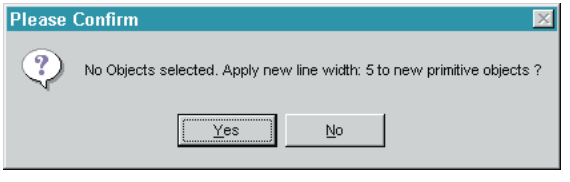
1. Select the object.
2. Change the line width either using the Line Width box (as described above) or choose Options|Line Width to open the Line Width dialog box:



2. Overtype the measurement in the Line Width text box (in your application's scale), then click OK.
3. The selected object is redrawn on the layout with the new line width.

To set the line width for new objects:

1. Click an empty space in the layout so that no object is selected.
2. Set a new line width either using the Line Width box or the Line Width dialog box (as described above). The following message appears:



3. If you click Yes, all new primitive objects you add will adopt the new line width.

MAKING PRIMITIVE OBJECTS ACTIVE

Initially, a primitive object is passive, that is, it is not included in an application's logic and therefore cannot be manipulated in the Prototyper. You can, however, give the primitive object a role in the logic by:

- Naming the primitive object.
- Adding properties to the primitive object.
- Making the primitive object dynamic and/or drag 'n drop.

See the following sections for details.

Naming a Primitive Object

If you name a primitive object, it behaves like an active object. Once named, the "active" primitive object:

- Can be the parent of any object on the layout.
- Has set functions and properties.
- Appears in the Logic Editor's object list.
- Can be assigned properties.

To name a primitive object:

1. Double-click the object; its parameter pane opens.
2. Type a name in the Name text box, then click Accept. Notice that the More button is enabled.

Adding Properties to Primitive Objects

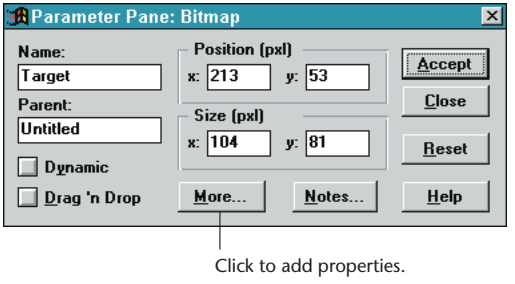
After you have named a primitive object, you can add properties to it that allow the object to contain data. The properties can hold any of the basic types of Rapid data (number, integer, string, array or point).

If, for example, your application is a radar system and you have a primitive object ("Target") that represents an airplane on the radar screen, you could add a number property ("Altitude") for storing the altitude of the airplane at any given time and a string property ("Type") for storing the airplane type.

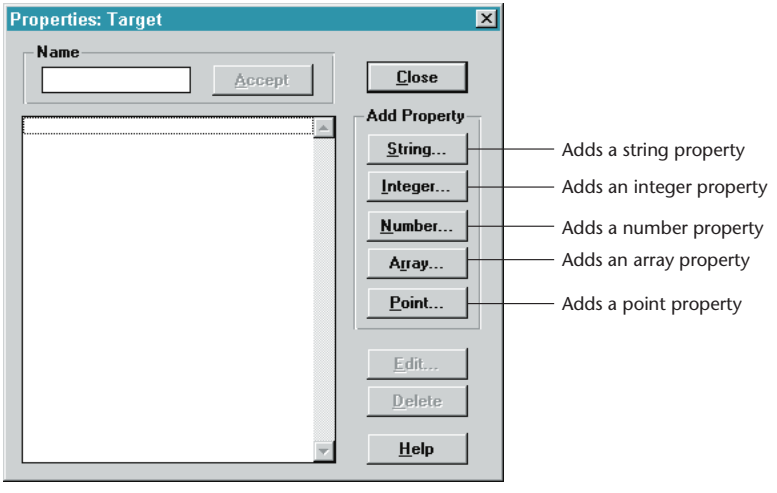
❖ *NOTE: You can also add properties (and events) to the root mode itself. This is useful when the application is a user-defined object (see Chapter 19: "User-Defined Objects").*

To add properties to an active primitive object:

When you name a primitive object and click the Accept button, the More button is enabled (as illustrated below for the bitmap object named "Target").



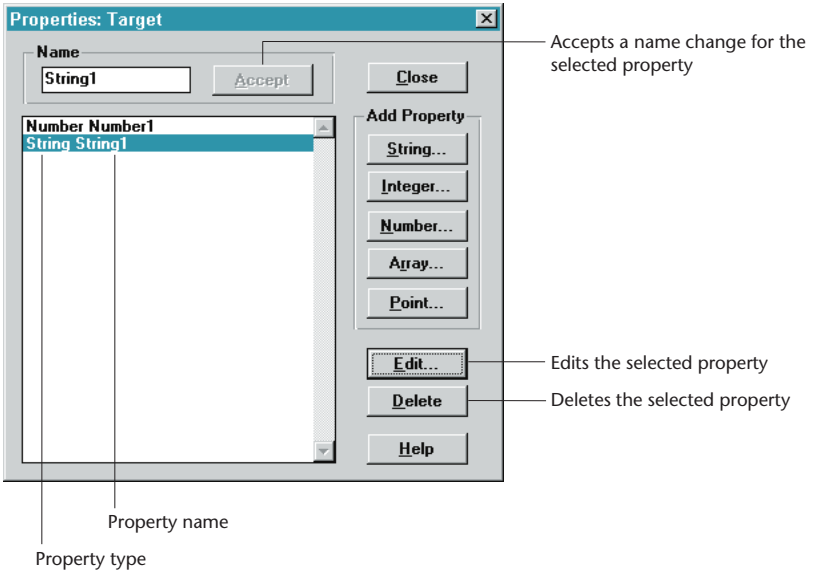
1. Click the More button to open the Properties dialog box:



2. Click the appropriate button in the Add Property group. The property's dialog box opens in which you can define initial values for the property. (See Chapter 5: "Nongraphic Objects" for an explanation of the String, Integer, Number, Point and Array dialog boxes.)

   Rapid gives the property a default name consisting of the property type and a sequential number (String1, Integer1, Integer2, etc.).

If you added a number and a string property, the Properties dialog box would appear as:



To change an existing property's name:

1. Select the property in the property list. Its name appears in the Name text box.
2. Edit the name in the Name text box.
3. Click Accept; the property's name is changed in the list.

To modify a property's parameters:

1. Select the property, then click the Edit button.
2. Make the necessary changes in the property's dialog box.

Making a Primitive Object Dynamic

Making a primitive object dynamic adds properties and functions to its logic that enable the object to be moved in a predetermined path in the Prototyper. A dynamic, primitive object can also be resized and have its color changed in the logic, as well as generate an event when clicked in the Prototyper.

To make a primitive object dynamic:

1. Double-click the object; its parameter pane opens (click once if the parameter pane is already open).
2. Type a name in the Name text box.
3. Click the Dynamic check box.
4. Click Accept.

Making a Primitive Object Drag ‘n Drop

Making a primitive object drag ‘n drop enables the object to be dragged in the Prototyper using the mouse. A drag ‘n drop object generates an event when it is dragged and dropped onto another object.

To make a primitive object drag ‘n drop:

1. Double-click the object; its parameter pane opens (click once if the parameter pane is already open).
2. Type a name in the Name text box.
3. Click the Drag ‘n Drop check box.
4. Click Accept.

CHAPTER 4

Active Objects

Active objects are control and display components that play "active" roles in an application's behavior. Active objects imitate their real-life counterparts in both appearance and behavior. For example, a Rapid rotary switch looks like a real, generic rotary switch.

Active objects appear in the Logic Editor's object list ready to be used to construct logic. The logic you construct controls an application's behavior. For example, if you add a rotary switch, a lamp and a text display to the Object Layout, you can construct logic that cause the following behaviors in the Prototyper: using the mouse, you can "turn" the rotary switch to a position which causes the lamp to light; or when the rotary switch is turned to another position, you can display text in the display.

Even if you don't use a specific object in a logic phrase, you can still manipulate the object in the Prototyper, e.g., turn a rotary switch's knob (although it would not cause a reaction in a different object), drag an object that has been designated as drag 'n drop.

In this chapter you will read about:

- The types of active objects, presented in the order in which they appear in the Object Palette.
- The parameters unique to each object.

❖ *NOTE: The dialog box for each object opens when you click the parameter pane's More button. Unless otherwise specified, default parameter values are shown in the dialog boxes.*





LAMPS

This class of active objects comprises round and square lamps.

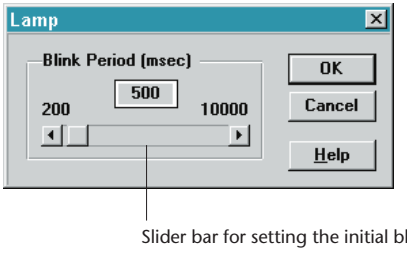
A lamp has two states: on and off, and three color elements: line, background and indication. When the lamp is off, the background color is displayed; when it is on, the indication color is displayed.

You can also cause a lamp to blink by using its *blink* function. When a lamp blinks, it alternates between the on and off states at the frequency determined by its blink period.

Blink period

The blink period is the total time that the lamp is on and off. For example, if the lamp's blink period is 500 msec, the lamp will blink on for one quarter of a second and off for one quarter of a second.

The Lamp dialog box is used to set the initial blink period:



Slider bar for setting the initial blink period



INDICATORS

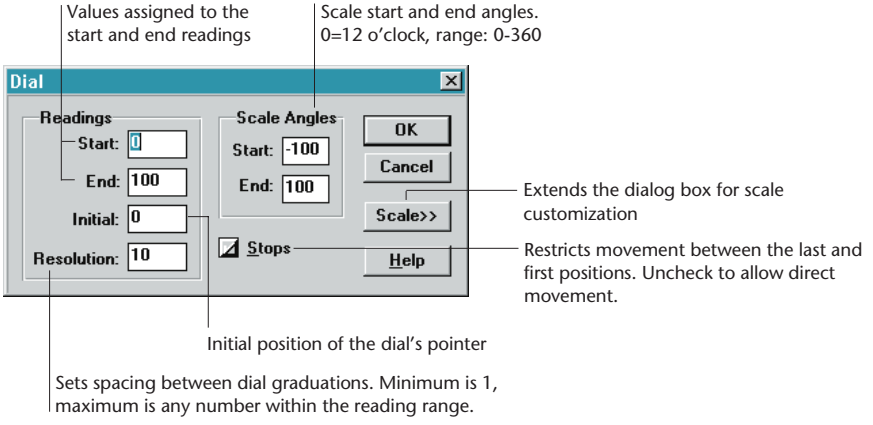
This class of active objects comprises scales that display numeric information in an analog fashion. There are six types of indicator objects: round and square dials, vertical and horizontal linear indicators, and vertical and horizontal bar indicators.



Dials

Round and square dials display analog information via a needle that moves around a circular dial. Examples are a car speedometer and a fuel gauge.

Round and square dials use the same dialog box:

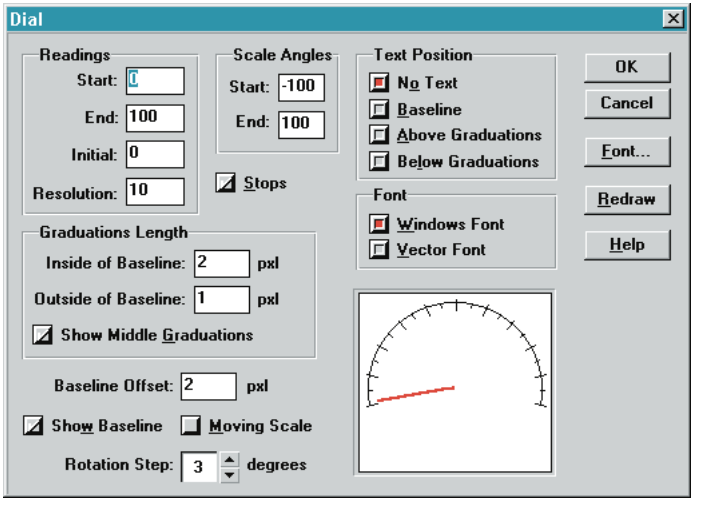


Customizing dial scales

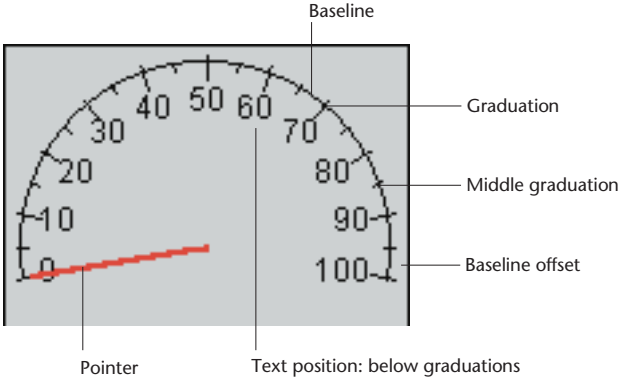
You can customize dial object scales. For example, you can set the font for the scale's text or decide to hide the text altogether, adjust the length of the graduations, etc.

To customize a dial's scale:

- In the Dial dialog box, click the Scale>> button; the dialog box extends to display the scale customization options:



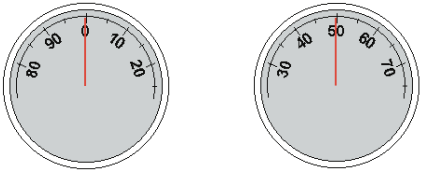
The parts of a dial scale are:



The parameters for customizing the scale are:

- **Text position**: Radio buttons for choosing the location of the text. The text can be placed either on the baseline, above or below the graduations. Click the No Text radio button if you do not want text to appear.
- **Font**: Radio buttons for choosing the font type for the text. Vector fonts are not as versatile as Windows fonts, but they are drawn 5– 6 times faster. They are especially useful when the scale is a moving scale and speed is important.
- **Graduations Length**: Text boxes for entering the length (in pixels) of the graduations inside and outside the baseline.

- **Show Middle Graduations:** Check box for showing or hiding middle graduations between the scale's graduations (e.g., if this box is checked for a scale with a resolution of 10, middle graduations will appear at 5, 15, 25, etc.).
- **Baseline Offset:** Text box for determining the distance of the baseline (in pixels) from the object's edge.
- **Show Baseline**: Check box for showing or hiding the baseline.
- **Moving Scale**: Check box for turning the scale into a moving scale—the pointer remains constant and the scale moves (e.g., a compass).



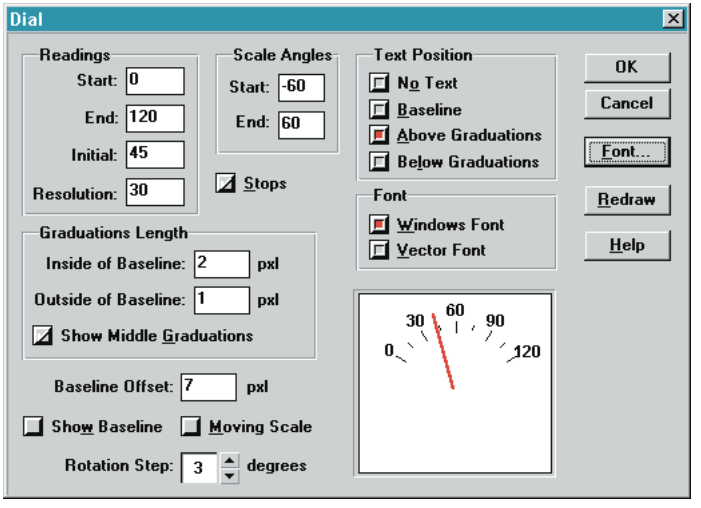
Example of a moving scale

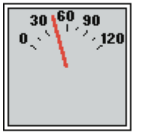
- **Rotation Step**: Stepper box for assigning a rotation step that determines the dial's sensitivity to changes in the reading. The higher the rotation step, the bigger change in the reading is needed to make the dial's pointer move (or the scale move, if it is a moving dial). This feature is useful to "dampen" the dial to minor fluctuations, thereby improving the graphic performance of the application. The rotation step's range is 0–10.

  ❖ *NOTE: You can change the value of the dial's rotation step in runtime by changing the value of the dial's rotationStep property.*

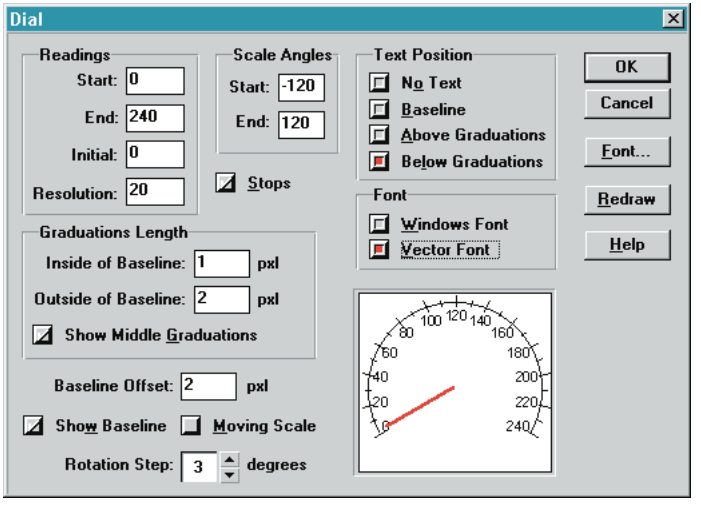
- **Scale preview window**: Scale display. Some changes appear as soon as they are made (e.g., changing the text position). Other changes appear when the Redraw button is clicked (e.g., changing the graduation lengths).
- **Font button**: Opens the Font dialog box for changing the scale's font. If you choose Windows fonts, you can only use True Type fonts on the scale, and the available font sizes depend on the size of the dial on the layout. If you choose vector fonts, the attributes you set only affect the font's size and weight (i.e., bold or regular).
- **Redraw button**: Redraws the scale in the scale preview window including changes made in the dialog box.

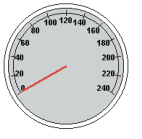
Two examples of Dial customization dialog boxes and the resulting dials are shown below:



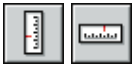


Dial customization dialog box and the resulting square dial





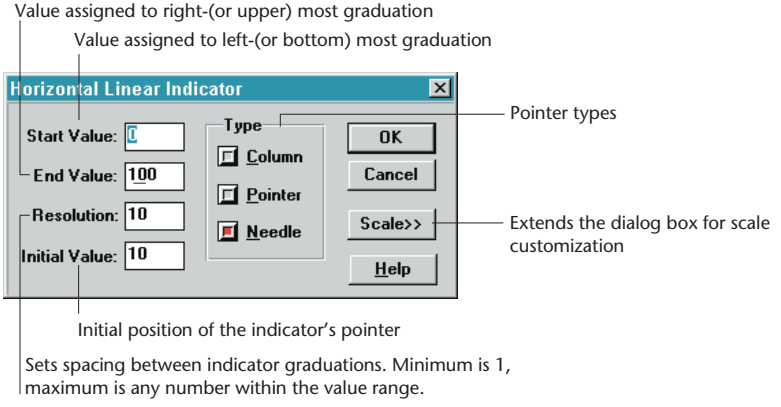
Dial customization dialog box and the resulting round dial



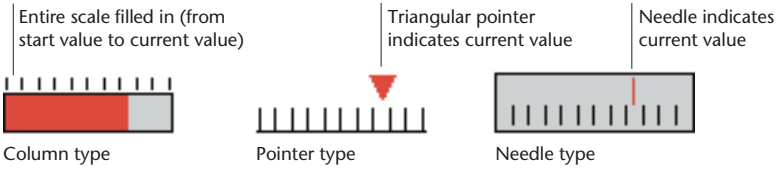
Linear Indicators

The linear indicator object is either a vertical or horizontal linear scale that displays analog information. A typical use for a linear indicator is the tuner on a radio receiver.

Vertical and horizontal linear indicators use similar dialog boxes. The dialog box for a horizontal linear indicator is shown below:



There are three types of pointers: column, pointer (triangular) or needle. The difference between the three pointer types is illustrated below, where the reading of each indicator has been assigned the same value:



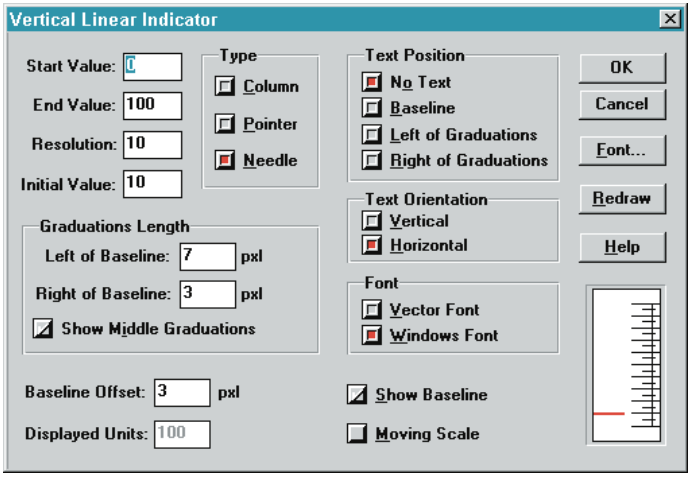
❖ *NOTE: You can edit an indicator's pointer in the Object Editor; see Chapter 9: "The Object Editor."*

Customizing linear indicator scales

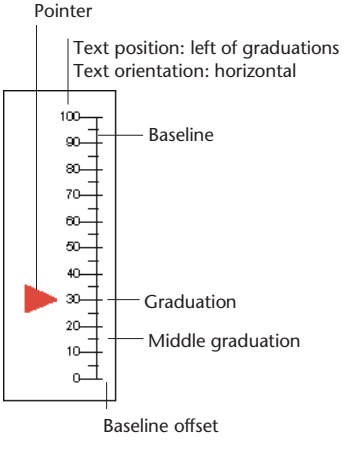
You can customize linear indicator scales. For example, you can set the font for the scale's text or decide to hide the text altogether, adjust the length of the graduations, etc.

To customize a linear indicator's scale:

- In the Linear Indicator dialog box, click the Scale>> button; the dialog box extends to display the scale customization options:

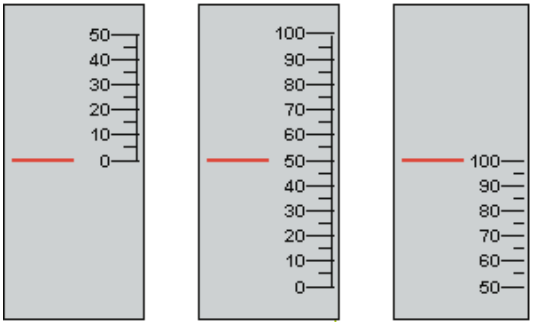


The parts of a linear scale are:



The additional parameters for customizing the scale are:

- **Text position**: Radio buttons for choosing the location of the text. For a vertical linear indicator, the text can be placed either on the baseline, to the left or to the right of the graduations. For a horizontal linear indicator, the text can be placed either on the baseline, above or below the graduations. Click the No Text radio button if you do not want text to appear.
- **Text Orientation**: Radio buttons for choosing whether the text is displayed horizontally or vertically.
- **Font**: Radio buttons for choosing the font type for the text. Vector fonts are not as versatile as Windows fonts, but they are drawn 5– 6 times faster. They are especially useful when the scale is a moving scale and speed is important.
- **Graduations Length**: Text boxes for entering the length (in pixels) of the graduations inside and outside (or above and below) the baseline.
- **Show Middle Graduations:** Check box for showing or hiding middle graduations between the scale's graduations (e.g., if this box is checked for a scale with a resolution of 10, middle graduations will appear at 5, 15, 25, etc.).
- **Baseline Offset/Height:** Text box for determining the distance of the baseline (in pixels) from the object's edge.
- **Show Baseline**: Check box for showing or hiding the baseline.
- **Moving Scale**: Check box for turning the scale into a moving scale, i.e., the pointer remains constant and the scale moves (e.g., a weighing scale).

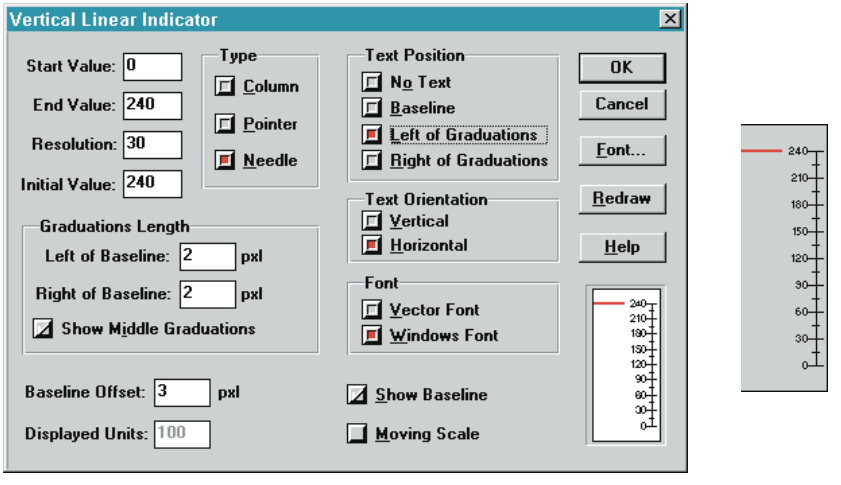


Example of a moving scale

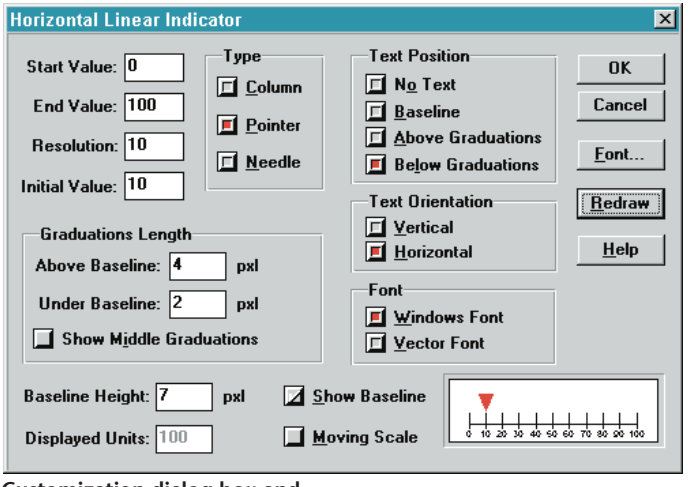
- **Displayed Units**: Text box for selecting the number of units displayed on the scale (moving scales only). In the illustration above, the displayed units is 100 units.

- **Scale preview window**: Scale display. Some changes appear as soon as they are made (e.g., changing the text position). Other changes appear when the Redraw button is clicked (e.g., changing the graduation lengths).
- **Font button**: Opens the Font dialog box for changing the scale's font. If you choose Windows fonts, you can only use True Type fonts on the scale, and the available font sizes depend on the size of the indicator on the layout. If you choose vector fonts, the attributes you set only affect the font's size and weight (i.e., bold or regular).
- **Redraw button**: Redraws the scale in the scale preview window including changes made in the dialog box.

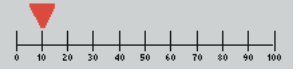
Two examples of Linear Indicator customization dialog boxes and the resulting indicators are shown below:



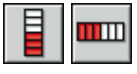
Customization dialog box and the resulting vertical linear indicator



Customization dialog box and...



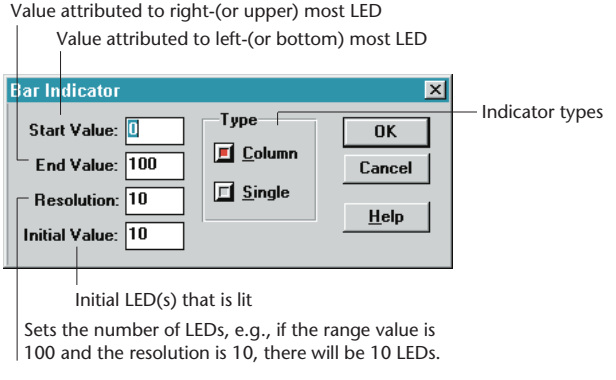
...the resulting horizontal linear indicator



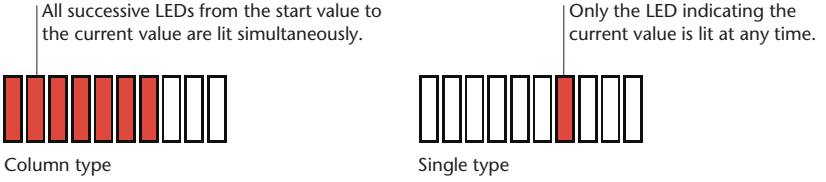
Bar Indicators

The bar indicator object is either a column or row of discrete, rectangular LEDs. A bar indicator displays analog information by the position of either a single lit LED or by a number of successively lit LEDs — depending on whether you have defined the indicator as Single or Column. A typical use for a bar indicator is the graphic equalizer on an amplifier.

Vertical and horizontal bar indicators use the same dialog box:



The difference between single and column type is illustrated below, where the horizontal bar indicator readings have both been assigned the same value:



Barrel Dials

The barrel dial is used to simulate mechanical barrel dials such as a car odometer. Multiple barrel dials can be linked together using Rapid logic to simulate more complex dials (e.g., different colors, fonts or sizes for the sections of an odometer).

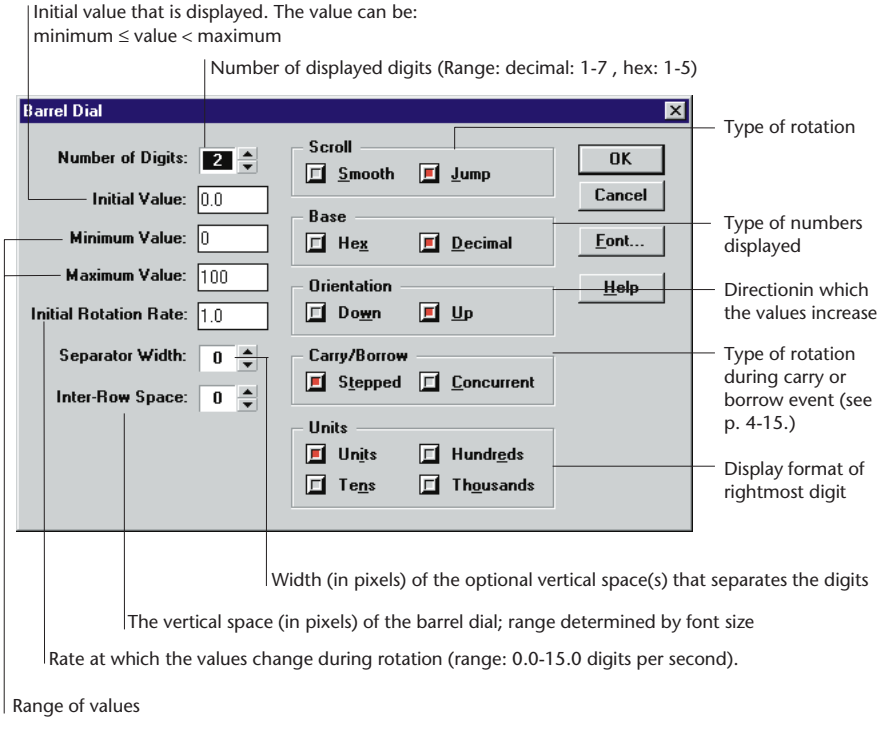
The Rapid barrel dial object simulates a mechanical device by displaying fractional values through the rotation of its digits.

When you add a barrel dial object in the Object Layout, it looks as follows:



❖ *NOTE: You cannot resize a barrel dial using its resizing handles or through the parameter pane. You must use its dialog box.*

The Barrel Dial dialog box defines its display and how it runs:



Setting the barrel dial's size

The barrel dial object's size is determined by the:

- Number of digits;
- Parameters set in the Font dialog box: font, font size, and font style;
- The display format of the rightmost digit;
- Inter-row space, and;
- Width of the separator.

The default size is Number of Digits: 2; font characteristics: Arial, Regular, 13 pt; rightmost digit displayed as units; inter-row space: 0; separator width: 0.

❖ *NOTES: If you change the barrel dial's font characteristics, your font selection becomes the default for new barrel dials created during the* **current** *working session. Every time you exit Rapid, however, the original, default font characteristics become valid.*

The inter-row space defines the number of pixels to be added between the rows of barrel dial values. The valid range for inter-row space is determined by the font size. For example, the range for an 8-pt font is 0-14 pixels; for an 18-pt font, 0-27 pixels.

Below are barrel dials of different sizes and formats:



Number of digits: 2,
Font size: 13 pt,
No separator
No inter-row space

Number of digits: 2,
Font size: 10 pt,
Separator: 1 pixel
Inter-row space: 0

Number of digits: 2,
Font size: 10 pt,
Separator: 1 pixel
Inter-row space: 6 pixels

Number of digits: 6,
Font size: 10 pt,
Separator: 1 pixel
Inter-row space: 0
Rightmost digit: Tens, i.e., the barrel dial value is 551237, which Rapid translates to 55123.70

Setting the barrel dial's values

The minimum possible value for a barrel dial is zero. The maximum possible value is determined by the number of digits. For example, if there are two digits, the maximum possible value is 100 (256 for hex values); if there are three digits, the maximum possible value is 1000 (4096 for hex values), etc.

The initial value can be any real number between the minimum and maximum values.

Determining the rotation

There are three fields that determine how a barrel dial will rotate: initial rotation rate, scroll and orientation. They are described in the table below:

FIELD	DESCRIPTION
Initial rotation rate	Rate at which the values change (range: 0.0-15.0 digits per second). The rate can also be changed in the logic. *NOTE: If the rotation rate is set to 0.0, the barrel dial will not rotate.*
Scroll	Type of rotation. There are two types: **Jump** (default): The display jumps from the current integral value to the next value. Fractional values are not seen, i.e., a value of 58.01 and a value of 58.99 produce the same display—58. **Smooth**: The display moves smoothly from the current value to the next value so that you see fractional values.  **Sample of a smooth scroll** *NOTE: If the scroll is smooth, but the rotation rate is high (or the font is small), the barrel dial will appear as if it has jump scrolling.* The scrolling type cannot be changed in the logic.
Orientation	Direction in which the values increase. There are two directions: **Up** (default): values increase from the bottom upwards:  **Down**: values increase from the top downwards.  The orientation cannot be changed in the logic.

Determining the carry/borrow rotation

During runtime, when a barrel dial rolls from its maximum to its minimum value, the *carry* event is generated; when it rolls from its minimum to its maximum value, the *borrow* event is generated. See p. 13-48 for a description of how you can use these events in the application logic.

In the Object Layout, you determine how the digits rotate when a *carry* or *borrow* events takes place. The alternatives are:

- **Stepped** (default): The display scrolls smoothly to the minimum (in the case of *carry*) or maximum (in the case of *borrow*) value, or
- **Concurrent**: The digits jump simultaneously to the minimum or maximum value.

❖ *NOTES: If the carry/borrow rotation is concurrent, but the rotation rate is high (or the font is small), then the barrel dial will appear as if the rotation is stepped.*

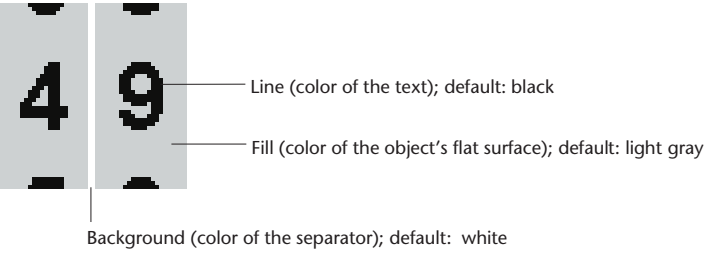
The type of carry/borrow rotation cannot be changed in the logic.

Editing the Barrel Dial's colors

❖ *NOTE: The barrel dial cannot be edited in the Object Editor.*

The barrel dial's colors can be changed in the Colors Edit dialog box (like all graphic objects) and through the logic (like dynamic primitive objects).

In the Colors Edit dialog box, the barrel dial object's color scheme is:



Any of the color elements can be made transparent.

In the Logic Editor, you can change the colors during runtime using the *backgroundColor*, *lineColor* and *fillColor* properties.

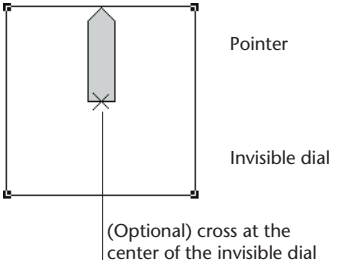


Pointers

You use the pointer object to create different types of pointers. Each pointer rotates around an invisible dial.

The pointer object can be used to create customized gauges or to simulate other components. For example, you could use two mirror-image, ellipse-shaped pointers to simulate a propeller.

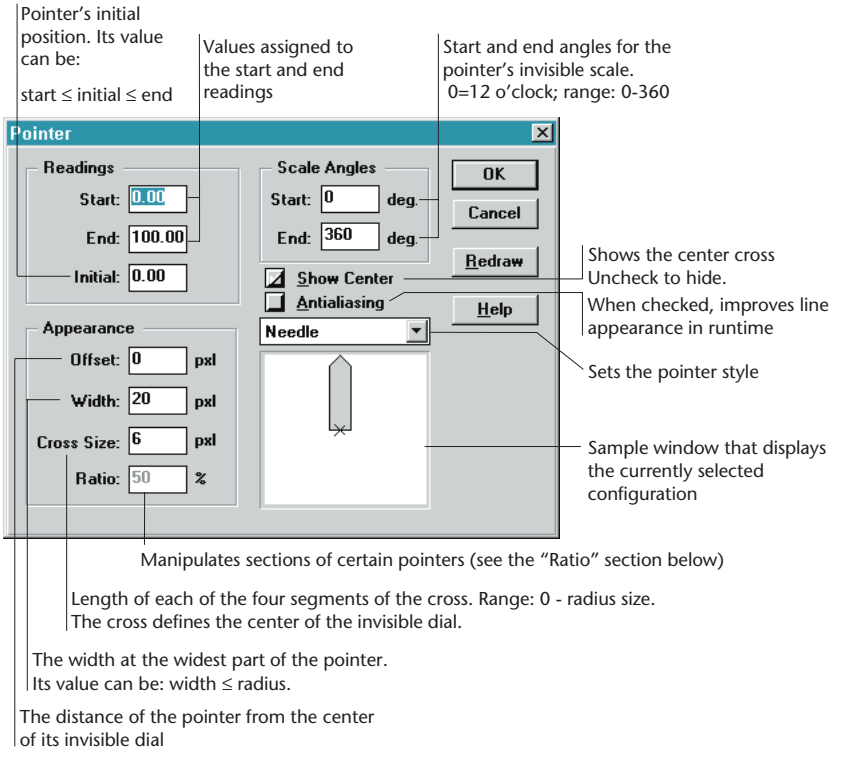
When you add a pointer object to the layout at its default size and shape, it looks like:



❖ *NOTE: The invisible dial is the same size as the object's enclosing rectangle.*

The pointer is treated like a circular object, and so has a center position and a radius. The default radius is 70 pixels.

You use the pointer dialog box to define the pointer's appearance and values. The dialog box below shows a pointer object with default values:



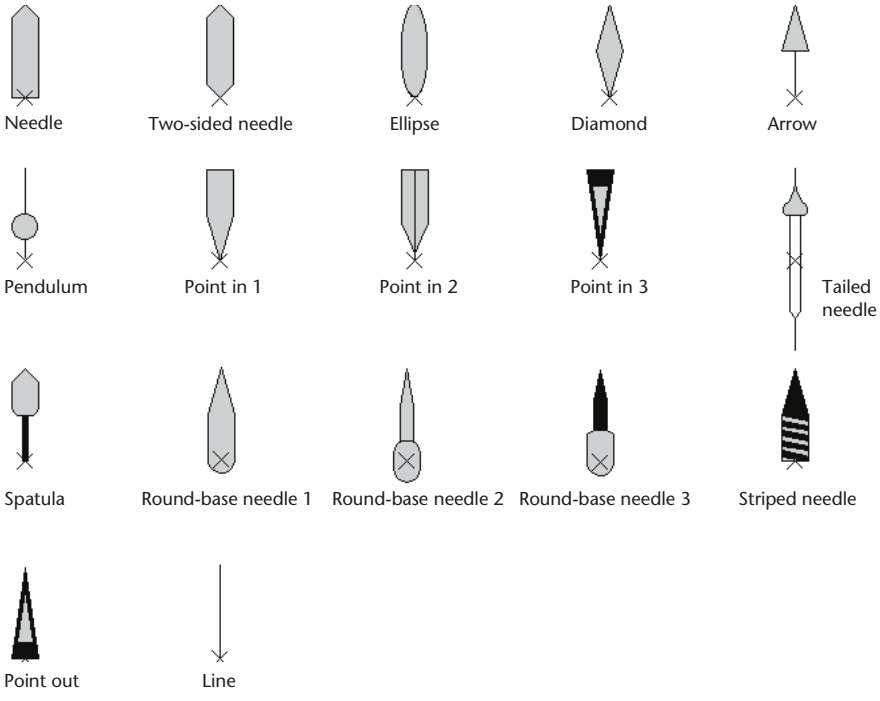
There are several factors that determine a pointer's appearance:

- Pointer style
- Width
- Offset
- Ratio
- Cross size

The first four factors are described below.

Pointer style

The pointer object has 17 styles to choose from:

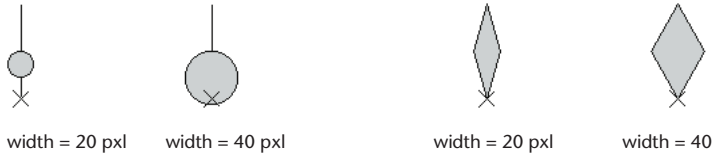


Width

The width field sets the width at the widest part of the pointer (measured in the application's scale units). Its value can be: width ≤ radius.

Examples:

Here are two sets of pointer objects. In each set, the first pointer has the default width (in pixels) of 20 and the second has a width of 40 pixels.



❖ *NOTES: The width field is disabled for the Line style pointer.*

For Round-base needle styles, width applies to the needle section of the pointer.

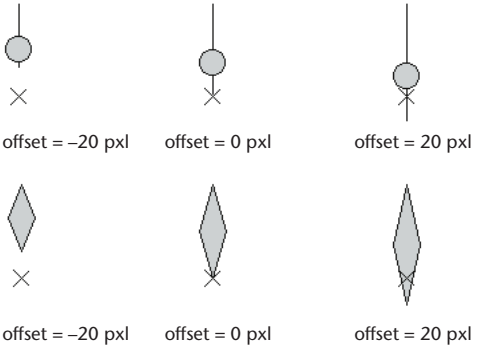
Offset

The offset field determines the distance of the pointer from the center of its invisible dial (measured in the application's scale units). When offset = 0, the bottom edge of the pointer is at the center. The value can be positive or negative. A positive offset elongates the pointer; a negative offset shortens it.

❖ *NOTES: The absolute value of the offset must not exceed the pointer's radius.*

For Round-base needle styles, the offset affects the shape of the base.

Examples:



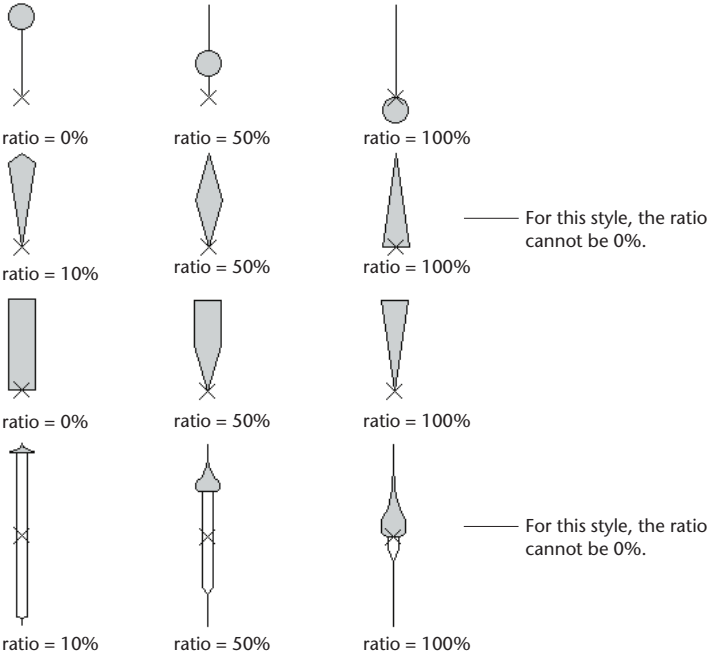
Ratio

(Applies to all pointer styles except Needle, Two-sided needle, Ellipse, Point in 3 and Point out.)

All the pointer styles, except those listed above, were developed in two or more sections. The ratio field lets you manipulate sections of the pointer style so that you can have greater design flexibility. The default ratio is 50%.

Examples:

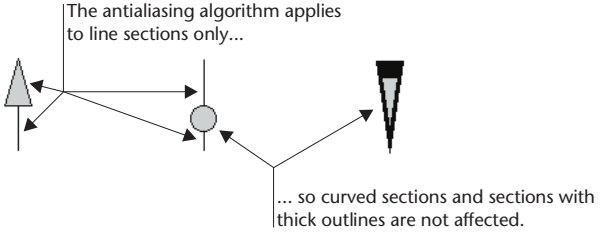
Here are four sets of pointer objects with different ratios:



❖ *NOTE: The ratio must be compatible with the width and offset so that the pointer can rotate without exiting its invisible dial.*

Antialiasing Option

The antialiasing option activates an algorithm which improves a pointer's line appearance in the Prototyper.



This option is disabled for the Ellipse, Point in 3 and Point out styles since they do not contain lines.

❖ *NOTES: For this option to be the most effective, the fill and background colors should be identical.*

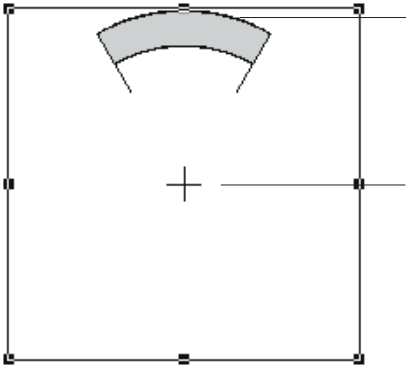
Using the antialiasing option slows redrawing of the pointer in the Prototyper.



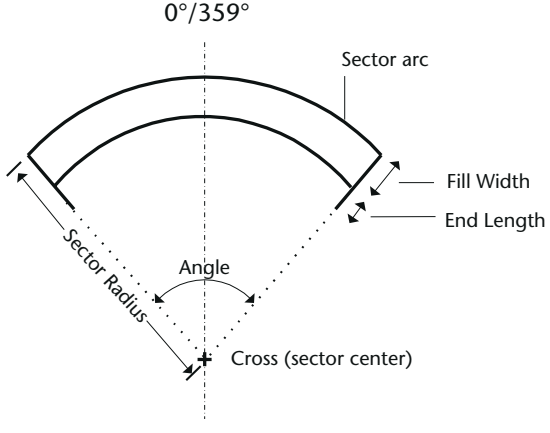
Sector Dial

The sector dial is similar to the other dial objects except that its reading is indicated by an arc defined by two radii, rather than by a pointer. The sector dial reading is given in degrees, from 0 to 359, indicating the center of the arc. A sector dial can be used, for example, in a radar display to designate the sector in which threats are active and/or the sector which is covered by the radar.

When you add a sector dial at its default size, it looks like:

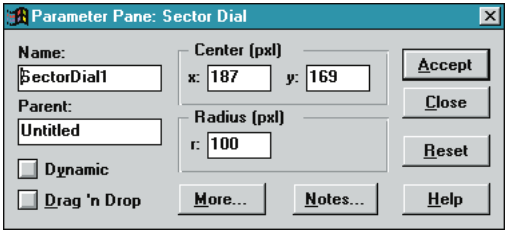


The parts of the sector dial are illustrated below:



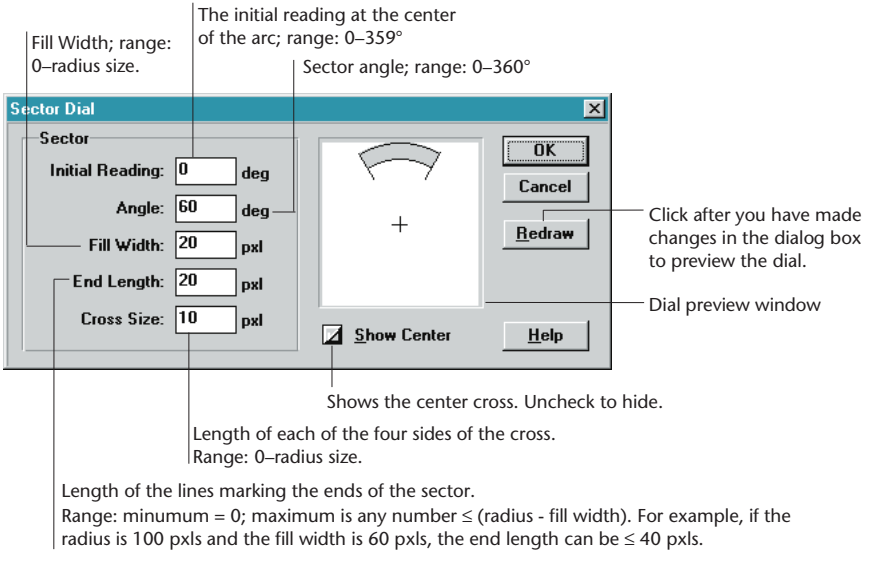
Defining the Sector Dial

The sector dial is treated like a circular object, and so has a center position and a radius (as shown in the parameter pane below):



Unlike other active objects, you can alter the width of its arc, end lengths and cross using the Line Width box/dialog box (see p. xx).

Use the Sector Dial dialog box to define the remaining properties:



❖ *NOTES: The Fill Width, End Length and Cross Size fields are measured in the application's scale units (e.g., if the scale unit is centimeters, all fields will be in centimeters).*

The sector dial cannot be edited in the Object Editor.

Defining the Sector Dial's Colors

The sector dial's colors can be changed in the Colors Edit dialog box (like all graphic objects) and through the logic (like dynamic primitive objects).

Using the Colors Edit dialog box

The sector dial's color scheme is determined by the colors in the Line, Fill and Background color boxes.

- The Line color box sets the color of the sector and cross.
- The Fill color box sets the color that appears in the fill width. This color only appears when the fill width is greater than zero.
- By default, the background is transparent.

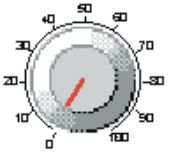
Changing colors via the Logic Editor

You can change the line and fill colors during runtime using the *lineColor* and *fillColor* properties.



POTENTIOMETERS

This class of active objects comprises devices whose output varies in relation to the movement of the object's pointer. A potentiometer's output value can be read at any time and can be used as a condition or as an input of numeric value. A typical use for a potentiometer is the volume control on a radio. There are three types of potentiometer objects:



Rotary potentiometer



Vertical slider



Horizontal slider

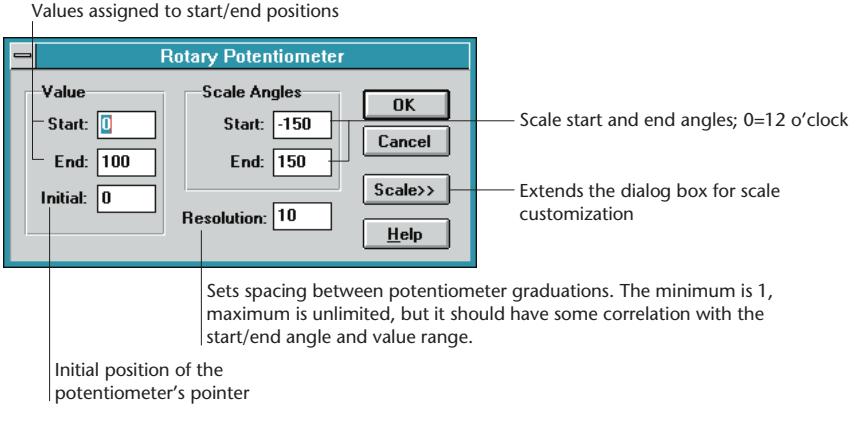


Rotary Potentiometer

The rotary potentiometer object has rotational displacement. There are no software limits on the permissible start/end values and start/end angles. The rotary potentiometer can be multi-turn, i.e., the start angle and end angle can be more than 360° apart.

The number of graduations between the start graduation (at the Start angle) and the end graduation (at the End angle) is determined by the resolution. Thus, if you define a start angle of 0, an end angle of 180 and a resolution of 10, there will be 36 graduations (including middle graduations) between the start and end values.

The values are set in the Rotary Potentiometer dialog box:

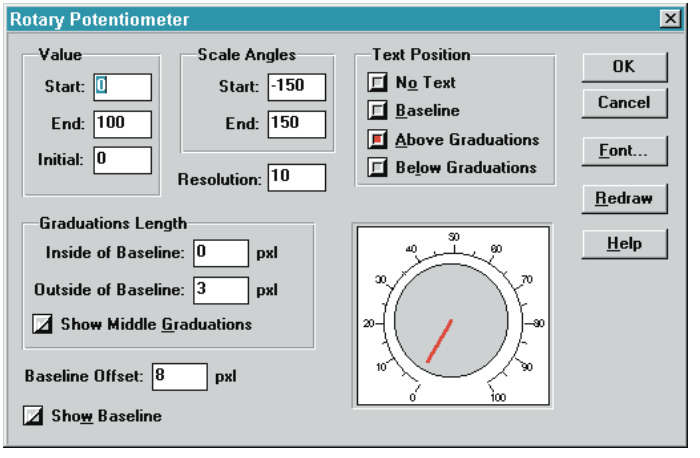


Customizing rotary potentiometer scales

You can customize the rotary potentiometer scale. For example, you can set the font for the scale's text or decide to hide the text altogether, adjust the length of the graduations, etc.

To customize a rotary potentiometer's scale:

- In the Rotary Potentiometer dialog box, click the Scale>> button; the dialog box extends to display the scale customization options:



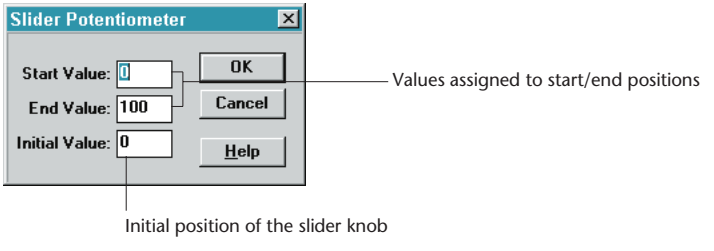
Refer to p. 4–4 for an explanation of the parameters appearing in the extended dialog box.



Slider Potentiometers

The slider potentiometer object is either a vertical or horizontal potentiometer whose knob moves linearly.

Vertical and horizontal slider potentiometers use the same dialog box:





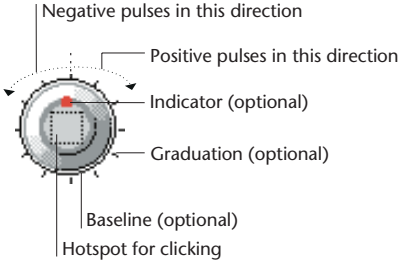
Jogdial

The jogdial object is a specialized potentiometer that generates pulses as it is rotated. One pulse is generated each time the jogdial rotation exceeds a predefined distance in degrees (pulse resolution). If the rotation is in the clockwise direction, a *positivePulse* event is generated; if counterclockwise, a *negativePulse* event.

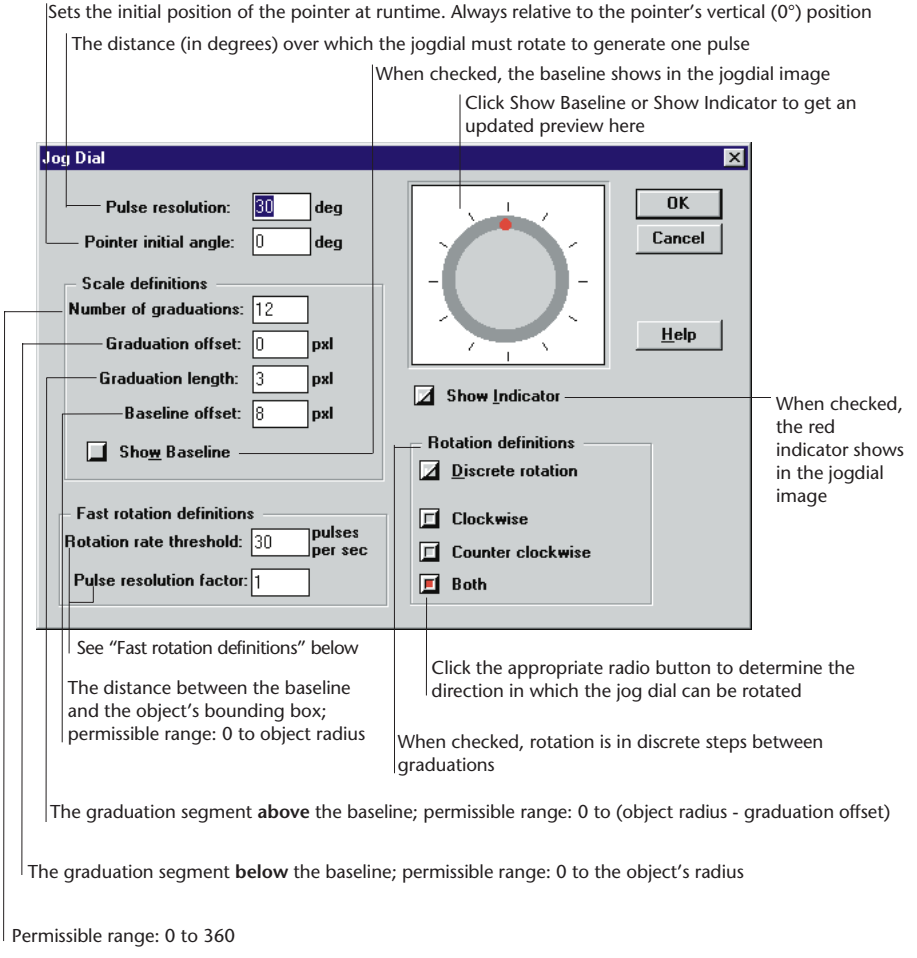
There is no absolute value for any given position of the jogdial's indicator. The same number of pulses are generated when it is rotated by, for example, 60° — whether the rotation starts from the vertical position or from any other position on the scale.

You determine in the Object Layout whether the jogdial can turn in a clockwise or counterclockwise direction — or both directions. In either case, you can continue turning the jogdial for an unlimited number of rotations.

A special case is fast rotation, when the jogdial is rotated at a speed higher than a predefined rotation rate threshold. In this case, the pulse resolution is adjusted by a specified pulse resolution factor.



All of the parameters described above are set in the Jogdial dialog box, as shown on the following page:



Fast rotation definitions

Rotation rate threshold

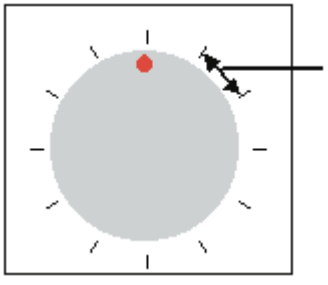
Sets the rotation speed (pulses per second) that, if exceeded, generates the *fastRotate* event. When this event is generated, the pulse resolution is adjusted according to the fast pulse resolution factor.

Pulse resolution factor

The factor by which the pulse resolution is adjusted when the jog dial is rotated at a speed greater than the rotation rate threshold, generating a *fastRotate* event. The new pulse resolution will be pulse resolution / pulse resolution factor. Thus, for example, if the pulse resolution is 1 pulse/30° and the pulse resolution factor is 2, at the *fastRotate* event the pulse resolution becomes 1 pulse/15°.

The permissible range for this value is between 1 and (pulse resolution * 10).

More on pulse resolution

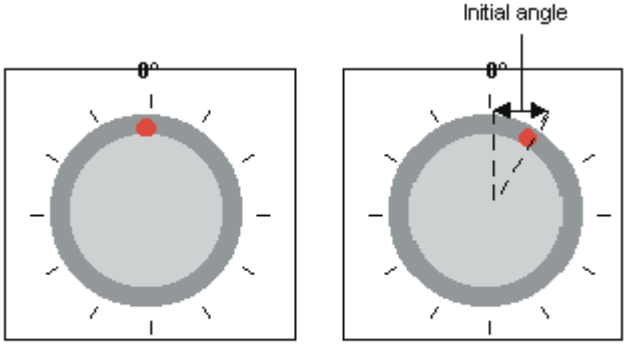


If pulse resolution = 30° and the # of graduations = 12, then one pulse is generated for each rotation equivalent to the distance between two graduations.

❖ *NOTE: It is not necessarily true that the rotation from one graduation to the next generates one pulse. If you want this to be the case, then you have to ensure that the pulse resolution and the number of graduations are properly coordinated. Thus, for example, if you define pulse resolution as 30° and the number of graduations as 6, then* **two** *pulses are generated for each rotation equivalent to the distance between two graduations.*

More on initial angle

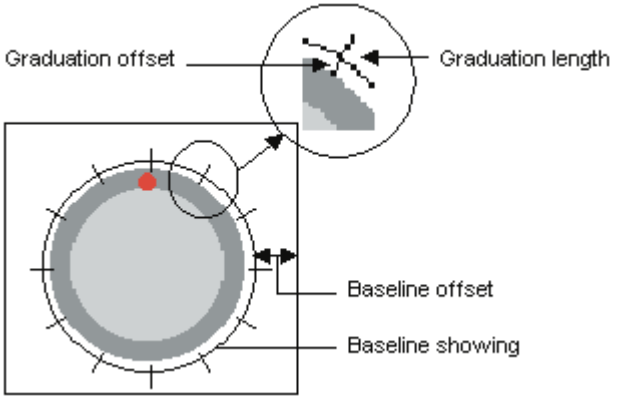
As shown below, the initial angle is always relative to the indicator's vertical position. You can also use the jogdial's *value* property (see p. 13-54) to set the indicator's position through the logic.



Default initial angle = 0 **Initial angle = 30**

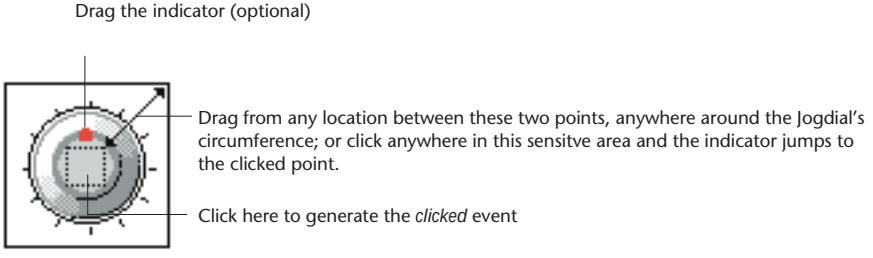
More on scale definitions

The following illustration shows the scale definition settings in more detail:



Jogdial hot spots

You manipulate the Jogdial in the Prototyper or Reviewer in the following ways:



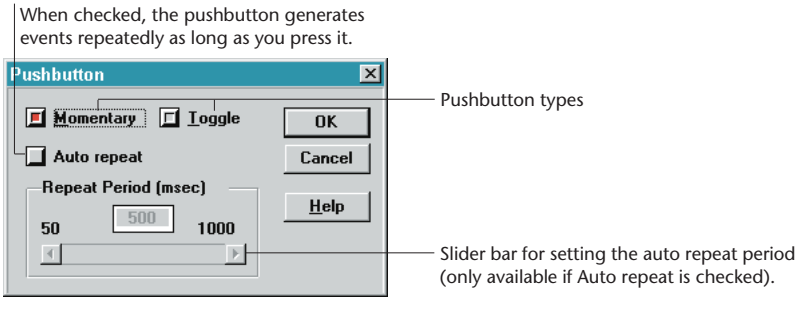


PUSHBUTTONS

There are three types of pushbutton objects:

TYPE		DESCRIPTION
	*Flat*	A rectangular pushbutton that has square corners.
	*Square*	A rectangular pushbutton with a recessed 3-dimensional border framing a protruding interior.
	*Round*	A circular pushbutton with a recessed 3-dimensional border framing a protruding interior.

All pushbuttons use the same dialog box:



You define a pushbutton as one of two types:

- Momentary, i.e., remains pushed in only as long as you press it (default).
- Toggle, i.e., remains pushed in until you press it again.

If a pushbutton is momentary, it can also be defined as auto repeat, i.e., the object generates events repeatedly as long as you press it, at the repeat period you set. For example, if your application requires a clock setup, pressing a pushbutton that has been defined as momentary and auto repeat can automatically increase the clock's seconds/minutes/hours.



SWITCHES

This active object class comprises various switches whose positions are either momentary or constant. A momentary position is one in which the switch is spring-loaded, i.e., will not stay in the position unless it is held. An example is the search-up/search-down switch on a car radio. A constant position is one in which the switch stays in its position without being held. An example is the power switch on a computer.

Some switches (e.g., rotary switch, stepper switch) can have more than two positions, and you can access the positions nonconsecutively. For these switches, when moving from one position to another, Rapid generates events for each intermediate position (i.e., when moving from position1 to position3, Rapid generates the events *position2 make* and *position2 break*). On the other hand, conditions for intermediate positions do not become valid (i.e., the condition position2 *is connected* would not become valid).

There are nine types of switches:


Rocker switch


Vertical toggle switch


Horizontal toggle switch


Rotary switch


Vertical and horizontal slider switches


Cross switch


Stepper switch


Joystick

Changing a position name

All switches, except the stepper switch, have named positions. The procedure for changing a position name applies to all of them.

To change a position name:

1. In the switch's dialog box, select a position name from the list. It appears highlighted in the text box.
2. Overtype the position name. As you type, the name changes in the list. If the selected position is the designated initial position, the name also changes in the Initial Position text box as you type.

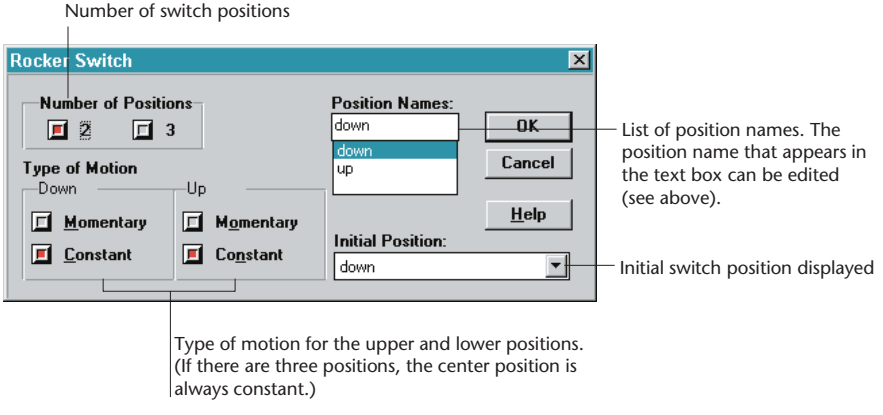
   For switches whose dialog box contains Type of Motion group boxes, the position names also change as you type (see the "Rocker Switches" and "Toggle Switches" sections below).
3. Click OK.



Rocker Switch

The rocker switch object is a vertical two- or three-position switch. A two-position switch can have either both positions constant, or one position constant and the other momentary. A three-position switch's middle position must be constant, but the other positions can either be constant or momentary.

With a three-position toggle switch, you can move the switch from the bottom position to the top position directly, but Rapid will always generate *make* and *break* events for the center position (see the "Switches" section above).





Toggle Switches

The toggle switch object is either a vertical or horizontal two- or three-position switch. The toggle switch objects have the same characteristics as the rocker switch object, but have different graphic appearances. Vertical and horizontal toggle switches use the same dialog box (see “Rocker Switch” above).

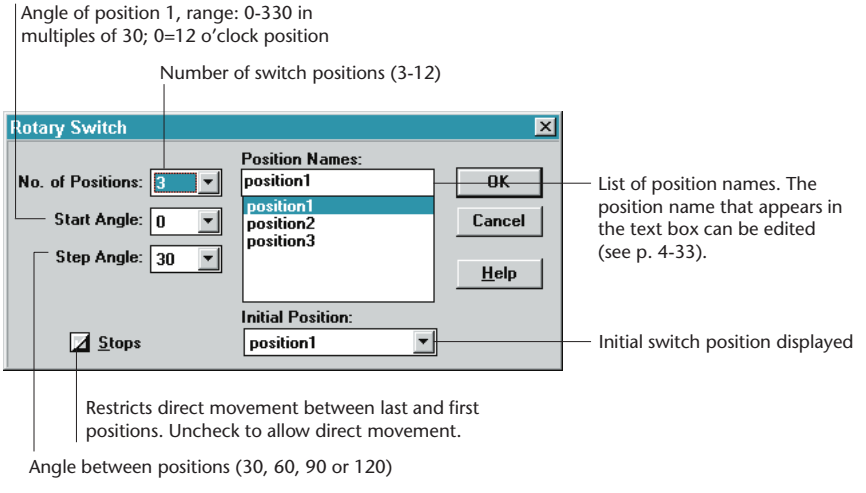


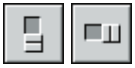
Rotary Switch

The rotary switch object is a multi-position switch with the positions arranged in an arc around a central knob. All positions are constant.

You can move the knob from one position to another position that is non-consecutive, but Rapid will always generate *make* and *break* events for the intermediate positions (see explanation above).

The positions and their names are set in the Rotary Switch dialog box:

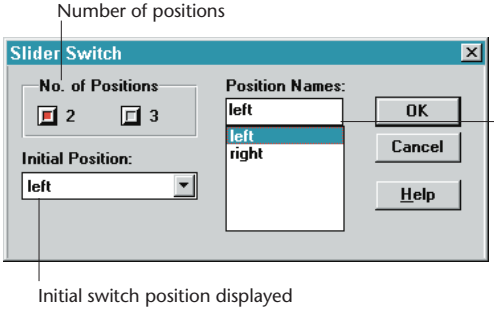




Slider Switches

The slider switch object is either a two- or three-position switch with the positions arranged vertically or horizontally in a straight line. All positions are constant.

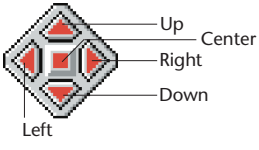
With a three-position slider switch, you can move the switch from the bottom position to the top position (or left to right) directly, but Rapid will always generate *make* and *break* events for the center position (see the "Switches" section above).



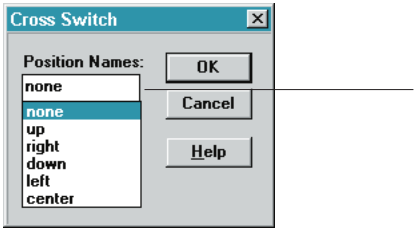


Cross Switch

The cross switch object is a multi-position switch that has five momentary positions (up, down, left, right and center). It also has a "none" position, that is, none of the momentary positions is pressed. "None" is the default position. The momentary positions are illustrated below:



The Cross Switch dialog box is used for changing the position names:

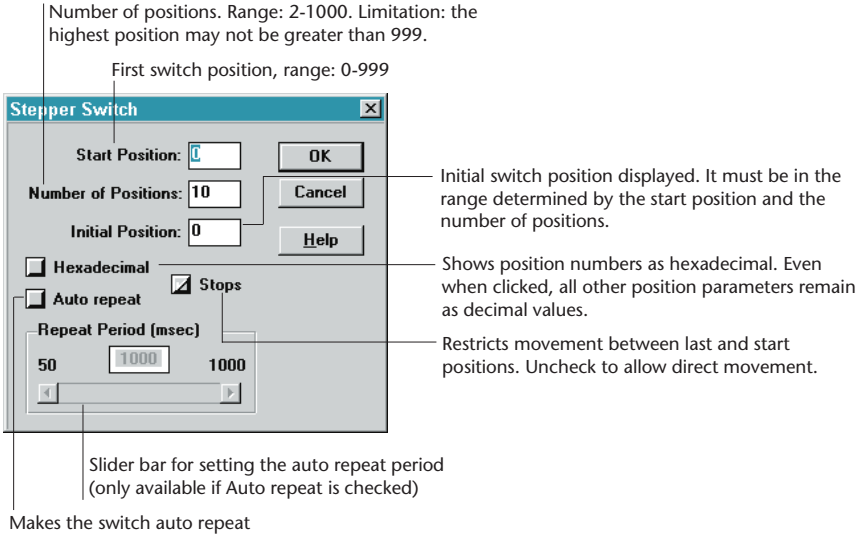




Stepper Switch

The stepper switch object is a switch whose position number is incremented or decremented by clicking its up/down arrows. The current switch position is displayed in a small window located between the arrows. All switch positions are constant. You cannot jump directly between positions that are not consecutive.

Position values are set in the Stepper Switch dialog box:



In the Prototyper, the stepper switch's current position increments or decrements by one each time the arrow is pressed. If you checked Auto repeat however, each time you press and hold an arrow, the position number increases (or decreases) at regular intervals set by the Repeat Period slider bar, until the mouse button is released.

❖ *NOTE: In the Object Editor, you can customize a stepper switch so that the switch position window is not located in the center of the switch (see p. 9-26, "Active Areas Editing").*



Joystick

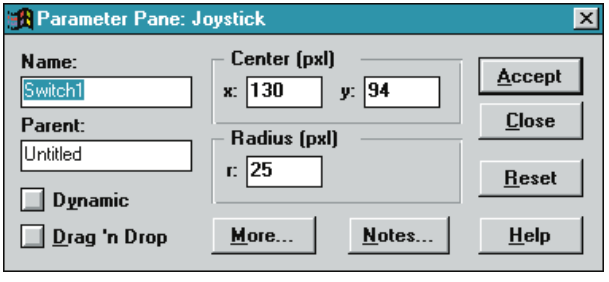
The joystick is an active object which simulates the functioning of a real joystick. It has nine momentary position: up, upLeft, upRight, down, downLeft, downRight, right, left, and center. In addition, it has a "none" position (i.e., none of the positions is pressed). Each momentary position can be defined as "auto repeat" so that it continuously generates events as long as it is clicked (and held).

The joystick object can be used to move other objects or bitmaps across the screen. For example, you can use a joystick to move the bitmap of a car along a road on a map.

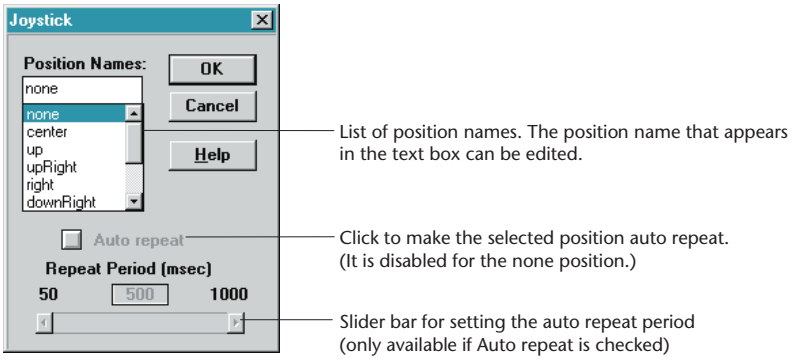
When you add a joystick object to the layout, it looks like:



The joystick is treated like a circular object, and so has a center position and a radius (as shown in the parameter pane below):



Use the Joystick dialog box to change its position names and set any of its momentary positions to auto repeat:



"None" is the default position.

Auto repeat feature

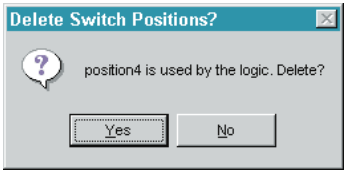
When you set a momentary position to be auto repeat, it generates events repeatedly for as long as you keep it clicked (either manually or through the logic). The rate at which events repeat is set by the Repeat Period slider bar.

You can set all of the momentary positions to be auto repeat; however, you can only set one repeat period.

❖ *NOTE: The Auto repeat check box is disabled for the None position.*

If you delete a switch position that is used in the logic

If you have used a switch position in the application's logic, you can still reduce the switch's number of positions in its dialog box. If, by reducing the number of positions, you would delete the position which is involved in the logic, the following warning appears:



If you choose Yes, the position is deleted. However, the logic statement remains in the Logic Editor, and Rapid replaces the reference to the position with the message <Missing property>.

For example, an application contains a rotary switch with four positions, and there is a condition which checks when the switch is connected to position 4:

& Switch.position4 is connected

If you reduce the number of the switch positions to three, the condition will appear as follows:

& Switch.<Missing property> is connected

When you run the application in the Prototyper, and Rapid tries to operate the logic referring to the deleted position, it will display a runtime error.

If you later increase the position numbers to include the deleted position, the position replaces <Missing property> in the logic statement.



DISPLAYS

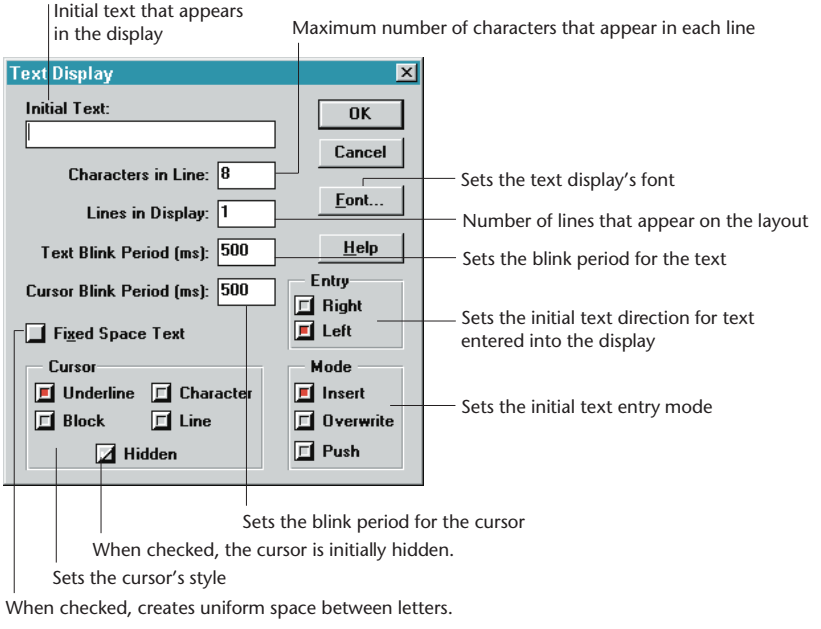
This class of active objects comprises two objects that display strings (i.e., collections of alphanumeric characters): text display and text window.



Text Display

The text display object is generally used for displaying values. For example, you could use a text display object to display the value of a potentiometer, the time, or text entered via the keyboard. Text displays are often used to simulate LCD displays.

A text display object's size, initial text and characteristics are set in its dialog box:



The text display parameters are described in the following sections.

Adding text to a text display

When you add a text display object to the layout, no text appears. You can display text in a number of ways:

- Enter the text display's initial text in its dialog box in the Object Layout.
- Build an activity or action in the application's logic that displays a string.
- Using the keyboard object, enter text directly into the display as the application runs in the Prototyper (see "Keyboard" on p. 5–28).

Setting the text display's size

A text display's initial size is determined by the:

- Characters in Line parameter.
- Lines in Display parameter.
- Parameters set in the Font dialog box: font, font size, font style.

Text display fonts

When using proportional fonts (e.g., Times New Roman), the length of the display is determined by the number of characters multiplied by the width of "W" (the widest letter). Therefore, a display containing several narrow letters will have the same width as a display that contains wide letters. To create a text display in which the letters are the same width (such as for an LCD display), check the Fixed Space Text check box.



Regular display

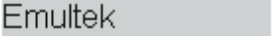


Display with fixed space text

For fixed-width fonts (e.g., Courier), all displays containing the same number of characters are, by definition, the same width.

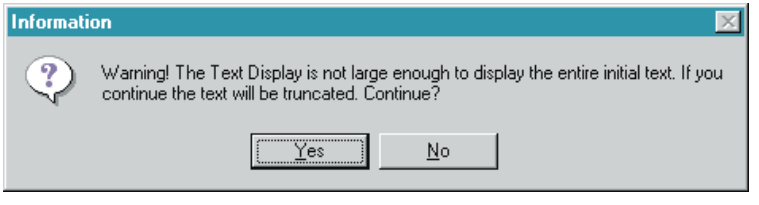
Whenever you change the font, font size and/or style for a text display, your font selection becomes the default for new text displays added to the current application. Your font selection will also be the default for the next new application created during the current working session. However, every time you exit Rapid, the original, default text display font style becomes valid.

If you enter text that is eight or less characters in length, the default text display will appear similar to:



Sample text display

If you enter text that is longer than the total number of characters in the display (that is, the number of characters per line × the number of lines), Rapid warns you that the text display is not large enough:

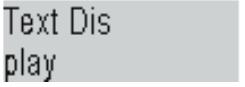


For example, if you enter "Text Display" as the initial text with the Characters in Line parameter set to eight and the Lines in Display set to 1, and then click Yes in the dialog box shown above, the text display will appear as:



and the last four letters will be removed from the Initial Text parameter. To prevent this situation, you should click “No,” then increase the Characters in Line parameter.

Using the same example, if Characters in Line is 8 and the Lines in Display is 2, Rapid will accept text longer than eight characters because it moves the remaining letters down to the next line:



Resizing the text display

You can resize a text display using the above parameters. However, you can also resize it as you would resize any graphic object— by using the parameter pane Size parameter or the text display’s resizing handles (see “Resizing Graphic Objects” on p. 2–28). When using proportional fonts, the font size changes to match the new dimensions.

No matter how large you make the text display however, the number of characters that appear is still determined by the Characters in Line parameter.

Making the text display transparent

You can make a text display transparent, so that you cannot see the display but the text itself remains visible. The text appears to “float” over whatever object is in the background.

To make the text display transparent:

1. Select the text display.
2. Choose Edit|Colors to open the Colors Edit dialog box.
3. Click the Transparent check box to check it.

Text entry mode

There are three text entry modes:

MODE	DESCRIPTION
Insert (default)	For Left entry, Rapid moves existing characters to the right to make room for a new string. The cursor moves one position to the right of the new string. For Right entry, Rapid moves existing characters to the left. The cursor moves one position to the left of the new string. Insert mode is used for regular typing.
Overwrite	New text replaces existing text, character by character. The cursor moves one position to the right for each character (or left for Right entry).
Push	Same as Insert mode except that the cursor remains in the same position. Push mode is generally used in conjunction with Right entry for calculator-style numeric entry.

Text entry direction

There are two text entry directions:

DIRECTION	DESCRIPTION
Right	Characters are entered starting at the right-most position, and the order of the characters in the text string is reversed.
Left (default)	Characters are entered starting at the left-most position.

Cursor style

You can display the text display cursor in one of four styles, or hide it altogether. The cursor styles are:

CURSOR STYLE	APPEARANCE
Underline (default)	A line under the cursor position character.
Block	A block around a character. The character appears in inverse video.
Character	The character at the current cursor position is itself the cursor. This style is only seen when the cursor is blinking.
Line	A vertical line at the cursor position.
Hidden (default)	Hidden. If this check box is checked, it overrides the other four styles. Even if you select a new cursor style in the Logic Editor, the cursor will still be hidden.

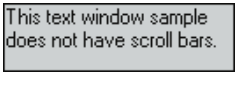
Blink periods

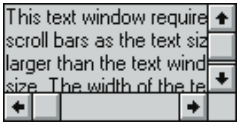
You can make the text and/or cursor blink by using the *blink* function in the Logic Editor. The initial blink period (i.e., the period of time in which the text/cursor is shown plus the time it is hidden) is set in the Text Display dialog box. For example, if the cursor's blink period is 500 ms, the cursor will blink on for one quarter of a second and off for one quarter of a second.



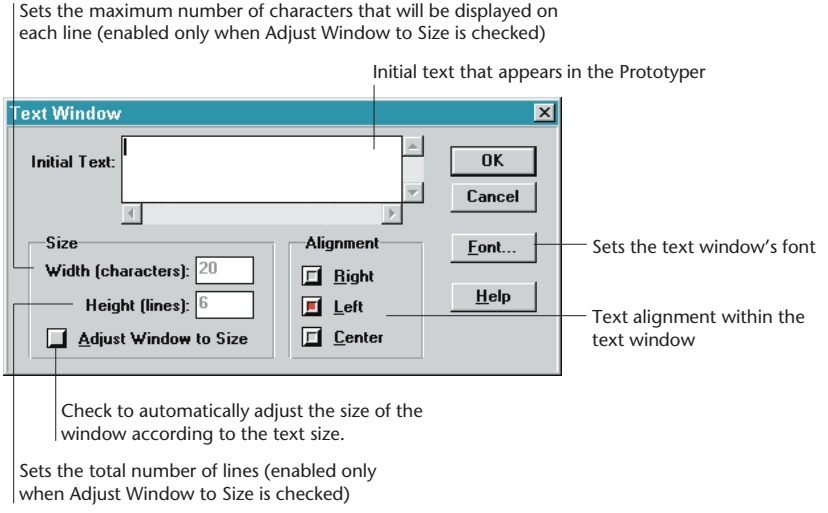
Text Window

The text window object is generally used for displaying long strings inside a window, with or without scroll bars. Examples of text windows are shown below:





You set the appearance of the text (i.e., text size, alignment and font) in the Text Window dialog box.



Adding text to a text window

When you add a text window object to the layout, no text appears. You can enter text in three ways:

- Enter the text window's initial text in its dialog box in the Object Layout.
- Build an activity or action in the application's logic that displays a string in the text window.
- Using the keyboard object, enter text directly into the text window as the application runs in the Prototyper (see "Keyboard" on p. 5–28).

Adding initial text

You can enter initial text into the Text Window dialog box in two ways: (i) by entering the text directly, or (ii) by preparing it in another Windows program and pasting it into the Text Window dialog box.

❖ *NOTE: When you enter text into the Text Window dialog box, you do not have to be concerned with line breaks because the text automatically wraps around.*

To add initial text to a text window:

1. Open the Text Window dialog box.
2. Enter text into the Initial Text text box.

3. Click OK. The text window is redrawn on the layout to include the text.
4. To view the text, run the application in the Prototyper.

To add initial text prepared in another Windows program:

1. Open the file in the other Windows program and copy the text.
2. Return to Rapid and paste the text into the Initial Text text box.

A text window's appearance

A text window's default size is 50 @ 50 pixels. No matter how much text you enter into it, the text window remains the same size (until you resize it). If more text is entered than can fit, Rapid adds scroll bars.

A vertical scroll bar is added when there is too much text; a horizontal scroll bar is added when a word is too long to fit in the window. (The text window does not break up words.)

The default font is MS Sans Serif, regular, 9 pt. You can change the font in the Font dialog box. Changing the font only affects the selected text window; the next one you add to the layout will have the original, default font. If the font size is too big for the text window, Rapid adds scroll bars.

❖ *NOTE: The size of a text window is not increased when a scroll bar is added. A horizontal scroll bar takes up half a line of text. To view the entire text, resize the window.*

Resizing the text window

There are three ways to resize a text window:

- Using the resizing handles.
- Using the parameter pane's Size parameter.
- Using the Text Window dialog box's Size parameters.

❖ *NOTE: Resizing a text window does not affect the text's font size.*

To resize a text window using the Text Window dialog box:

1. Check the Adjust Window to Size check box. The Width and Height parameters are enabled.
2. Enter the appropriate numbers in the Width and Height text boxes. (Refer to "More about resizing using the dialog box parameters" below.)

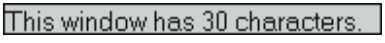
3. Click OK. The text window is redrawn on the layout at the new size.

More about resizing using the dialog box parameters

When you use the Text Window dialog box to set a text window's size, the width of a line of text is determined by the number you enter in the Width (characters) text box. This number does not refer to the actual characters you enter into the Initial Text text box, but to "W" characters (the widest letter). For example, if the width is set at 15, Rapid interprets that to be 15 W characters. If the actual characters are narrower than W, the text window will accept more than 15 characters.



Width set at 15 characters



Same width setting

The height of a line is determined by the font size.

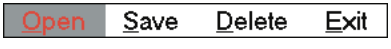
❖ *NOTE: In the Prototyper, a text window will always appear in the foreground of the screen, even if you moved it to the back.*



MENU

This active object class comprises one object, the menu object. A menu object looks and behaves similarly to an opened menu. The menu can be either vertical (default) or horizontal:



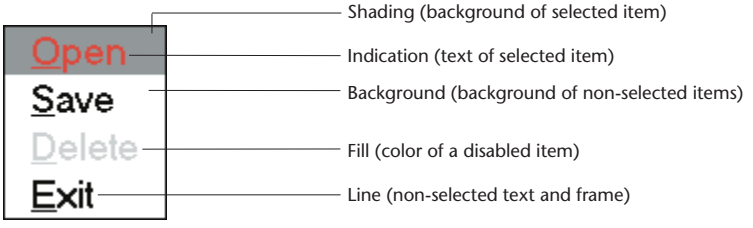


One letter of each menu item's text can be assigned as a mnemonic, i.e., a hot key that allows the user to select the item in the Prototyper via the keyboard. Each menu item is a property of the menu object in the Logic Palette.

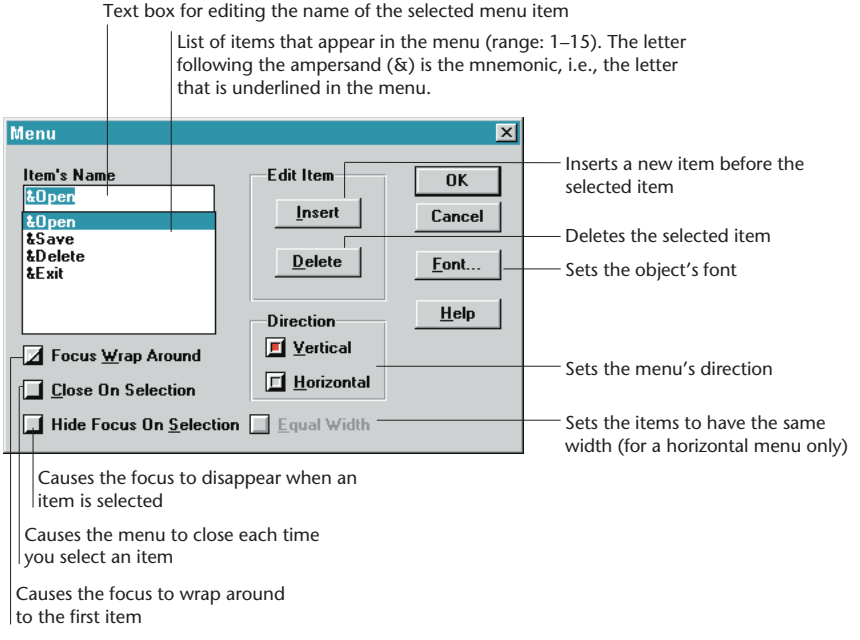
❖ *NOTE: Menu object mnemonics are activated by typing the letter only, without pressing Alt. (Pressing Alt activates Rapid's menu bar.)*

A menu's height is determined by the number of menu items and its width is determined by the length of the longest item text; you cannot resize it. The maximum number of items is fifteen.

The color scheme applied to the menu object is:



Use the Menu dialog box to add or delete items and to change the characteristics of the focus:



Editing an item's name

To edit an item's name:

1. Select the item from the item list.
2. Edit the name in the text box.
3. Add an ampersand (&) **before** the mnemonic letter.

Inserting a new item

To insert a new item:

1. Select the item that you want to appear immediately **below** the new item.
2. Click the Insert button; a blank line appears above the selected item.
3. Enter the new item's name, including an ampersand (&) for a mnemonic if necessary.

Menu options

The Menu dialog box has several options that affect the menu's appearance and behavior:

OPTION	DESCRIPTION
Direction	Determines whether the menu will be a vertical (default) or horizontal menu.
Equal Width	(For horizontal menus only.) Sets the width of each menu item to be the same. The width is determined by the longest item.
Focus Wrap Around	When checked, causes the focus to wrap around to the first item. That is, if the focus is on the last item and the down arrow is pressed, the focus moves to the first item.
Close on Selection	When checked, causes the menu to close each time you select an item.
Hide Focus on Selection	When checked, causes the focus to disappear when an item is selected.

❖ *NOTES: Don't name an item "focus" because this name is used by the focus property.*

You can delete an item that is used in the logic, but if you do so, the phrase <Missing property> appears in place of the item (see "If you delete a switch position that is used in the logic" on p. 4–38).



ANIMATED OBJECT

The animated object runs through a sequence of bitmaps to produce an animation. Just like an animator creates a set of frames which animates a cartoon character, you can create a set of frames (bitmaps) that can be "flipped" through at a specified speed to achieve the illusion of movement.

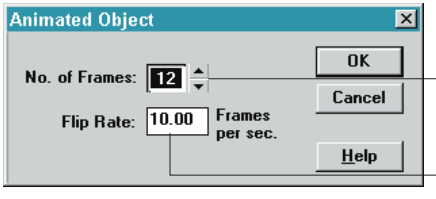
The animated object has its own object class in the Object Palette. When you add an animated object at its default size, it looks like:



Frame 1 of 12 frames.
This set of frames creates an animation of a ball moving down and up.

The animated object is a non-circular object with x and y values for position and size. It is dynamic (i.e., the dynamic check box is checked and cannot be unchecked).

Use the Animated Object dialog box to set the number of frames and the rate at which the frames will be flipped:



Frame: a bitmap that comprises an element of an animated sequence (2–12)

Number of frames displayed per second (0.01–16)

Changing the Animated Object's Bitmaps

You can change the animated object's bitmaps in the Object Editor by editing its frames.

❖ *NOTE: You cannot create a new animated object in the Object Editor, only edit an existing one.*

To change the default animated object's bitmaps:

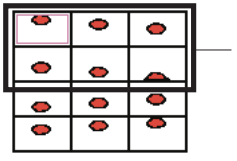
1. Select the object in the Object Layout, then choose Objects|Edit Object (Ctrl+E). The Object Editor opens with the first bitmap displayed.

Bitmaps displayed in the Bitmap Selector box

All twelve frames are displayed in the Bitmap Selector box, even if you set the number of frames to be less than twelve in the Animated Object dialog box.

Example

You set the number of frames to be six. When you edit the bitmaps, edit only the first six bitmaps and ignore the rest.



2. To draw **new** bitmaps, use the editing tools (pencil, line, etc.). To use **existing** bitmaps, copy and paste them. (See the following two sections.)

Using the editing tools

All of the editing features available in bitmap-editing mode can be used to edit the animated object's bitmaps. Since animated objects do not have active areas or pointers, the Active Areas and Pointer modes are disabled. The Test mode is also disabled. If you want to test the object's animation, you must run the application in the Prototyper.

Using existing bitmaps

You can copy existing bitmaps from Rapid or other Windows graphic programs and paste them into the animated object's frames.

To use an existing bitmap:

1. Copy a bitmap either in the Object Layout or in another Windows graphics program.
2. In the Object Editor, select the frame from the Bitmap Selector box in which you want the copied bitmap to appear. Choose Edit|Paste (Ctrl+V).

   The bitmap appears in the upper-left corner of the edit area.
3. Click and drag the pasted bitmap anywhere in the edit area, then release the mouse button to fix the position.

Setting the animated object's colors

The animated object's colors can be changed only in the Object Editor. However, if you want to make one of the colors transparent, use the Colors Edit dialog box in the Object Layout.



FLOW OBJECTS

This class of active objects allows you to simulate the storage and movement of a substance or item along a set path. The substance could be a liquid (e.g., water or fuel), a fluid-like material (e.g., gas), or even the movement of traffic along a road or people through a corridor! To create this effect, you:

(a) Add a series of pipes to define a path along which the material will move.

(b) Define the relative speed of the flow along each pipe.

(c) Assign a flow pattern for each pipe that will simulate the movement of the material in the Prototyper.

(d) Add valves to direct the flow between the different pipes.

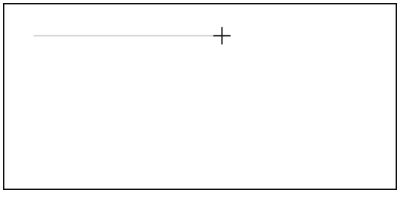
(e) Add tanks (see p. 4–56) at strategic points between the pipes to simulate reservoirs, fuel tanks, etc.



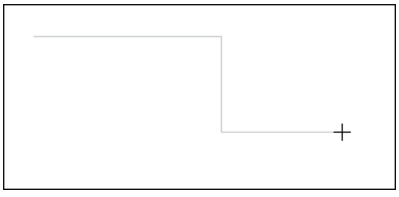
Pipe

To draw a pipe:

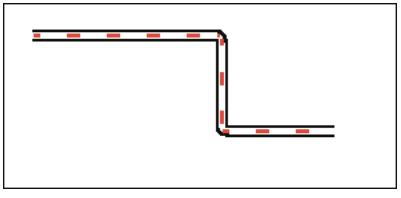
1. Position the crosshairs at the point where you want the pipe to start. Click and drag the endpoint of the line to the desired position. The line you draw snaps to a horizontal or vertical position.



2. Release the mouse button; you have drawn the first section of the pipe. The crosshairs remains at the end position to allow you to click and drag again to draw the next pipe section (at a 90º angle to the first section).



3. Double-click when you have completed drawing the path of the pipe; Rapid displays the completed object, showing its flow pattern:



Once you have begun to add a pipe object, you cannot choose any other object or menu option until you complete the above procedure. A single pipe object must be continuous, without any splits, but can have any number of 90º angles. You cannot create joints such as a "T" joint from a single pipe.

❖ *NOTE: You cannot make a pipe object dynamic nor drag 'n drop.*

To create a "T" joint:

1. Add two separate pipes of the required length at a 90º angle to each other.
2. Position the two pipes so that the end of one slightly overlaps the edge of the other (use the Shift+Arrow keys for fine positioning).
3. Select one of the pipes, then choose Layout|Move to Back or Move to Front.

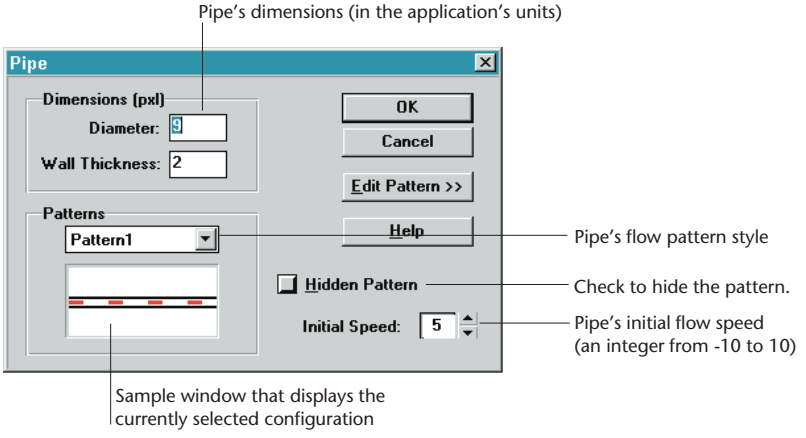
To create a "+" joint:

- Follow the procedure explained above for creating a "T" joint.

In the Pipe dialog box you can set the following aspects of the object:

- Thickness of the pipe wall.
- Diameter of the pipe.
- Initial speed of the flow through the pipe.
- The pipe's flow pattern.

The units of thickness and diameter are given in the application's scale units. The units of the flow speed are **relative** units, not absolute, and range from -10 to 10.



Setting the initial flow speed and direction

A pipe's flow by default moves in the direction in which you drew the pipe. For example, if you drew the pipe from left to right, its flow will be from left to right. If you wish to obtain a flow in the opposite direction, set the flow speed to be negative. A speed of zero produces a stationary flow pattern.

You can alter the flow's speed in the application's logic.

Setting the initial flow pattern

You can assign one of six patterns to the pipe's flow. The patterns are non-graphic system objects that Rapid adds to the application only after you have added a pipe. You can edit the pattern itself (the pattern style and its colors) in the Object Layout and in the application's logic.

The pattern you assign now is only the initial pattern—the pipe object has logic that allows you to change from one pattern to another.

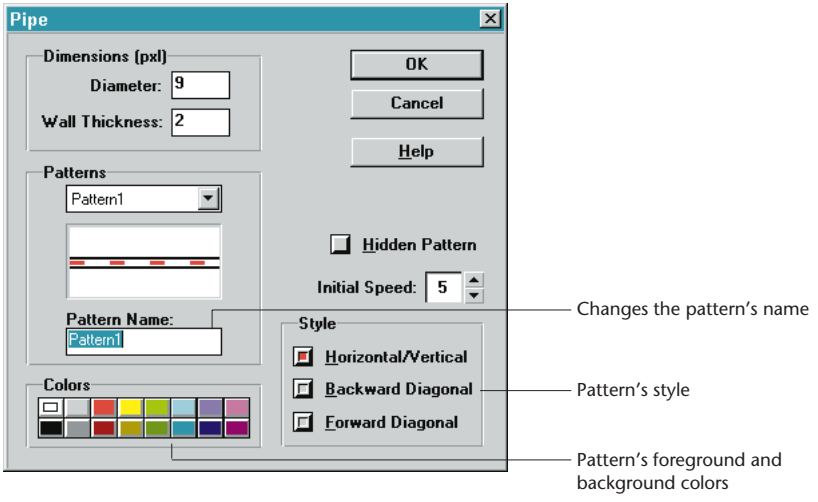
To select a pattern:

- Select a pattern from the drop-down list.

Editing a Pattern

To edit a pattern:

- Click the Edit Pattern button; the Pipe dialog box unfolds to show the Edit Pattern options:



Pattern styles

Click a style radio button to select a flow pattern style. The styles available are:

- Horizontal/ vertical (default). If you select this option, Rapid assigns a horizontal pattern to a vertical pipe and vice versa.
- Backward diagonal
- Forward diagonal



Backward diagonal | Vertical or Horizontal | Forward diagonal

Pattern colors

To select the forecolor (the color of the stripes):

- Click a color from the palette.

To select a background color:

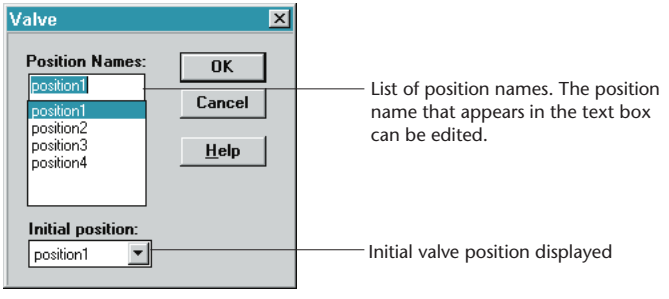
- Right-click a color from the palette.

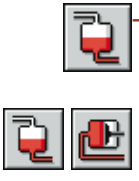


Valves

Rapid provides five valve objects with which you direct the flow between one pipe (or set of pipes) and another. The valve objects behave in a similar fashion to switch objects except that the number of positions for each valve type is fixed. Clicking a valve's active area causes the valve to turn so that the pointer (by default, colored red) connects to that active area.

The number of positions for each valve is fixed. In the Valve dialog box, you can edit the default names of the valve's positions:

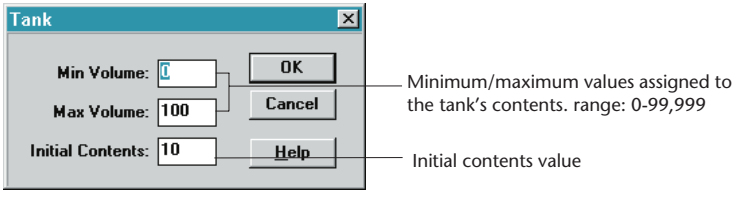




TANKS

This class of active objects contains two objects: a vertical and horizontal tank. They are used to simulate the storage and/or collection of a moving fluid. You can add tank objects at strategic points between pipes to simulate reservoirs, fuel tanks, etc. The horizontal tank can also be used to simulate a piston or an actuator (e.g., in a hydraulic pump).

The vertical tank fills from the bottom upwards (and empties from the top downwards), while the horizontal tank fills from left to right (and empties from right to left). They use the same dialog box:



In the Logic Editor, you can change the value of the tank's *contents* function to produce the effect of a tank filling or emptying.

CHAPTER 5

Nongraphic Objects

Nongraphic objects are objects that have no graphic representation on the layout, but play important "behind the scenes" roles in an application's behavior.

❖ *NOTE: Rapid automatically generates the System nongraphic objects. They appear in every application.*

In this chapter you will read about:

- Adding nongraphic objects.
- Managing existing nongraphic objects (renaming, editing, copying, deleting and replacing).
- The classes and types of nongraphic objects. The objects are presented in the order in which they appear in the object palette.

❖ *NOTE: Unlike active objects, an initial dialog box opens when you add a nongraphic object. Unless otherwise specified, the parameter values shown in this chapter are the default values.*

Multimedia and communications objects

There are two specialized classes of nongraphic objects which are documented in separate chapters:

- multimedia objects: these objects use the standard Microsoft® Windows Media Control Interface (MCI) so that you can include multimedia features in your Rapid applications. The objects are explained fully in Chapter 6, "Multimedia Objects".
- communications objects: these objects take advantage of the dynamic data exchange (DDE) mechanism to allow your Rapid application to communicate with other Windows programs or DLLs. The objects are explained fully in Chapter 7, "Communication Objects".

ADDING NONGRAPHIC OBJECTS

To add a nongraphic object to the Object Layout:

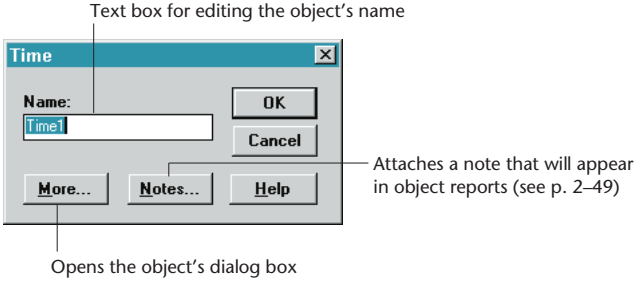
1. Select the object you want to add. To do so, either:

   Choose Objects|Add, then select the object from the New Objects list.

   Or:

   From the Object Palette, select an object class in the left column and the object in the right column.

2. A dialog box, like the one shown below, opens (this dialog box is used by all the nongraphic objects for adding the object to the application):



   - If you click OK, the object is added to the application with its default name and default values.
   - If you want to modify its name, make changes in the Name text box and then click OK.
   - If you want to modify its values, click the More button to open the object's dialog box. (Refer to each object's section in this chapter for an explanation of its parameters.)

❖ *NOTE: If you want to change the object's name, you do so in the Nongraphic Objects dialog box (see "Managing Nongraphic Objects" below).*



3. After you finish adding the object, the nongraphic icon appears in the top-left corner of the layout frame.

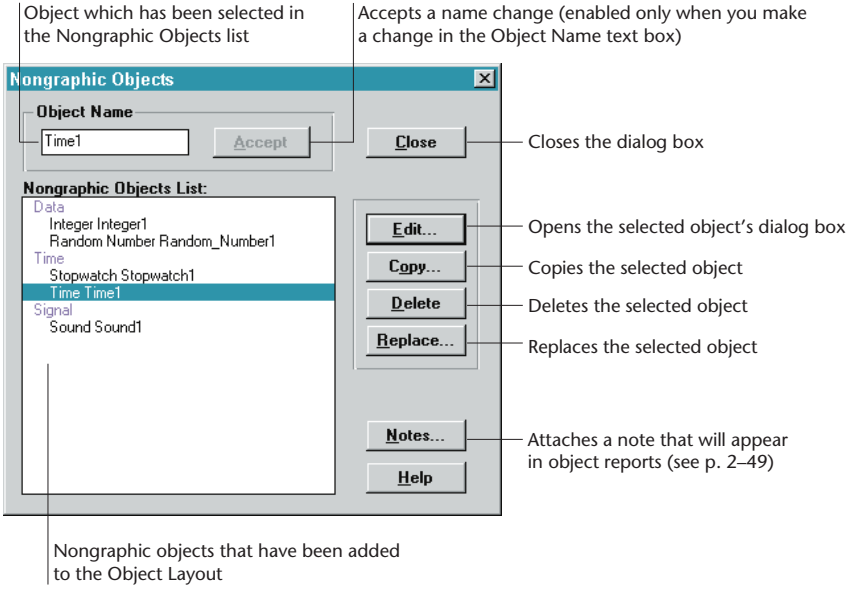
❖ *NOTE: You can drag the nongraphic icon to a more convenient location on the layout if it's in the way.*

MANAGING NONGRAPHIC OBJECTS

Once you have added a nongraphic object to the layout, you can rename, edit, copy or delete it, or replace it with another nongraphic object, using the Nongraphic Objects dialog box.

To use the Nongraphic Objects dialog box:

1. Either choose Edit|Nongraphics or double-click the nongraphic icon; the Nongraphic Objects dialog box opens (a sample dialog box is shown below).



2. Select an object from the Nongraphic Objects list; its name appears in the Object Name text box. The management buttons (i.e., Edit, Copy, etc.) are enabled.
3. Perform the necessary changes. The possible operations are described in the following sections.

❖ *NOTE: The System objects (i.e., ASCII, Keyboard, SystemDate and SystemTime) do not appear in the Nongraphic Objects list since you cannot edit them.*

Renaming a Nongraphic Object

The object's name is used to identify it in the Object Layout and Logic Editor. By default, the object name consists of the object type and a sequential number. (See p. 2–11 for rules about valid object names).

To rename an object:

1. Select the object in the Nongraphic Objects list. Its name appears in the Object Name text box.
2. Modify the name in the text box, then click Accept.

Editing a Nongraphic Object

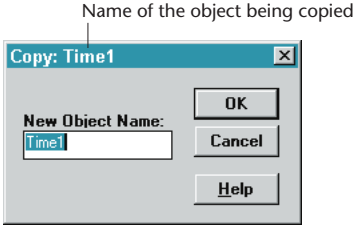
To edit a nongraphic object:

1. Select the object in the Nongraphic Objects list.
2. Click the Edit button or double-click the object; the object's dialog box opens.
3. Make the necessary changes, then click OK.

Copying a Nongraphic Object

To copy a nongraphic object:

1. Select the object in the Nongraphic Objects list.
2. Click the Copy button to open the Copy dialog box:

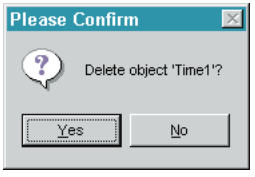


3. Enter a name for the new object, then click OK.
4. The copied object appears in the Nongraphic Objects list. You can select it and edit its parameters.

Deleting a Nongraphic Object

To delete a nongraphic object:

1. Select the object in the Nongraphic Objects list.
2. Click the Delete button; Rapid asks you to confirm the delete operation:



3. Click Yes.

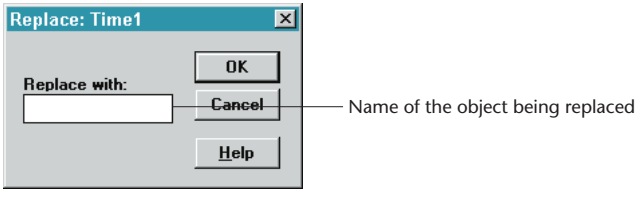
   (If the object is used in the logic, Rapid warns you that deleting the object will delete the related logic. You can continue or cancel the delete operation.)

Replacing a Nongraphic Object

You can replace a nongraphic object with another object of the same type (e.g., a stopwatch with another stopwatch). This feature is especially useful if you have already built logic around the object that you want to replace. Without it you would have to delete the object, together with its logic, add a new object and build the logic from scratch.

To replace a nongraphic object:

1. Select the object in the Nongraphic Objects list.
2. Click the Replace button to open the Replace dialog box:



3. Enter the name of an object (of the same type) that appears in the Nongraphic objects list, then click OK.

4. The replaced object is removed from the list.

The following sections describe (by class) the types of nongraphic objects, as well as the parameters unique to each type.



DATA OBJECTS

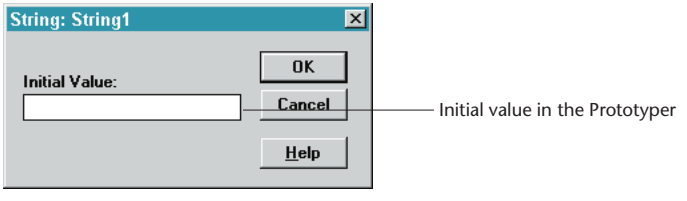
This class of nongraphic objects comprises objects that hold various kinds of data in a variable. There are eight types of data objects: string, number, integer, random number, holder, array, data store and point.



String

The string object is a variable that holds a string (i.e., a collection of of alphanumeric characters). If you use a string as part of an arithmetic procedure, Rapid automatically converts it into a number. For example the character '3' is converted into the numeric value 3.0.

The String dialog box opens when you add or edit a String object. It is used to assign or change the String object's initial value.



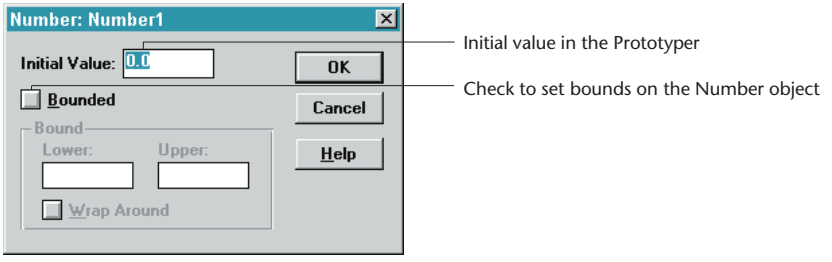
❖ *NOTE: The string dialog box also opens when you define strings in other circumstances (e.g., as a string property for an active primitive object, as a user property for a user-defined object).*



Number

The number object is a variable that holds a real number (e.g., 5.32). If you enter an integer, Rapid converts it to a real number (e.g., 3 is converted to 3.0).

The Number dialog box opens when you add or edit a number object. It is used to assign or change the number object's initial value and, optionally, to set bounds on the object.

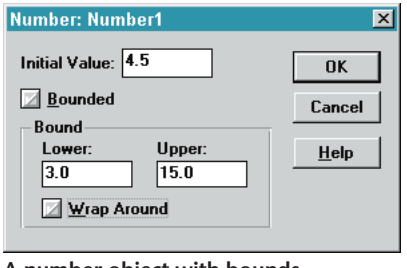


❖ *NOTE: The number object is accurate to six decimal places.*

Bounded number

If you check the Bounded check box, you can set lower and upper bounds on the number object. Setting bounds prevents Rapid from assigning numbers less than the lower bound or greater than the upper bound. For example, if the upper bound is set at 15.0 and the application logic assigns a value of 16.3, the actual value of the number object will be 15.0.

Setting bounds is useful when you are dealing with numbers that are limited to a specific range.



A number object with bounds

Wrap around number

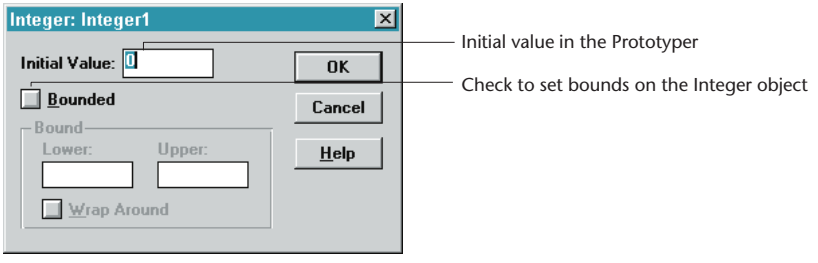
If you check the Wrap Around check box, Rapid "wraps around" to the lower bound when the value of the number exceeds the upper bound. For example, the lower limit is set at 3.0, the upper limit is set at 15.0, and the value of the number object is currently 15.0. If the activity, *changeBy: 1.2*, produces the value, 16.2, the actual value of the number object will be 4.2.

❖ *NOTE: The Number dialog box also opens when you define numbers in other circumstances (e.g., as a number property for an active primitive object, as a user property for a user-defined object).*



Integer

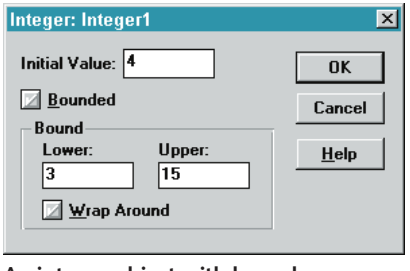
The integer object is a variable that holds an integer. The Integer dialog box opens when you add or edit an integer object. It is used to assign or change the object's initial value and, optionally, to set bounds on the object.



Bounded integer

If you check the Bounded check box, you can set lower and upper bounds on the integer object. Setting bounds prevents Rapid from assigning integers less than the lower bound or greater than the upper bound. For example, if the upper bound is set at 15 and the application logic assigns a value of 17, the actual value of the integer object will be 15.

Setting bounds is useful when you are dealing with integer objects that are limited to a specific range.



An integer object with bounds

Wrap around integer

If you check the Wrap Around check box, Rapid "wraps around" to the lower bound when the value of the integer exceeds the upper bound. For example, the lower bound is set at 3, the upper bound is set at 15, and the value of the

integer object is currently 15. If the mode activity, *changeBy: 1*, produces the value, 16, the actual value of the integer object will be 3.

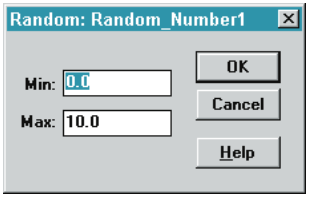
❖ *NOTE: The Integer dialog box also opens when you define integers in other circumstances (e.g., as an integer property for an active primitive object, as a user property for a user-defined object).*



Random Number

The random number object is a read-only number that is assigned a random value by Rapid. The random number object produces a number value with up to six decimal places. Random object has all the functions of the number object, except := (assign) and *changeBy:* since only Rapid assigns the values.

You can set minimum and maximum values to a random number object in its dialog box:



Just like the number object, entering bounds prevents Rapid from assigning numbers greater than the maximum value or less than the minimum value. These values cannot be changed in the Logic Editor.

When you start the Prototyper, Rapid generates a random number. The *next* function is used to generate a new random number. (The number remains constant until the *next* function.)

❖ *NOTE: Do not expect to get the same random number value after you save an application, since the value changes when you load the application.*



Holder

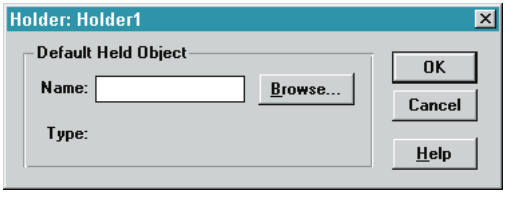
Holder objects are used for "holding" active objects, active primitive objects and groups. They are useful in applications containing several objects of the same type that use the same logic. Instead of writing the logic for each object, you can write the logic once for the holder and then instruct the holder to hold each object in turn.

Holders can also dynamically generate and delete objects in runtime. A holder possesses the properties and functions of the object being held as well as its own functions.

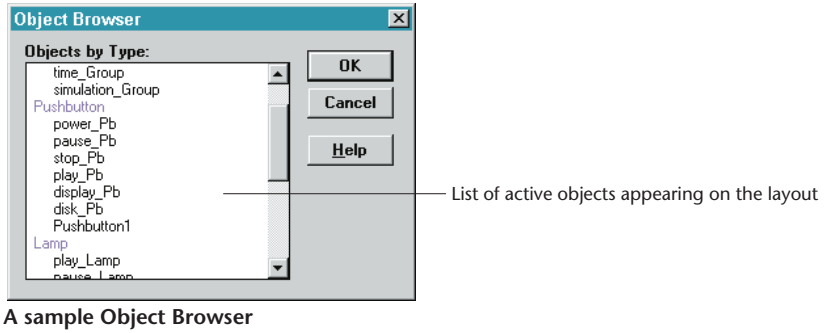
An application can contain more than one holder, but only one object can be held in each holder. In the application's logic, you can only replace the object that is initially stored in the holder with objects of the same type.

To assign an object to a holder:

- In the Holder dialog box, enter the name of an active object:



If you do not know the object's name, click the Browse button to open the Object Browser:



A sample Object Browser

The active and group objects are listed by class. Select an object, then click OK. You are returned to the Holder dialog box. Notice that the object name and type are displayed.



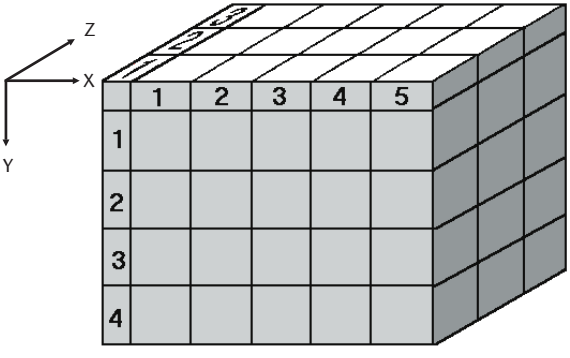
Array

The array object is a multi-dimensional matrix that can contain the following types of data:

- Integer
- Number
- String
- Objects

The array is useful when a value is determined by multiple variables, such as an array of temperatures indexed by latitude, longitude and altitude.

The illustration below uses a 3-dimensional array to schematically represent the structure of a multi-dimensional array object. For obvious reasons, however, the geometrical representation of multi-dimensional arrays breaks down after the third dimension.



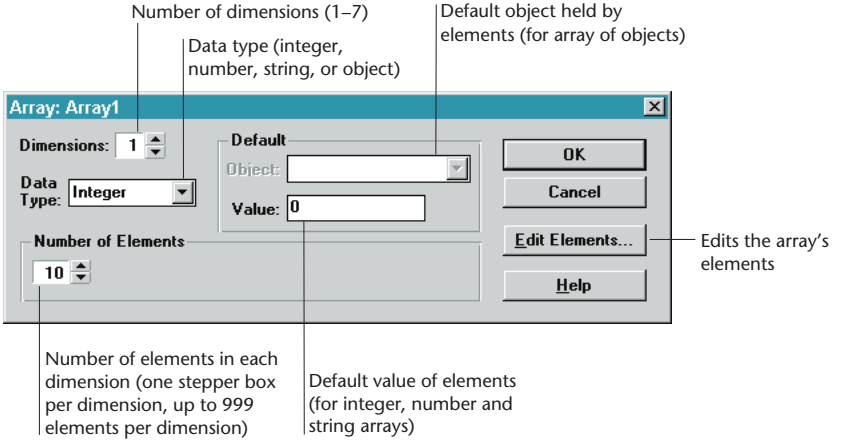
Array of objects

An array of objects is a special kind of array that contains a specified type of graphic object (graphic objects are active objects, active primitive objects and groups). For example, you can create an array of text displays or an array of lamps.

An array of objects allows you to carry out identical logic on a series of different objects of the same type by referring to array indexes (made up of integers) rather than to specific objects. It can also generate copies of existing objects, and delete generated objects, in runtime.

Creating an array

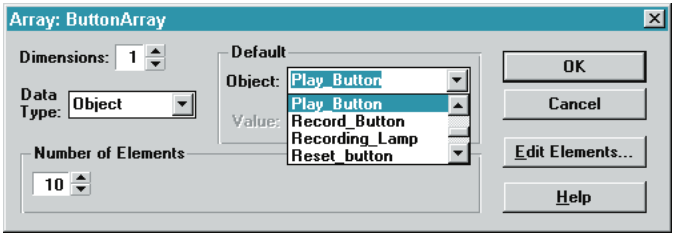
When you add an array, the following dialog box opens:



To define an array:

1. Define the array's structure by setting the number of dimensions (up to seven).
2. Select the type of data that the array is to hold (integer, number, string, or object).
3. Provide a default value.

   If you are defining an object array, enter a default object for the array to hold by selecting an object from the drop-down list. The list contains all the application's active objects and groups.



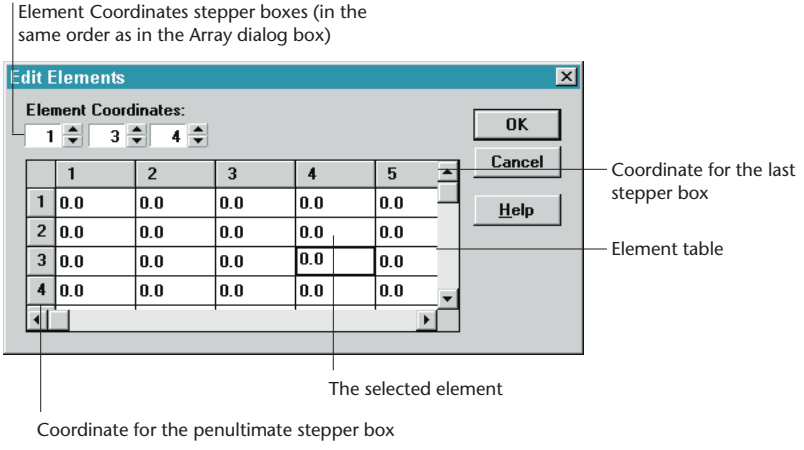
4. Set the number of elements in each dimension.

Editing an array's elements

Initially, all the array's elements have the same value or hold the same object—the one defined as the default value/object in the dialog box. You can edit the value of individual elements so that their initial values will not all be equal.

To edit individual elements (in a data array):

1. Click the Edit Elements button to display the elements. The Edit Elements dialog box shown below is of a three-dimensional array:



   The numbers appearing in the Element Coordinates stepper boxes comprise the selected element's index (in the illustration above, [1,3,4]). The last two boxes display the coordinates in the Element table. The default value appears in each element.

2. Select an element. You can select an element in three ways:

   (a) Set the Element Coordinates stepper boxes to the appropriate index.

   (b) Click a coordinate directly in the Element table.

   (c) Press Ctrl + the arrow keys to move among the elements.

3. Overtype the default value.

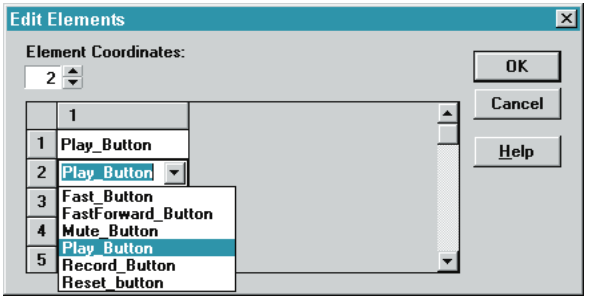
4. Repeat steps 1–3 for each value you want to change.

To edit individual elements (in an object array):

1. Click the Edit Elements button.
2. In the Edit Elements dialog box, you can edit the default elements in two ways:

   (a) Type object names directly into the table.

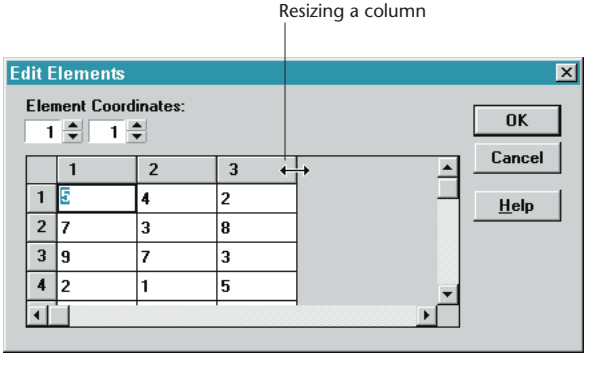
   (b) When you select an element, it becomes a drop-down list that contains all application objects of the same type as the default object.



For example, if the default object is Play_Button, you can edit the elements so that element 2 holds Mute_Button, element 3 holds Record_Button, etc.

Changing the width of element cells

The columns in the array's Edit Elements dialog box are resizable:





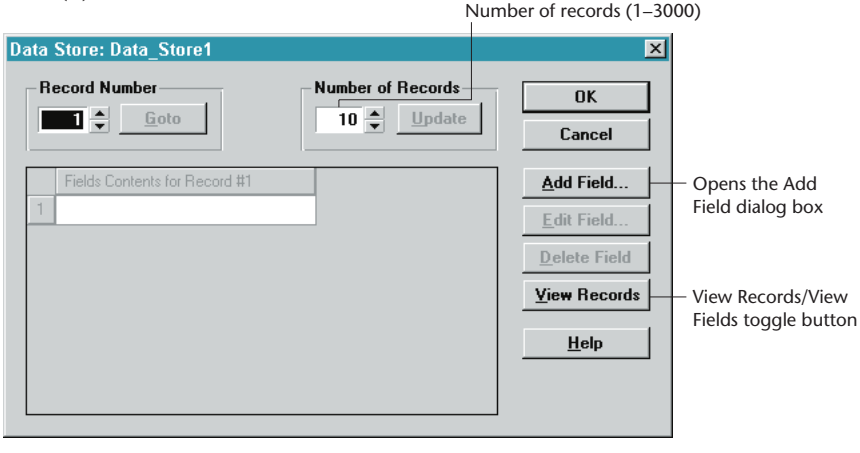
Data Store

A data store object stores multiple records, each of which is made up of various fields. A field can be either an integer, number or string, with the same rules that apply to integer, number and string objects. A single data store can contain all the field types. In the Logic Palette, each field appears as a property of the data store. The illustration below shows a schematic representation of the structure of a data store.

	Field 1 (string)	Field 2 (number)	Field 3 (integer)	Field 4 (string)	- - - →
Record 1					
Record 2					
Record 3					
Record 4					
↓					

Creating a data store

In the Data Store dialog box, you (a) define how many records it will store and, (b) add and edit the fields.



To change the number of records:

1. Click the up/down arrow of the Number of Records stepper box or type the number directly. The Update button is enabled.
2. Click Update to accept the number change.

   If you increased the number, Rapid increases the number of records accordingly.

   If you decreased the number, Rapid warns you that doing so will delete the records above the selected number, and you can cancel the decrease records operation.

❖ *NOTE: Each time you change the number of records, be sure to click Update.*

Viewing data: two alternatives

Rapid offers you two ways to view a data store's contents—the fields view and the records view:

Fields View

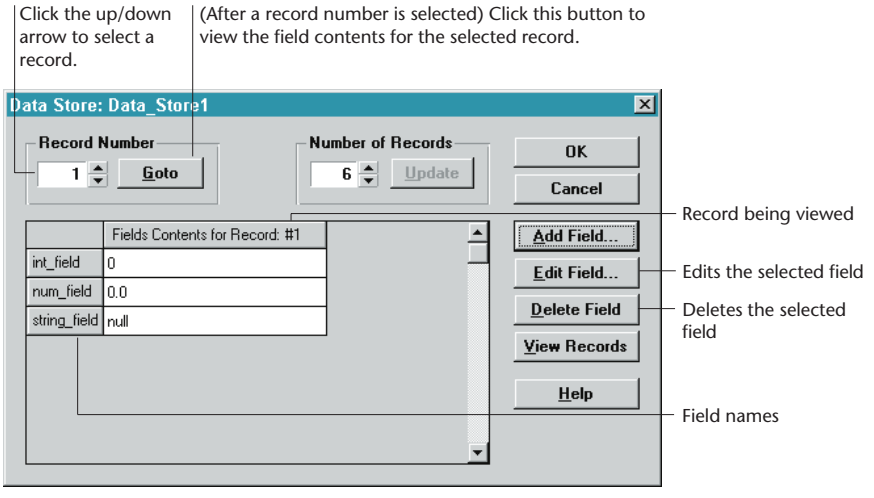
The fields view is your initial view of the data store when you open the Data Store dialog box. This view shows all the fields of one particular record.

Records View

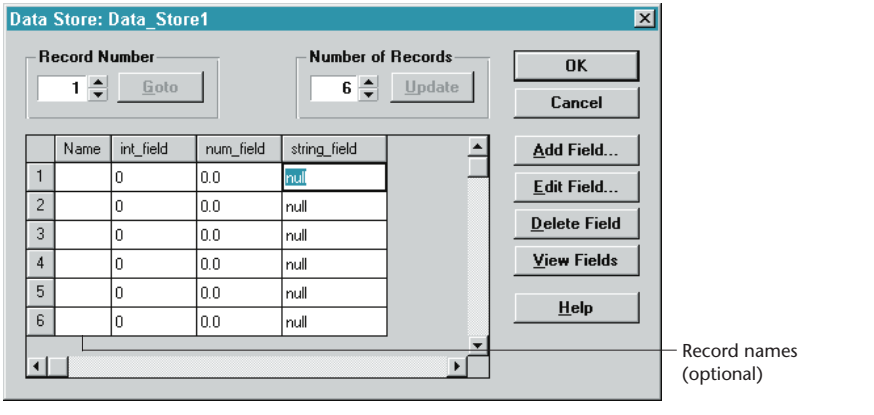
The records view shows all the records and their fields in table form.

Moving in a data store table

To move from cell to cell in a data store table using the keyboard, press Ctrl + the arrow keys.



A data store with three fields (in Fields view)



A data store with three fields (in Records view)

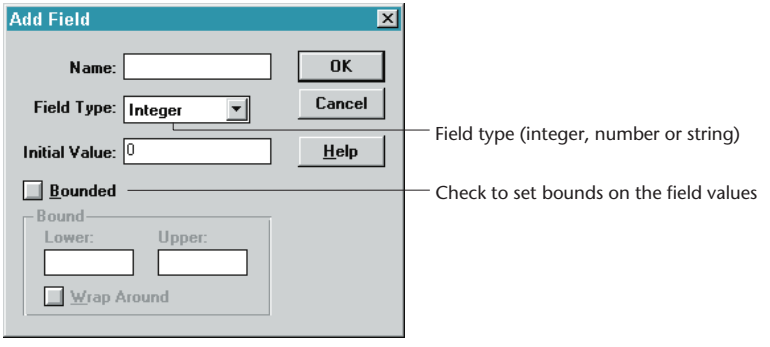
To view the field contents for a different record:

1. Click the up/down arrow of the Record Number stepper box or type a number directly. The Goto button is enabled.
2. Click Goto to view the field contents for the selected record.

Adding and editing fields

To add a field:

1. Click the Add Field button to open the Add Field dialog box:



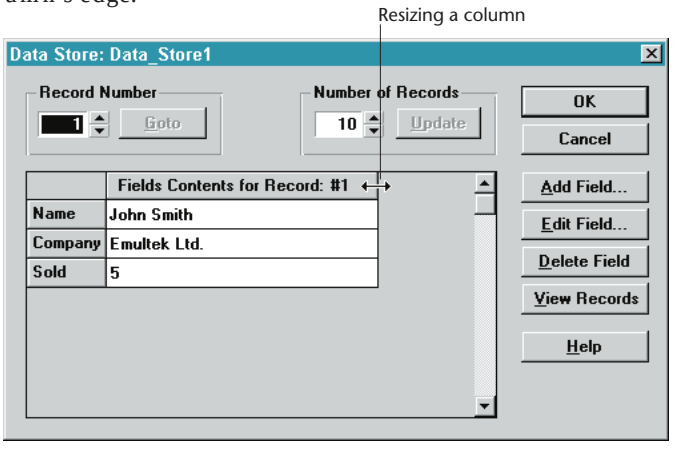
2. Type a name in the Name text box.
3. Select the field type (integer, number or string).
4. Either type a value in the Initial Value text box or accept the default value. (Default values—Integer: 0, Number: 0.0, String: empty.)
5. (For Number and Integer field types) If you would like to set bounds for the field values, check the Bounded check box and enter minimum and maximum values. (See the "Number" and "Integer" sections earlier in this chapter for more details about setting bounds.)
6. Click OK; the field name and default value are displayed in the Data Store dialog box.
7. Repeat steps 1–5 for each field you want to add.

To edit a field name and/or initial value:

1. Place the cursor in the Field Contents cell for the field you wish to edit.
2. Click the Edit Field button to open the Edit Field dialog box (identical to the Add Field dialog box).
3. Make the necessary changes in the Name and/or Initial Value text boxes. You cannot change the field type, but you can add or change bounded values.
4. Click OK; your changes appear in the Data Store dialog box.

Resizing the columns

You can resize the dialog box's columns (in either view) by dragging the column's edge:



Giving names to records

Each record of a data store is identified by its record number. You use this number in the Logic Editor when you build logic referring to a data store's record. As well as a number, you can give a record a name.

In the application's logic, a record with a name is referred to by its number and/or its name. You can only assign a name to a record in records view.

To assign a name to a record:

1. (Make sure that you are in records view.) Select a record name cell.
2. Type in a name.

Editing data store elements

You can edit the values of individual data store field elements in either field or records view.

To edit a data store element:

1. Select the cell.
2. Type the new value.



Point

The point object is a variable that holds a Rapid coordinate (x@y). You can use the point object to move dynamic graphic objects. It is especially useful when your application contains several modes and you have an object that moves to different positions in each of the modes. It is also useful when you want to place several objects in the same position.

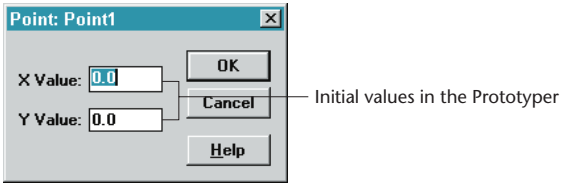
Example:

A dynamic circle (Circle1) is located at (10@10) and the value of Point1 is (20@20). If you build the entry activity, **Circle1.position := Point1**, the circle will appear at (20@20) each time the application enters that mode.

In addition, you can use the point object to resize dynamic, active primitive objects. You can assign its value to the:

- *size* property (for frame, imported bitmap and title text objects).
- *radius* property (for circle objects).
- *start* and *end* properties (for line objects).

The Point dialog box opens when you add or edit a point object. It is used to assign or edit a point object's initial values:



❖ *NOTE: The Point dialog box also opens when you define points in other circumstances (e.g., as a points property for an active primitive object, as a user property for a user-defined object).*



TIME OBJECTS

This class of nongraphic objects comprises objects that hold time data in a variable. You can display the data, use it in calculations, or base triggers on it.

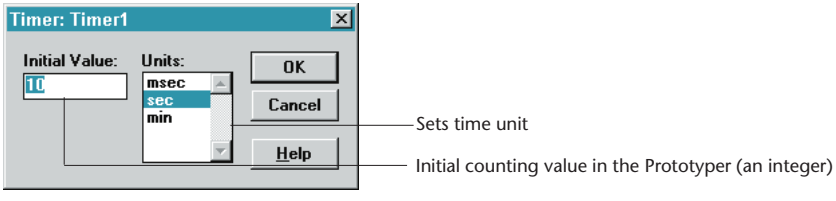


Timer

The timer object counts down from an initial value and stops when it reaches zero (generating the event, *tick*).

The timer can be stopped before it reaches zero, in which case it resumes counting from where it was interrupted when started again. The timer can also be reset, in which case it starts counting from the initial value when started again. The timer also has an "automatic restart" option, that is, when it reaches zero it generates a tick event then starts at the initial value once again.

Use the Timer dialog box to set the initial count value and the counting units:



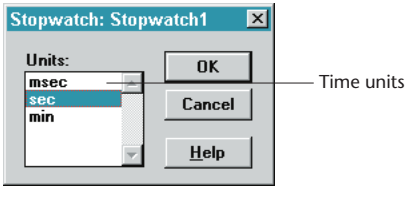


Stopwatch

The stopwatch object counts up from zero in the selected time units and generating an event, *tick*, as each time unit passes by. It can be stopped then restarted from either its current time value or the initial time value. The stopwatch's value (i.e., the elapsed time) can be read at any time.

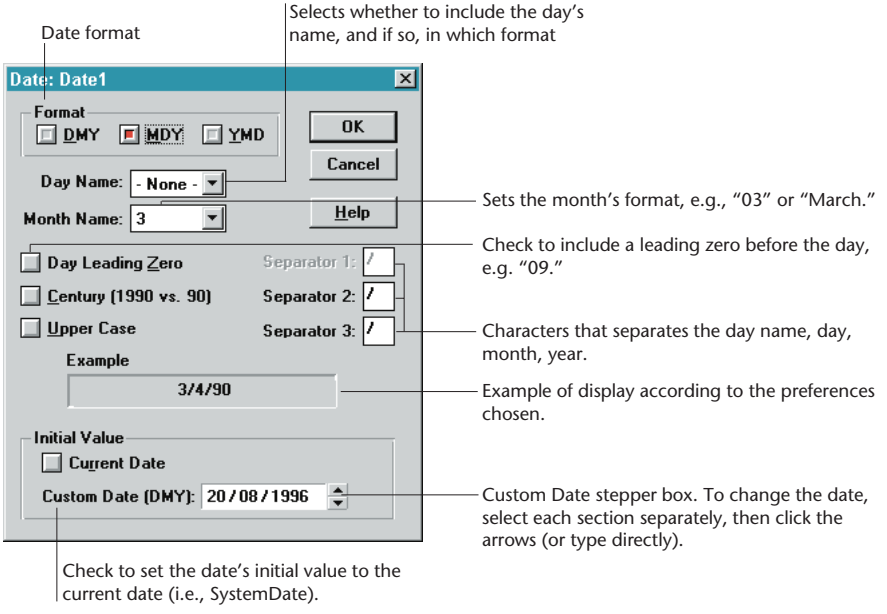
❖ *NOTE: If the stopwatch is counting in milliseconds, the tick is not generated.*

Use the Stopwatch dialog box to select the time units:

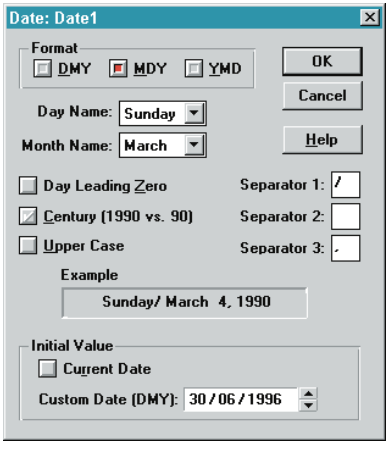


Date

The date object allows you to set any date and to display it in an application. You can set the date in a number of formats: day-month-year, month-day-year, with or without the day's name, etc.



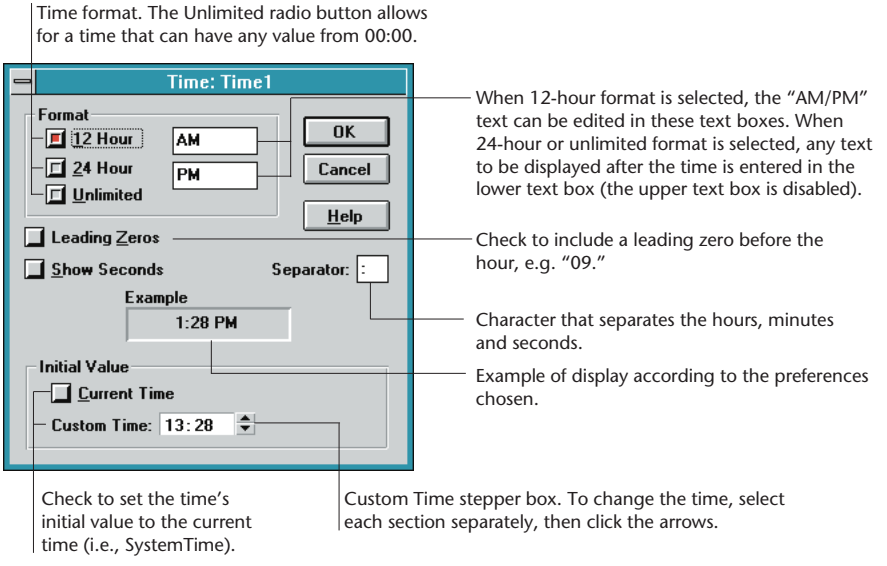
The dialog box below would display the date: Sunday/ June 30, 1996.



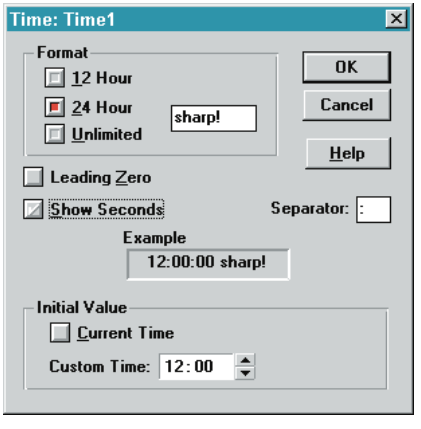


Time

The time object allows you to set any time and to display it in an application. You can set the time in a number of formats: 24-hour, 12-hour, with or without seconds, etc.



The dialog box below would display the initial time: 12:00:00 sharp!





SIGNAL OBJECTS

This class of nongraphic objects comprises objects that emit some kind of signal, either:

- To the user (such as an audible beep).
- To Rapid itself (to instruct it to carry out some operation).



Event

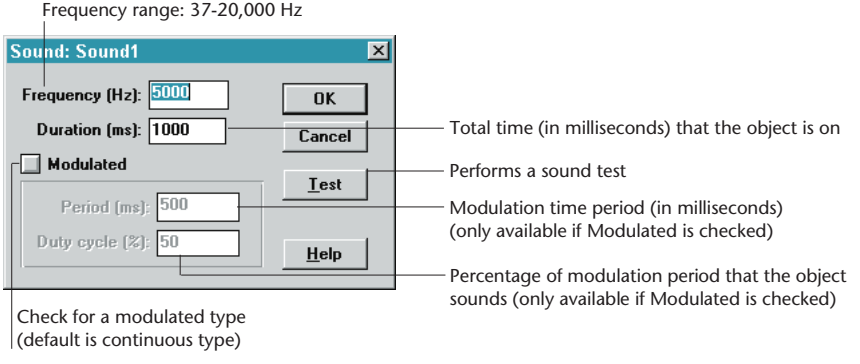
The event object is used to simplify Rapid logic when designing complex applications. For example, you can generate the event in one mode, and then use the event to trigger a transition in a concurrent mode. For more details about using the event object, see “Event Object” on p. 13–32.

The event object is the only nongraphic object that does not have an additional dialog box. When you add an event, the More button is disabled.



Sound

The sound object generates sound at a specified frequency, either continuously or intermittently. The frequency, duration and modulation option are set in the Sound dialog box:



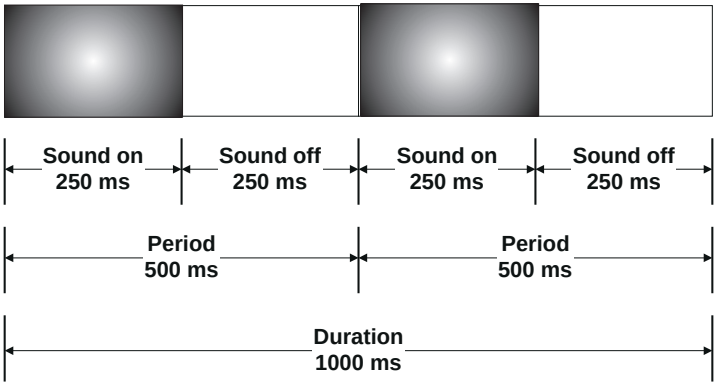
❖ *NOTE: The Sound dialog box shown above is not the default dialog box (although it contains the default values). When the dialog box is first opened, the Modulated check box is not checked and Period and Duty cycle are hidden.*

To make the sound object intermittent:

1. Click the Modulated check box; the Period and Duty Cycle parameters are enabled.
2. Type a number in the Period text box (or accept the default value). The number you enter cannot be greater than the Duration value.
3. Type a number in the Duty Cycle text box (or accept the default value).

Example:

If you use the default duration of 1000 ms, the default modulation period of 500 ms, and the default duty cycle of 50%, the object sounds for 250 ms, then is silent for 250 ms, then sounds for 250 ms, then is silent for 250 ms, after which time the object is inactive (see illustration below).



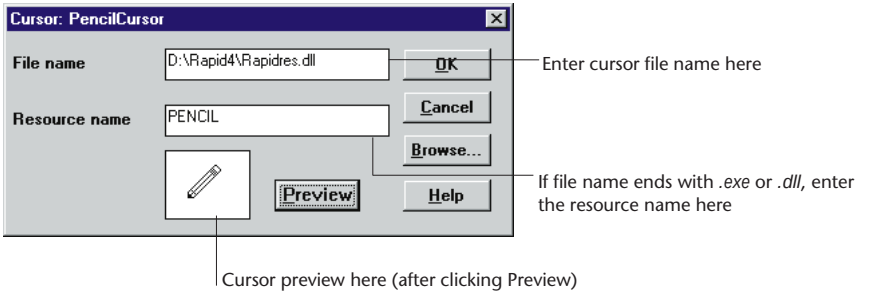


Cursor

The cursor object lets you add custom cursors to your application, to be used in the logic in conjunction with the mouse system object (see p. 5-29 for a brief description). In this way you can specify a unique cursor (other than the system cursors already supplied with Rapid) that will show when the mouse is located over a certain graphic object, switch position, group or UDO.

You define a cursor by specifying a *.cur*, *.ani*, *.exe* or *.dll* file. In the case of a resource file, you must specify the cursor's resource name as well.

The cursor object's More dialog box appears as follows:

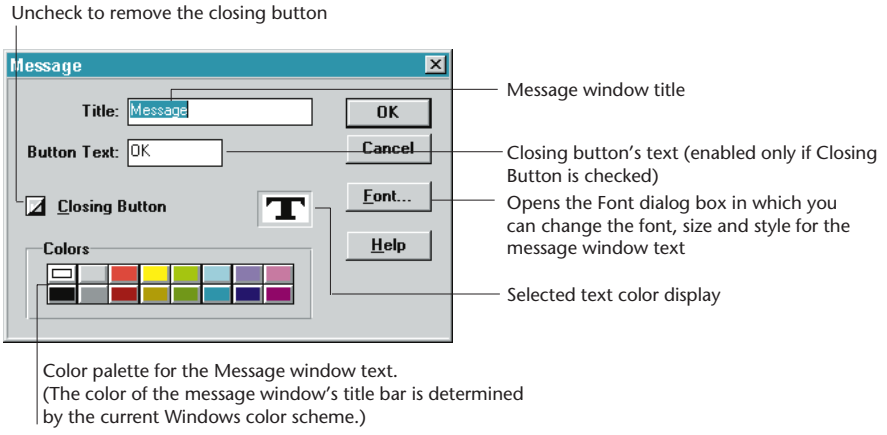




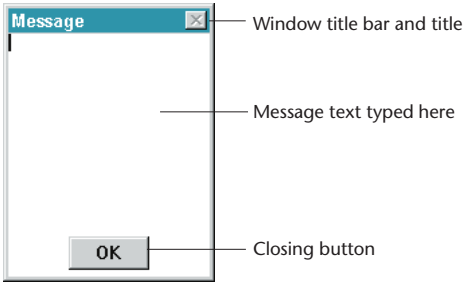
Message

The message object displays a message in a typical Windows pop-up pane. In the Logic Editor, you can construct actions and/or activities that open and close the message in the Prototyper.

When you add a message object to the layout, the Message dialog box opens, together with the message window (in the upper left corner of your screen):



In the Message window, you type the actual message text:



To set up a message window:

1. In the Message dialog box:

   Enter the title bar text (if any), the closing button text (or uncheck the Closing Button check box if you do not want a closing button), and select a color for the message window text (default: black).

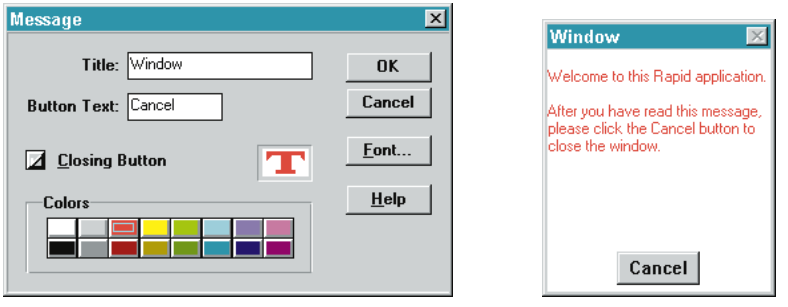
2. When you click inside the message window, the preferences you set in the dialog box appear. Make any additional changes in the Message dialog box, then click again in the Message window.

3. In the message window, enter the message text. The message window features text wrap around.

4. Move the message window to the location at which you want it to appear in the Prototyper. You can also resize the window, if necessary.

5. When you are satisfied with the size, location and content of the Message window, click OK in the Message dialog box. Both the dialog box and the message window close. The message window is displayed in the Prototyper when the *open* function is used.

Below is an example of a defined message object and its resulting window:



SYSTEM OBJECTS

Rapid automatically generates the following nongraphic objects in every application: ASCII, Keyboard, SystemDate and SystemTime. You can include them in your application's logic.

You cannot edit or delete the system objects.

ASCII

The ASCII object allows you to assign ASCII codes and string characters with specific ASCII values. For example, the following logic assigns the character whose ASCII value is 169 (the copyright symbol):

ASCII withValue: 169

Using the ASCII object in conjunction with the keyboard object (see below) allows you to test for when the user presses certain ASCII codes, for example, bs (backspace) and cr (carriage return).

Keyboard

When the Keyboard object is activated, it allows the computer keyboard to become an object that you can use during runtime. For example, you could use the keyboard to enter text into a text display or text window.

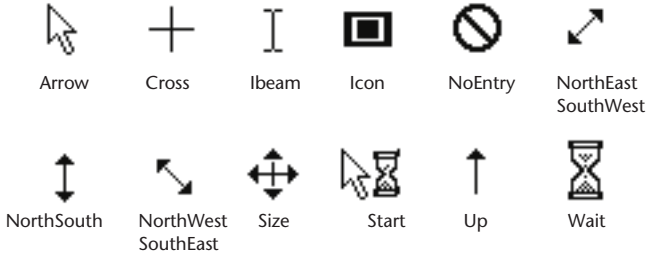
Mouse

The Mouse object provides two services:

- It makes the system mouse position and events available to the logic.
- It assigns cursors (either system—see below, or custom—see p. 5-26) to graphic objects and switch positions. The assigned cursor appears during run-time when the mouse is located over the object or position.

SystemCursors

Rapid provides twelve predefined cursors which appear in the Objects list of the Logic Palette, as a subset of the System group:



SystemDate

The SystemDate object contains the computer's real-time date. The format may be D/M/Y (default) or M/D/Y, as specified in the Windows Control Panel. The values of SystemDate can be assigned to the date object.

SystemTime

The SystemTime object contains the computer's real-time clock. Either a 12- or 24-hour (default) clock is used, as specified in the Windows Control Panel. The values of SystemTime can be assigned to the time object.

CHAPTER 6

Multimedia Objects

Rapid's multimedia objects allow you to use the Windows-compatible multimedia devices installed in your computer within your Rapid applications. You can, for example, use your MIDI sequencer or wave audio device to play music or a recorded sequence at appropriate points in your Rapid prototype. You can use your digital video or overlay device to project video clips in the device's viewing window—or onto a viewing area that you have defined in your application layout. In short, this class of nongraphics objects can help you bring your application to life.

This chapter covers how to define multimedia objects in the Object Layout, as well as how to apply their logic in the Logic Editor. For a detailed discussion on how to build logic in Rapid, see Chapter 12, "Building Logic" in this user manual.

In addition, you will find complete information on the multimedia object functions in the *Object Reference* manual.

In this chapter you will read about:

- MCI devices and Rapid's multimedia objects.
- Adding and defining multimedia objects in the Object Layout.
- Rapid syntax for multimedia objects.
- Opening and playing multimedia objects.
- Displaying an image in a destination window.
- Stopping, pausing and closing multimedia objects.

ABOUT MEDIA CONTROL INTERFACE (MCI)

Microsoft® Windows provides application developers with services for developing multimedia applications using the extended capabilities of a multimedia personal computer. In particular, Windows provides a Media Control Interface (MCI) for controlling media devices such as audio hardware, movie players and videodisc/videotape players.

The Media Control Interface has various device types. A device type is a set of MCI drivers that share a common command set and are used to control similar multimedia devices or data files. The following table lists the defined device types; each type is supported by Rapid as a separate multimedia object:

DEVICE TYPE	DESCRIPTION
Animation	Animation device
CD Audio	Audio CD player
DAT	Digital Audio Tape player
Digital Video	Digital Video in a window
Overlay	Overlay device (analog video in a window)
Scanner	Image Scanner
Sequencer	MIDI Sequencer
VCR	Videotape recorder or player
Videodisc	Videodisc player
Wave Audio	Audio device that plays digitized waveform files

THE MULTIMEDIA OBJECTS



The multimedia object buttons are:


Wave Audio


Digital video in a window


Animation device


Audio CD player


Sequencer


Videotape recorder (VCR)


Videodisc player



Digital audio tape player


Image scanner


Overlay device



Undefined MCI device

Compound vs. Simple Multimedia Objects

Some objects rely on data stored in a file for playback. The Wave audio object, for example, only plays data stored in *.wav* files. These objects are known as compound objects and their data file is known as a **device element**.

The following objects are compound multimedia objects:

- Animation device
- Digital video in a window

- Overlay device
- Scanner
- Sequencer
- Wave audio
- Other (undefined MCI device)

Simple objects do not require data files for playback as their data is stored on media particular to that device (e.g. compact disc, videotape, etc.).

The following objects are simple multimedia objects:

- CD Audio player
- Digital audio tape recorder (DAT)
- Videotape recorder (VCR)
- Videodisc player

ADDING AND DEFINING A MULTIMEDIA OBJECT

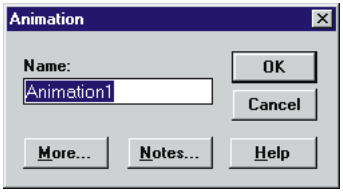
Since you use the same method to add all the multimedia objects to the Object Layout, the instructions for adding an object are presented for one object type only (the animation object).

After you have added an object to the layout, you can define it in its main dialog box. The instructions for defining a multimedia object apply to all the multimedia objects. The only difference is between compound and simple objects; compound objects require you to designate a device element and simple objects do not.

Adding a Multimedia Object

To add a Multimedia object:

1. Select the object you want to add.
2. The object's initial dialog box opens:



Click OK to add the object (**without** changing its default properties) or click the More button to open the object's main dialog box in which you define the object's properties.

To rename the object:

- Overtype the default name in the Name text box.

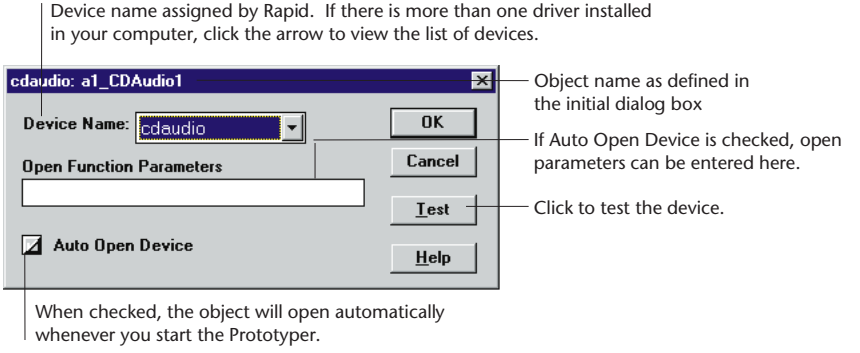
❖ *NOTE: You cannot change the object's name in its main dialog box. If you want to change its name after you have added it to the application, you can do so using the Nongraphic Objects dialog box (see "Managing Nongraphic Objects", p.5-3).*

Defining the Multimedia Object's Properties

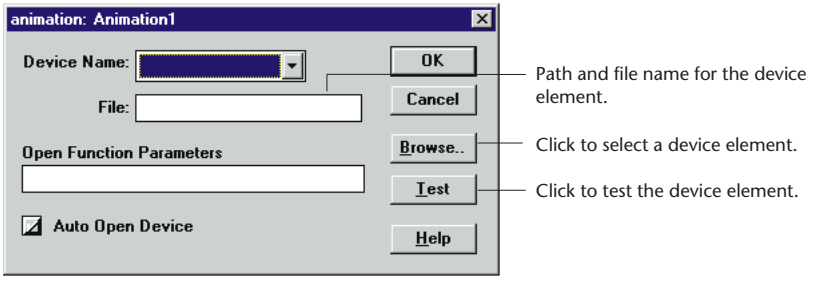
The general instructions for defining an object's properties are given below, together with an illustration of one simple object (the CD audio object) and one compound object (the animation object).

To define the object's properties:

1. After you click the More button, the object's main dialog box opens:



Dialog for a simple multimedia object



Dialog for a compound multimedia object

2. Define the object as necessary. The dialog box fields are explained below.

Device Name

Rapid assigns names to the devices that are supported by your computer. (The device names are defined in the [MCI] section of the Windows SYSTEM.INI file.). These names appear in the Device Names list box for the corresponding objects. Thus, for example, the animation device name(s) appear in the animation object's dialog box, the CD audio device name(s) appears in the CD audio object's dialog box, and so on. If no device has been installed in your computer, this box is empty.

File

❖ *NOTE: This field and the Browse button only appear for compound objects.*

In this field, you enter the path and file name of the device element you want the object to activate. For example if you want your wave audio object to play the Chimes wave audio file (as supplied with Windows) , then enter "c:\windows\chimes.wav".

Click the Browse button to select a device element from the dialog box.

Test Button

For a simple multimedia object, clicking the Test button opens and closes the device. In case of error, an error message is displayed. If the test succeeds, no error message is displayed.

For a compound multimedia object, clicking the Test button plays the device element from beginning to end. You can stop the test for objects which have windows (e.g., digital video) by closing the window.

Auto Open Device Check Box

Before a multimedia object will operate in the Prototyper, it must be opened. To do so, you either include an *open:* function in your application's logic (see "Opening an Object", p. 6-12) or check the Auto Open Device check box in the object's dialog box. By checking Auto Open Device, the object opens automatically whenever you start the Prototyper (and closes automatically when you stop the Prototyper).

❖ *NOTE: When you open a compound multimedia object, its data file is loaded into memory. Because of memory limitations, you might not be able to open many objects simultaneously. In this case, you should not define the objects to open automatically, but should open each object in the logic only when it is needed and close it afterwards.*

Open: Function Parameters

If you defined the object to open automatically (by checking Auto Open Device), you can include *open:* function parameters in the object's dialog box. When more than one parameter is entered, the parameters must be separated by spaces.

For example, to include a parameter that defines an animation object's destination window as a pop-up window, type: "style popup" (without the quotes). To include two parameters, type: "Style popup shareable".

Refer to the *Object Reference* manual for a full listing of each object's *open:* function parameters.

Managing Multimedia Objects

Once you have added a multimedia object to the layout, you can rename, edit, copy, delete or replace it with another multimedia object, using the Nongraphic Objects dialog box. For details about using this dialog box, see p. 5-3, “Managing Nongraphic Objects”.

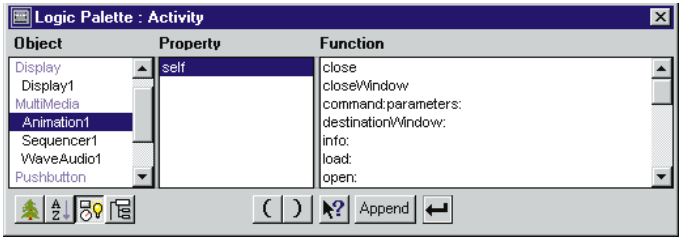
USING MULTIMEDIA OBJECTS

MCI provides standard commands for playing multimedia devices and recording multimedia resource files. In the Logic Editor, the MCI commands are converted into the objects’ functions.

To control a multimedia object, you must open the device and then send functions to it. The device closes automatically when the Prototyper closes. You can also use the *close:* function to close a device (especially useful if your computer has memory limitations).

Rapid Syntax for Multimedia Objects

In the logic palette’s object list, the multimedia objects are listed within their own group:



All the multimedia objects have only one property: *self*.

Syntax for functions that do not require parameters

The syntax for functions that do not require parameters is:

<object name> <function>

Examples of functions that do not require parameters or arguments are: *close*, *pause* and *stop*.

Syntax for functions that require parameters

All functions that end with a colon (:) require parameters and/or arguments. The syntax for these functions is:

<object name> <function:> *'<parameter> <argument> [notify] [wait]'*

WHERE:	REPRESENTS:
<parameter>	Provides more detail about the function (e.g., *from* and *to* in the example below).
<argument>	The variable(s) needed to complete the function (e.g., the numbers *0* and *100* in the example below).
[notify]	An optional parameter that directs the object to generate an event in one of four situations: - the device completes the action described by the function, - an error takes place, - a new function prevents a previous function from being satisfied, - another notification request supersedes the pending notification parameter
[wait]	An optional parameter that directs the device to wait until the requested action is completed before returning control to the application.

See pp. 6-20 and 6-21 for more details on the *notify* and *wait* parameters, respectively.

Example:

CDAudioObject play: 'from 0 to 100 notify wait'

When using functions that require a parameter

Some functions require parameters and/or arguments, and others do not. Functions which require a parameter appear in the Function list followed by a colon (e.g., *load:*). When the function is appended to the edit box, Rapid prompts you to add a string (i.e., the parameter and/or arguments).



When you add the parameter, you must enclose it (and the argument, if any) within apostrophes just as with any Rapid string. For example, if you want to load an animation file, the logic statement will be:

Animation1 load: 'engine.flc'

A logic statement can contain more than one parameter and/or argument. The following example contains three parameters and two arguments:

Animation1 play: 'from 10 to 1000 fast'

If a function requires a parameter, you must always include the apostrophes even if you do not include the parameter. For example:

CDAudio play:''

This action/activity causes the device to play from beginning to end.

Function Summary

The following table summarizes the functions that are available to the multimedia objects. (Not every function is applicable to each object.) Full details of each object's functions and their parameters are given in the *Object Reference* manual.

FUNCTION	DESCRIPTION	OBJECTS
close	Closes the device and unloads the device element (if any) from memory.	All
closeWindow	Closes the destination window in which the image is displayed.	Animation, Digital Video, Overlay
cue:	Prepares the device for playing or recording.	Wave Audio
delete:	Deletes a data segment from the *.wav* file.	Wave Audio

FUNCTION	DESCRIPTION	OBJECTS
destination-Window:	Defines a frame object as the destination window in which the image is displayed.	Animation, Digital Video, Overlay
escape:	Sends device-specific information to the device.	Video Disc
freeze:	Disables video acquisition to the frame buffer.	Overlay
info:	Returns textual information from the device: product name, model, etc.	All
is ready	A condition that returns True if the device is ready.	All
load:	Loads a device element from disk.	All except Audio CD, DATA, sequencer, Video Disc
open:	Initializes the device and loads the device element (if any) to memory.	All
pause	Pauses playing or recording.	All
play:	Starts playing the device.	All
put:	Defines the area of the source, destination and frame windows.	Animation, Digital Video, Overlay
realizePalette:	Tells the device to select and realize its palette into the display context of the displayed window.	Animation, Digital Video
record:	Starts recording input data. All data that is recorded after a file is opened is discarded if it is not saved before the file is closed.	All except Audio CD, Video Disc
resume	Resumes playing or recording on a paused device.	All
save:	Saves data to a disk file.	All except Audio CD, VCR, Video Disc
seek:	Finds the specified positon and stops.	All
set:	Sets the operating state of the device.	All
spin:	Starts or stops the disc spinning.	Video Disc
status:	Returns status information for the device.	All

FUNCTION	DESCRIPTION	OBJECTS
step:	Steps the play one or more frames forward or reverse. The default action is to step forward one frame.	Animation, Digital Video, Video Disc
stop	Stops playing or recording.	All
unfreeze:	Enables the frame buffer to acquire video data.	Overlay
update:	Repaints the current frame into the display context.	Animation, Digital Video
where:	Returns a string giving the x and y coordinates (in pixels) of the source, destination or frame area.	Animation, Digital Video, Overlay
whereScaled:	Returns a string giving the x and y coordinates (in the application's units) of the source, destinationor frame area.	Animation, Digital Video, Overlay
window:	Controls the display window.	Animation, Digital Video, Overlay
notifyAborted notifyFailure notifySuccess notify-Superseded	Generate an event when the device completes the action described by the function. See " Using the Notify Parameter" on p. 6-20.	All (as events for use in triggers)

Caveat

For a Rapid function and/or parameter to work as expected, it must be supported by the object's driver. If you use an unsupported function/parameter in your application's logic, a runtime error message appears when the application is run in the Prototyper.

Opening an Object

Before you can use an object in the Prototyper, you must open it. The *open:* function loads the driver into memory (if it is not already loaded.) For compound multimedia objects, the device element is also loaded into memory. The number of devices that you can open at any one time is limited only by the amount of available memory.

An action/activity that opens an object looks like:

<MMObject> open: ''

where <MMObject> is the name of a multimedia object.

Remember that you must add the apostrophes, even if you do not add an opening parameter.

❖ *NOTE: If you checked the Auto Open Device check box in the object's main dialog box, you do not need to include the open action/activity in the application's logic (see Auto Open Device Check Box on p. 6-76-PAGE 7).*

Including opening parameters

There are various parameters associated with the *open:* function that control aspects of the object's presentation. Among these is the style of window in which the image is presented—the destination window (for the animation, digital video and overlay objects only).

An action/activity that opens an object and performs an additional operation looks like:

<MMObject> open '<opening parameter>'

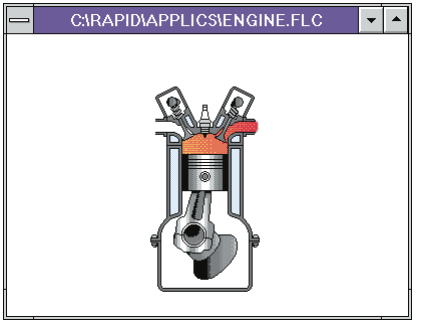
For details about each object's *open:* parameters, see the *Object Reference* manual.

❖ *NOTE: If you defined the object in the Object Layout so that it will open automatically, you can enter opening parameters in the Open Function Parameters text box—without the apostrophes (see Auto Open Device Check Box on p. 6-76-PAGE 7)*

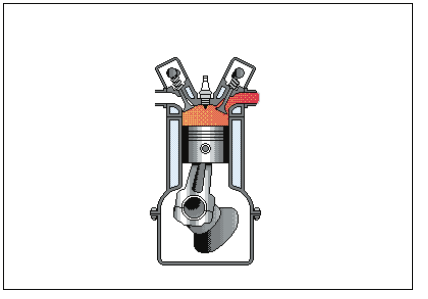
Defining the window style

The image of animation, digital video and video overlay objects is displayed in a default window that the device itself creates. The default style is a standard window which can be moved, resized, etc., while the image is playing in the Prototyper.

You can use the *open:* function's *style popup* parameter to define a pop-up style display window. The pop-up style displays the image only and does **not** allow you to adjust the window in any way while it is playing in the Prototyper. The illustrations below show these two types of window style:



Default display window style



Pop-up display window style

To define a pop-up window style:

- If the object does not open automatically:

  Build the following action or activity:

  <MMobject> open: 'style popup'

- If the object opens automatically:

  In the object's main dialog box, type "style popup" in the Open Function Parameters text box (see p. 6-76-PAGE 7).

  Both window styles are external to Rapid. If you click any of Rapid's windows (e.g., on a switch in the Prototyper), that window becomes the active window and may overlap or even completely obscure the multimedia window. If this happens, you can use any of the standard Windows techniques to bring the image to the foreground, such as pressing Alt+Tab. To avoid this problem, rearrange the windows on your screen (including the display window, if possible).

  ❖ *NOTE: You can also display the image within a Rapid frame object (see "Displaying an image in a destination window" on p. 6-156-15).*

Playing an Object

The *play:* function starts playing the object. If no parameters are added, play starts from the current position and plays until either:

- the object is paused/stopped, or
- the end of the media (for simple objects) or device element (for compound objects) is reached.

An action/activity that plays a device from the position where it was stopped to the end looks like:

```
<MMObject> play: ''
```

Remember that you must add the two apostrophes, even if you do not add a parameter.

Designating a play range

The *from* and *to* parameters indicate the positions at which the object should start and stop playing. For example, the following line plays an animation object from the fifth frame to the twentieth frame:

```
AnimObject play: 'from 5 to 20'
```

Displaying an Image in a Destination Window

You can display a multimedia image as part of your application's layout by placing it within the boundaries of a frame object. In this case, the destination window does not behave as a regular display window—the image is simply "projected" onto your layout. You can manipulate any other object (switch, pushbutton, etc.) without disturbing your view of the image.

General procedure

To display an image within a frame object, you need to:

- add a frame object to your layout and name it, and

To display an image in a destination window:

1. In the Object Layout, add a frame object (it does not matter which type since Rapid uses the object's enclosing rectangle to determine the frame's size).
2. Name the frame object in the parameter pane.

3. In the Logic Editor, add the following action or activity immediately after the object's *open* function:

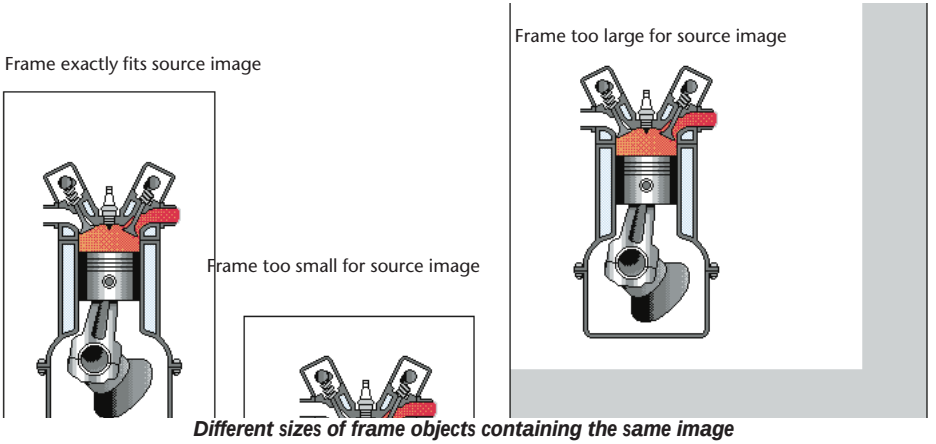
   <MMObject> destinationWindow: <FrameObject>

❖ *NOTE: <FrameObject> (unlike other arguments) is* **not** *enclosed in apostrophes.*

Any *play:* function after this action/activity will cause the image to be displayed within the frame that you defined in the Object Layout.

Defining a destination window as the exact size of the image

Some animation, digital video and overlay devices automatically resize their images to fit the frame (using the *window:* function, *stretch* parameter); others do not. For devices without stretch capabilities, the displayed image size is constant. As a result, the image only fits into the frame object if you exactly define the frame's size to match the image source's size. Below are illustrations showing the results of displaying the same animation image in differently sized frame objects:



Different sizes of frame objects containing the same image

To define the frame object to be the exact size of the image, you must first retrieve the image source's size and then make the frame object the same size.

Retrieving the image source's size

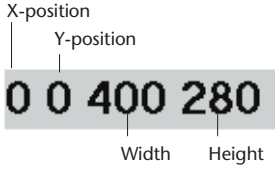
By constructing a mini-application using a text display, you can retrieve the precise size of the image source.

You use the *where:* and *whereScaled:* functions to return the position and size of the source or destination area for a particular object. The information is returned as a string in the form: *x y w h*, where *x* and *y* specify the coordinates of the top-left corner of the frame's enclosing rectangle, and *w* and *h* specify its width and height. *where:* returns these figures in pixels; *whereScaled* returns these figures in the application's units (centimeters, inches, etc.).

To display the precise size:

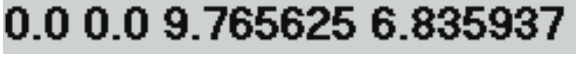
1. In the Object Layout, add a text display object.
2. In the Logic Editor, add the entry activity:

   <TextDisplay>.contents := <MMObject> where: 'source'
3. Start the Prototyper; the position and size of your multimedia object is displayed in pixels. Your text display should look similar to:



Source position and size, displayed in pixels (using the *where:* function)

If your application's scale units are not pixels, replace the *where:* function in the above example with *whereScaled*.



Source position and size, displayed in centimeters (using the *whereScaled*: function)

4. Once you know the width and height of the source, write it down. If you are using units other than pixels, round the figures to three decimal places. You can, if you wish, delete the text display from the layout; it has served its purpose.

Adjusting the frame object's size

To adjust the frame object's size:

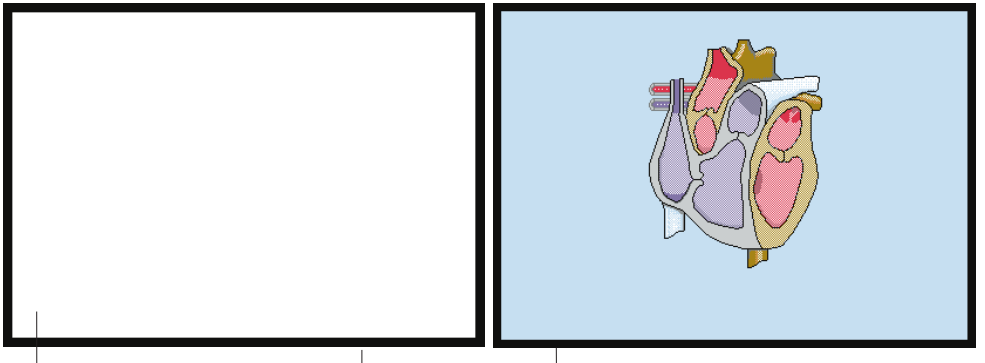
1. In the frame object's parameter pane, enter the image source's width (*w*) and height (*h*) in the x and y Size text boxes, respectively.
2. Click Accept.
3. When you start the Prototyper, the image will fit exactly into the frame object.

Presenting the image within a border

If you define a frame object as the image's destination window, the image will take up the entire area of the frame, including the frame's outline, and so will be displayed without a border.

To present the image within a border:

1. In the Object Layout, add another frame object that is larger than the image's frame.
2. Choose Layout|Move to Back to place this frame behind the image frame. The illustrations below show the frames in the Object Layout and the result in the Prototyper:



Time Formats

The units assigned to the position values vary from device to device, depending on the time format used by each object. The objects have a default time format and some have several different time format options. If you want to specify a time format other than the default, use the *set:* function.

Examples:

- The default time format for animation objects measures the position in frames. Another option allows you to set the time format to measure in milliseconds.
- The default time format for CDAudio objects measures the position in minutes, seconds and frames. Another option allows you to set the time format to measure in tracks, minutes, seconds and frames.

Play Parameters

Some objects have the capability to play at different speeds. For example, the digital video object's *play:* function has the following parameters in addition to *from* and *to*:

PARAMETER	PURPOSE
fast	Plays the sequence at a fast rate.
reverse	Specifies that the play direction is backwards.
scan	Plays the sequence as fast as possible without disabling the video.
slow	Plays the sequence at a slow rate.
speed	Plays the sequence at a specified speed, measured in frames per second.

The following logic statement plays a digital video object from the fifth frame to the fortieth frame at the speed of eight frames per second:

```
DigVideo1 play: 'from 5 to 40 speed 8'
```

Getting Information about an Object

Every object has the *status:* and *info:* functions. These functions return information about the object which can be displayed in your application by adding a text display object and using the *contents* property and := function.

Examples:

The following action/activity displays the number of frames of a digital video object's device element:

TextDisplay1.contents := DigVideo1 status: 'length'

The following condition tests whether the length (in frames) of the digital video object is equal to 20:

& DigVideo1 status: 'length' = 20

❖ *NOTE: You cannot request more than one item of information in a single function. The MCI can only return one string or value in response to an information request.*

Using the 'Notify' Parameter to Generate Events

Adding *notify* as a parameter directs the object to generate an event when the device completes an action. A notification event can indicate one of the following results:

EVENT	DESCRIPTION
notifyAborted	Generated when you send a new function that prevents a previous function from being satisfied. For example, sending the *stop* function cancels a notification pending for the *play: 'to 500 notify'* function. If your function interrupts a function that has a notification pending, and your function also requests notification, MCI aborts the first notification immediately and responds to the second notification.
notifyFailure	Generated if a device error occurs while a device is executing the MCI command. For example, a notification fails when a hardware error occurs while the device is playing.

EVENT	DESCRIPTION
notifySuccess	Generated when the action is completed without interruption.
notifySuperseded	Generated when the device has a notification pending and you send it another notify request. When a notification is superseded, MCI resets the driver to correspond to the notify request of the new function.

A notification may be used by Rapid as a trigger in the Logic Editor in the form of "if the notification is successful, perform an action".

Example:

You can add a pop-up Message object that contains a message and activate it when the *notifySuccess* event occurs. The following logic achieves this result:

LOGIC TYPE	SYNTAX
Activity/Action	**Animation1 play: 'from 1 notify'**
Internal trigger	**Animation1 notifySuccess &**
Action	**Message1 open**

Using the Wait Parameter

Normally, control over the application returns to the user immediately after a command is initiated, even if it takes several minutes to complete the action. For example, after a VCR device receives a rewind function, control returns when the tape starts to rewind; it does not wait for the tape to finish rewinding.

The *wait* parameter directs the device to wait until the requested action is completed before returning control to the application. For example, the following logic will not return control to the application until the playback is complete:

Animation1 play: 'from 1 to 50 wait'

Stopping and Pausing Objects

The *stop* function suspends the playing or recording of an object.

The *pause* function suspends operation, but leaves the device ready to resume playing or recording immediately with the use of the *resume* function.

What is the difference between *stop* followed by *play:* and *pause* followed by *resume*?

If an object's *play:* function includes a *from* parameter, then when you play, stop and replay the object, it returns to the *from* parameter position. If you play, pause and resume, the object plays from the current position.

Example:

Your application contains the following activity:
DigitalVideo1 play: 'from 5 to 40'

While the device is playing in your application, you send the *stop* function to the object. Let's say that the sequence stops at frame 25. If you restart the video with the same *play:* function, it will reset the *from* position to 5.

If you pause the device and then resume, the position values are not changed. That is, the sequence will continue playing from frame 25.

If, however, an object's *play*: function does not include a *from* parameter (e.g., **DigitalVideo1 play: ''**), the *stop* and *pause* functions have the same effect, i.e., playing the device after stopping it continues from the current position just as resuming play after pausing.

Closing Objects

The *close* function closes the ojbect and unloads its device element from memory (for compound objects). Since a device closes automatically when you stop or close the Prototyper, you do not have to include a close statment in your application's logic. However, if your PC has memory limitations, you might choose to include close statements.

When an object is used in more than one mode, you can either include a close statement in each of the modes, or close the object once in a concurrent mode. For example, if a wave audio object plays a *.wav* file at different times in your application, you can always turn it off by using one activity or action in a concurrent mode which is always active, e.g., a child of the root mode.

CHAPTER 7

Communication Objects

Rapid has several objects that allow your application to communicate with other Windows programs or DLLs. Most of these objects take advantage of the dynamic data exchange (DDE) mechanism, giving Rapid the ability to:

- Send and receive data to and from other programs.
- Send and receive commands to and from other programs.
- Send and receive data and/or commands to and from other programs on an external device, via RS232 communications.

These objects are useful when you want to perform calculations on a Rapid application's data in an external program such as a spreadsheet. You can also use them to control the behavior of another Rapid application, allowing you to use Rapid as a CBT authoring tool.

In this chapter you will read about:

- DDE basics—what is DDE and what is it used for?
- DDE Client and DDE Server objects—how to enable Rapid to exchange data with other programs. Included in this section are a few short examples that demonstrate how to transfer data between two Rapid applications and between Rapid and a different Windows program.
- DLL objects—how to integrate complex functions into your applications.
- Commlink object—how to communicate with external devices via one of the computer's serial ports.
- Simulation manager—a special object that uses DDE to control another Rapid application.

DDE BASICS

DDE (Dynamic Data Exchange) is a protocol that enables two Windows applications to exchange data continuously and automatically. For example, you might want to build a Rapid application that uses data supplied by a spreadsheet program; as the spreadsheet data changes, Rapid responds accordingly. The channel of communication between the two applications is known as a DDE channel.

DDE Server and Client

Two applications exchanging data through a DDE channel are said to be involved in a DDE conversation, or link. Rapid can conduct more than one conversation with the same application at the same time, but each conversation occurs on a different channel.

The application that initiates a DDE link and receives the data is known as the DDE client. The application that responds to the DDE client and provides ("serves") the data is known as the DDE server. Rapid can act as a client with one application and at the same time be a server to another.

Applications, Topics and Items

The data that is to be transferred from the server to the client has a unique "address" that is defined by three parameters:

- Application
- Topic
- Item

When you want a client to initiate a DDE link, you must provide it with these parameters.

Application

Every Windows application that supports DDE has a unique application name. This name is usually the name of the *.exe* file for that application, without the *.exe* extension. Rapid's application name is "Rapid". Examples of other application names are "123w" (Lotus® 1-2-3®) and "WinWord" (Microsoft® Word for Windows™).

Topic

The topic is the "subject" of the DDE conversation, i.e., the file that is open in the application. An example of a Rapid topic is "CAMERA.RPD". Examples of other topics are "LETTER.DOC" and "SHEET1.XLS".

Item

The item refers to the specific piece of data within the topic that is exchanged between the server and the client. An item may be a cell or a range of cells in a spreadsheet, a bookmark in a word processor document, etc. Each application has its own way of representing items.

Any item that you assign to a Rapid DDE object becomes a property of that object. Rapid items can be of the number, integer or string data type.

❖ *NOTE: When communicating between two Rapids or Reviewers, you can use any application, topic and item names.*



DDE OBJECTS

The communication class of objects contains two nongraphic DDE objects:

- DDE Client
- DDE Server

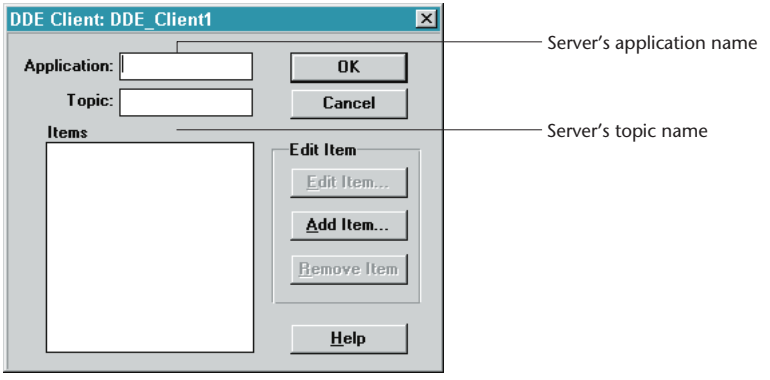
Following this section are some examples that demonstrate how to work with DDE objects.



DDE Client

The DDE client object initiates a DDE conversation with a server application, exchanging numeric or string data. This object also allows you to send commands to the server application.

When you add this object, the following dialog box opens:



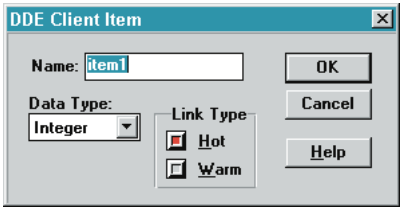
The application and topic names must be the names that identify the server application (see "Applications, Topics and Items," above). The DDE client object can exchange data with more than one item in the server application, but all the items **must** have the same application and topic address.

For example, one DDE client object can exchange data with two different cells in an Excel spreadsheet; the application name would be "Excel", the topic name would be the spreadsheet name (e.g., "sheet1.xls") and the object would have two items, one for each cell.

Defining the DDE client's items

To define items for the DDE client:

- Click the Add Item button to open the DDE Client Item dialog box:



Item name

The item name must be in a form that is acceptable to the server application. For example, an acceptable item name for a spreadsheet cell is "R3C2" (i.e., row 3, column 2). Each application has its own method for naming its data items; if you are not familiar with your server application's terminology, refer to its user documentation.

Data type

You must define the type of data that is contained within the item: integer, number or string.

Link type

An item can be linked to the server in one of two ways, "Hot" or "Warm."

- Hot—when the data within the server's item changes and the server sends the new value to the client, the client's value is automatically updated.
- Warm—when the data within the server's item changes and the server sends the new value to the client, the server notifies Rapid that the data has been sent. Each time the data changes, a *dataChanged* event is generated. The DDE client object is only updated if it sends an explicit request (*requestData*).

DDE client logic

Each of the items that you defined for the DDE client is a **property** of the object. The functions of the property are either integer, number or string functions—depending on the item's data type. In addition, each item property has the following functions:

FUNCTION	DESCRIPTION
requestData	Requests data from the server. This function is only applicable if the link type is **warm**.
sendCommand:	Sends a command, in the form of a string, to the server. The format of the command depends on, and should be supported by, the server application.
is open	A condition that tests whether the DDE client's link to the server is open.
dataReady	An event, generated when the server sends data to the DDE client. This event is only applicable if the link type is **warm**.

Sending data to the server

The DDE client's := function sends data to the server, changing the server item's value. For example, the following action sends the value 7.5 to an item named R1C2 in the server that is linked to DDEClient1:

```
DDEClient1.R1C2 := 7.5
```

Receiving data from the server

If the server returns data to the DDE client, the DDE client's response depends on its link type:

- If the link is hot, the DDE client's value automatically changes.
- If the link is warm, the DDE client receives the *dataReady* event, and you can choose whether or not to update the DDE client's value.

  If you want to update the value, use the *dataReady* event to trigger the *requestData* action:

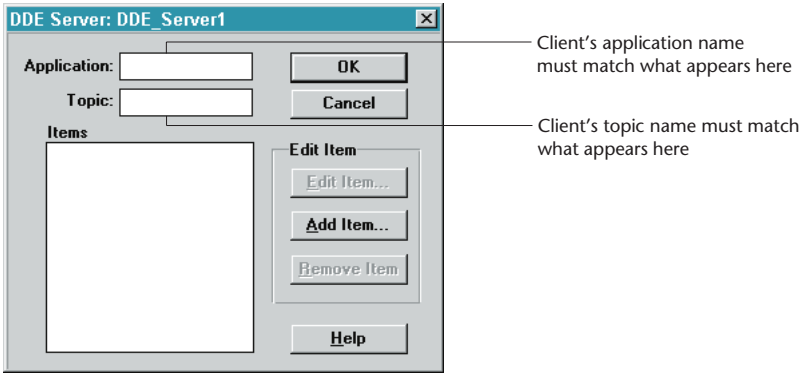
  Trigger: **DDEClient.item1 dataReady &**
  Action: **DDEClient.item1 requestData**



DDE Server

The DDE server object allows your Rapid application to send numeric or string data to a client application.

When you add this object, the following dialog box appears:



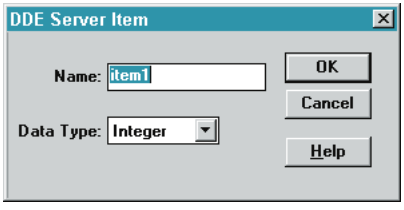
You can choose any application and topic names that you want, in which case you must ensure that the client is set up according to these names. If you cannot alter the client's application or topic names, you must use those names in the DDE server's dialog box.

The DDE server object can provide data to more than one item in the client application. For example, one DDE server object can exchange data with two different cells in an Excel spreadsheet; the application name would be "Excel", the topic name would be the spreadsheet name (e.g., "sheet1.xls") and the object would have two items, one for each cell.

Defining the DDE server's items

To define items for the DDE server:

- Click the Add Item button to open the DDE server Item dialog box:



Item name

You can enter any name as a server item name. This can be a name such as "Item1," "Item2," etc., or something more descriptive such as "PotentiometerValue." When you set up the DDE link from the external client program to Rapid, you must ensure that the DDE Server object's item name is identical to the client's item name.

Data type

You must define the type of data that is contained within the item: integer, number or string.

DDE server logic

Each of the items that you defined for the DDE server is a **property** of the object. The functions of the property are either integer, number or string functions—depending on the item's data type. In addition, the DDE server's item properties have the following condition:

FUNCTION	DESCRIPTION
is open	A condition that tests whether the DDE server's link to the client is open.

Sending data to the client

The DDE server's := function sends data to the client, changing the client item's value. For example, the following action sends the value 7.5 to an item named R1C2 in the client that is linked to DDEServer1:

```
DDEServer1.R1C2 := 7.5
```

EXAMPLES USING DDE OBJECTS

This section presents three examples that demonstrate how to use Rapid's DDE objects for:

- Passing data from one Rapid application to another.
- Passing data from a Rapid application to an Excel spreadsheet.
- Passing data from Rapid to Excel, performing a calculation on the data and passing back the result.

❖ *NOTE: Even if you do not have Excel installed on your computer, reading these examples will give you a good idea of how to link Rapid to other programs through DDE links.*

Example 1: Passing Data Between Rapid Applications

In this example, changing the value of a potentiometer in one application (the client) increases the reading of a dial in another application (the server).

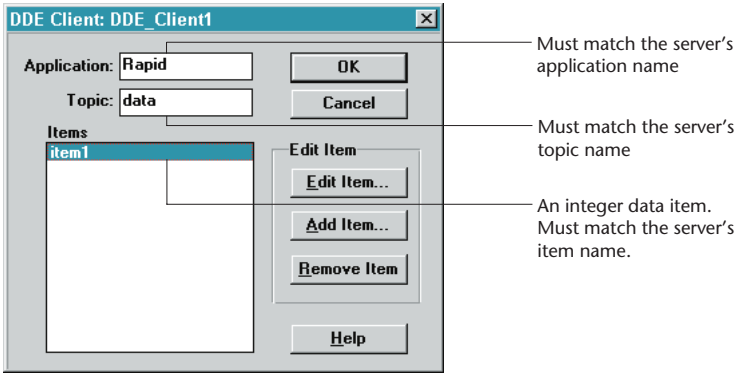
Adding the client application's objects

The client application needs two objects:

- A potentiometer
- A DDE client

To add the client application's objects:

1. Start a new application.
2. Add a potentiometer, accepting its default values settings. (You can use any potentiometer, though this example shows a rotary potentiometer.)
3. Add a DDE client object, and define its settings according to the dialog box shown below:



Building the client application's logic

In the Logic Editor, build the following mode activity:

DDE_Client1.item1 := Potentiometer1.value

This activity assigns the value of the potentiometer to the DDE client's item.

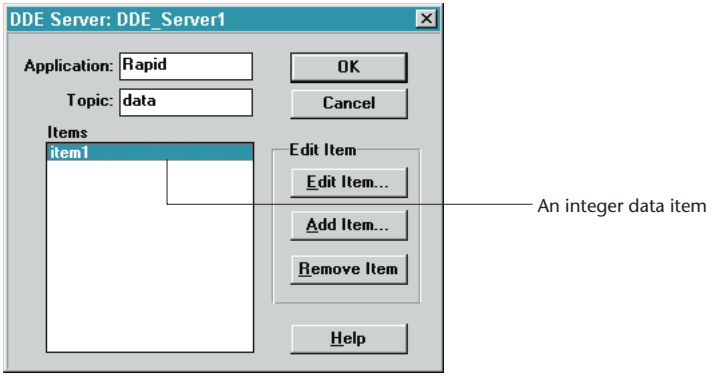
Adding the server application's objects

The server application needs two objects:

- A dial
- A DDE server

To add the server application's objects:

1. Save the client application, then start a new application.
2. Add a dial, accepting its default values settings.
3. Add a DDE server object, and define its settings according to the dialog box shown below:



Building the server application's logic

In the Logic Editor, build the following mode activity:

Indicator1.reading := DDE_Server1.item1

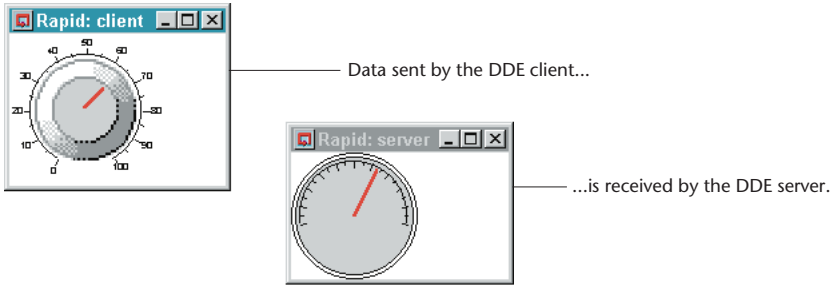
This activity assigns the value of the DDE server's item to the indicator's reading.

Running the two applications

To run the two applications side-by-side, you can either:

- Run them both in different copies of the Reviewer, or:
- Run one in the Reviewer and one in Rapid.

The illustration below shows both applications running in Reviewers:



Example 2: Passing Data From Rapid to Excel

In this example, you will learn how a Rapid application can pass integer data to an Excel spreadsheet using a DDE client object.

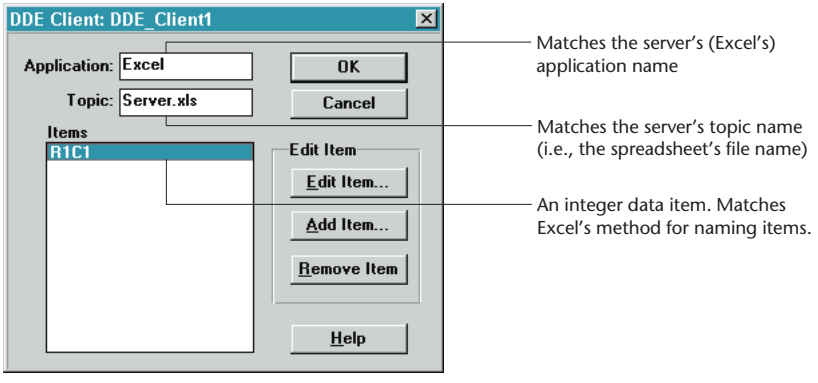
Adding the Rapid application's objects

The Rapid application needs two objects:

- A potentiometer
- A DDE client

To add the client application's objects:

1. Start a new application.
2. Add a potentiometer, accepting its default values settings.
3. Add a DDE client object, and define its settings according to the dialog box shown below:



Building the application's logic

In the Logic Editor, build the following mode activity:

DDE_Client1.R1C1 := Potentiometer1.value

This activity assigns the value of the potentiometer to the DDE client's item.

Creating the Excel spreadsheet

To create the spreadsheet:

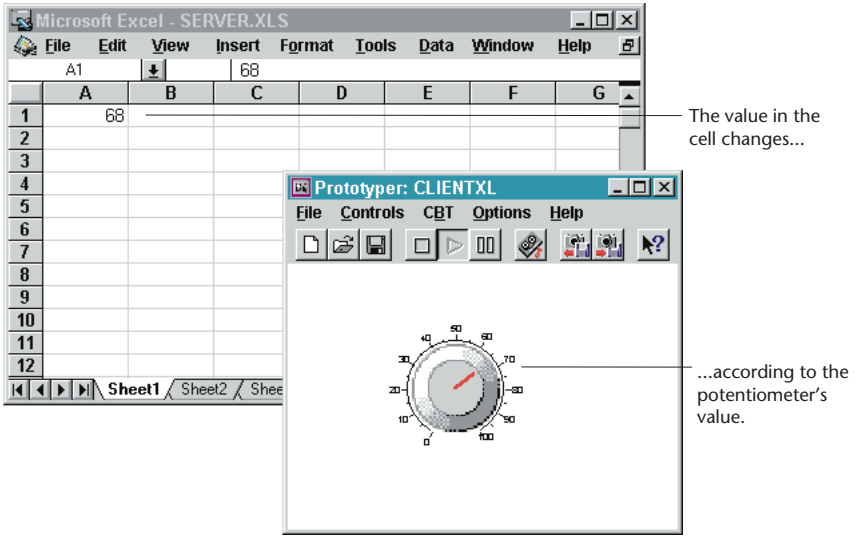
- Open a new Excel file and save it with the name *server.xls*.

  ❖ *NOTE: The name of this file must match the topic name you defined in Rapid's DDE client object.*

Running the application

To send data from Rapid:

1. Open the Prototyper alongside the Excel window.
2. Start the application.
3. Adjust the value of the potentiometer. As you do so, notice how the value in cell A1 of the spreadsheet changes.



Connecting Rapid to an Excel chart

To make the Excel file more attractive, you can add a chart (e.g., a column chart) to the sheet and base its data on cell A1. As you change the potentiometer value in Rapid, the Excel chart updates.

Example 3: Excel Performing Calculations on Data from Rapid

This example demonstrates how Excel can perform a calculation on data it receives **from** Rapid and how it can return the result **to** Rapid.

Editing the Rapid application

The Rapid application from example 2 needs two additions:

- A text display to display the result of the Excel calculation.
- A second item for the DDE client object that will receive the data from Excel.

To edit the application:

1. Add a text display below the potentiometer, accepting its default settings.
2. Add a second integer item to the DDE client object and call it R1C2. (Excel will perform the calculation in the second cell of the first row.)
3. In the Logic Editor, add a second mode activity:

   Display1.contents := DDE_Client1.R1C2

   This activity displays the value that the DDE client receives from Excel's R1C2. (Because the item has a hot link—the default setting—the client is automatically updated.)

Editing the Excel spreadsheet

The spreadsheet needs one addition:

- A formula in cell A2 that will perform a calculation on the data in cell A1. (A1's data is received from Rapid.) In this example, Excel will calculate the square of the integer that Rapid passes.

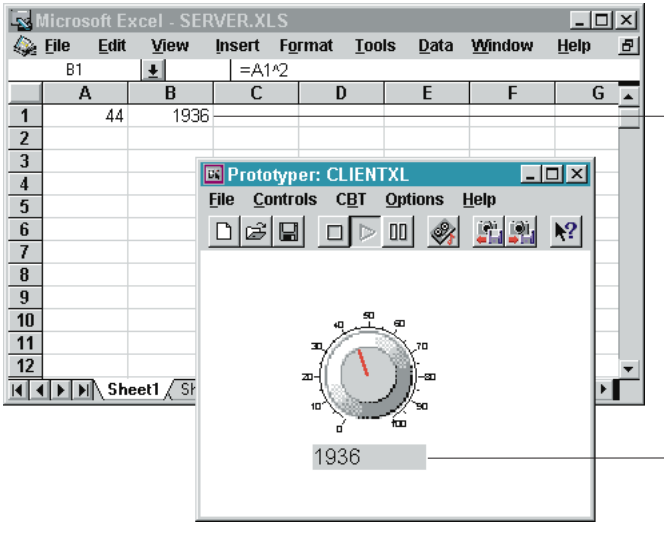
To add the formula:

- Insert the following formula into cell A2:

  ```
  =A1^2
  ```

Running the application

- Run the application alongside the Excel window and adjust the potentiometer's value. Rapid passes the data to Excel, Excel performs the calculation on the data (in cell A2) and then passes the result back to Rapid. Rapid displays the result in the text display.



❖ *NOTE: Calculating the square of an integer is, of course, a basic calculation that can easily be done in Rapid itself. The purpose of this example is simply to demonstrate how easy it is for Excel to perform any calculation and pass the result to Rapid.*



DLL OBJECT

The DLL object allows your Rapid application to call functions and/or routines stored in an external DLL (dynamic link library). A function/routine is written by a programmer in any programming language and is compiled into a DLL. The DLL object is useful in the following circumstances:

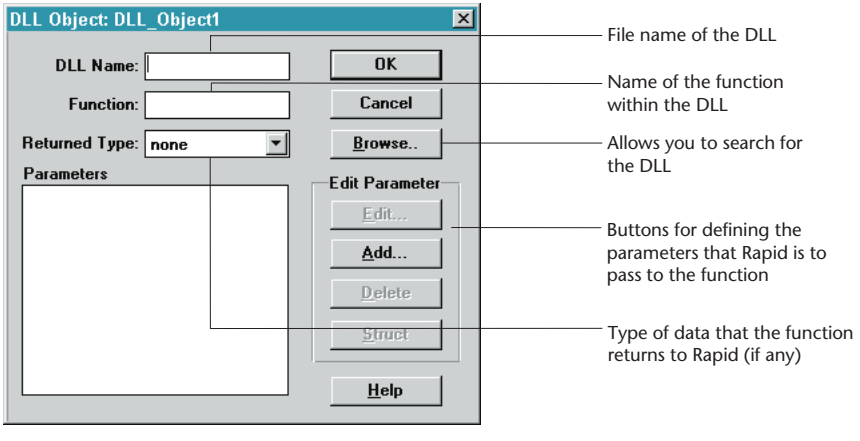
- You need to access hardware drivers or code written in a language such as C.
- You have a complex function (e.g., mathematical) and you would like to use it to manipulate data provided by Rapid.
- You need to communicate with third party software that provides its own DLLs.

❖ *NOTE: Rapid's DLL object calls 32 bit DLLs only; it cannot call 16 bit DLLs.*

An example of a use for the DLL object would be in a Rapid radar application, with a target moving across the screen. You could use a DLL object to call an external DLL that calculates and returns the target's next position based on data that it receives from Rapid (target type, current position, velocity, etc.) In such an application, the sequence of events would be:

1. Rapid assigns the values of the target's parameters to the DLL object's parameters.
2. Rapid calls the DLL's function.
3. The DLL performs the calculation and returns the new target coordinate values to Rapid.
4. Rapid displays the target in its new position on the radar screen.

When you add a DLL object, the following dialog box opens:



DLL name

The DLL name might need to be the DLL's **full** pathname, for example, C:\RADAR\DLL\TARGET.DLL. However, if the DLL is in any of the following folders, the DLL name alone is sufficient (i.e., TARGET.DLL):

- Application folder (that is, where the current application is located).
- Rapid folder (that is, where *rapid35.exe* is located).
- Any folder in your system search path.

❖ *NOTE: The usual extension for DLL files is .dll; other extensions though, such as .exe, are acceptable.*

Function

The function name is the full calling name of the function within the DLL. If you do not enter precisely the correct name, you will receive a runtime error when you run the application in the Prototyper.

❖ *NOTE: A DLL object can only call one DLL function.*

Returned type

The returned type is the type of data that the DLL function returns to Rapid (if any). The default value is "none," that is, the function does not return any data to Rapid. The other values conform to standard C definitions, as summarized below:

TYPE	LENGTH	RANGE
char	8 bits	-128 to 127
unsigned char	8 bits	0 to 255
short	16 bits	-32,768 to 32,767
unsigned short	16 bits	0 to 65,535
long	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	32 bits	0 to 4,294,967,295
float	32 bits	3.4×10^{-38} to 3.4×10^{38}

TYPE	LENGTH	RANGE
double	64 bits	1.7×10^{-308} to 1.7×10^{308}
string (signed char)	8 bits	A pointer to a null terminated sequence of chars.

❖ *NOTE: In 32 bit Windows, the int type is 32 bits wide (i.e., long). In 16 bit Windows, int was only 16 bits wide (i.e., short).*

If the DLL is to return a value, that is, the returned type is anything other than "none," the DLL object has as one of its properties *returnValue*, which has arithmetic or string functions (depending on the returned type).

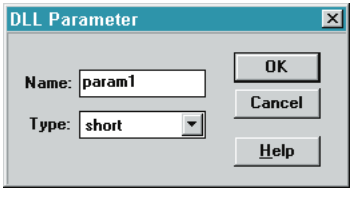
❖ *WARNING! You must enter precisely the correct type of data returned by the DLL function, otherwise running the application might cause Rapid to crash.*

Defining DLL Parameters

In the Parameters list box in the DLL object's dialog box, you define the parameters that Rapid is to pass to the DLL function. Each parameter becomes a property of the object that, depending on its type, is subject to number, integer or string functions.

To add a parameter:

1. Click the Add button; the DLL Parameter dialog box opens:



2. Enter a name for the parameter (remember, this name becomes a property name of the DLL object).
3. Select the type of data that will be passed by the parameter to the DLL function; the available data types are identical to the data types returned by the function (see previous page) with the exception of "none."

4. Click OK; Rapid adds the parameter to the Parameters list. The Parameters list shows the parameter's name and data type; each parameter is numbered sequentially, as shown below:

   1. param1 - long
   2. param2 - long
   3. param3 - char

To edit or delete a parameter:

- Select it in the Parameters list and click the Edit or Delete buttons.

❖ *WARNING! You must enter precisely the correct type of data passed to the DLL function, otherwise running the application might cause Rapid to crash.*

Pass by value vs. pass by reference

A DLL function uses data from a parameter in one of two ways:

METHOD	DESCRIPTION
Pass by value	The parameter passes a "copy" of its value to the DLL function. The original parameter remains unchanged; the result of the function is usually returned as the *returnValue* property.
Pass by reference	The parameter passes a reference to the value's address. The DLL function performs its calculation on the parameter itself, changing the parameter.

The DLL function itself determines whether the data is to be passed by value or by reference. If the function needs Rapid to pass the parameter(s) by reference, you need to define it as a structure.

❖ *NOTE: String is always pass by reference, it does not need to be in a structure.*

Structures

A structure is a complex data type made up of single parameters. Some functions require that one or more of their parameters be structured.

Some functions require structures of parameters within structures. In these cases, you need to define all the parameters as one long structure, in the correct order.

To define parameters as a structure:

1. Select the parameter that you want to include in a structure. (Shift-click to include more than one parameter; you cannot structure non-consecutive parameters.)
2. Click the Struct button.

   If you selected a single parameter (pass by reference), the characters "s:" appear before the structured parameter:

   1. s:param1 - long
   2. param2 - long
   3. param3 - char

   If you selected several parameters, they are grouped under one structure. Rapid renumbers the parameters accordingly, as below:

   1. s:param1 - long
   param2 - long
   2. param3 - char

To unstructure parameters:

- Select the structure's parameters and click Unstruct; the parameters are renumbered accordingly.

DLL Object Logic

Each of the parameters that you defined for the DLL object is a property of the object. The functions of the properties are arithmetic or string functions. In addition, the DLL Object's *self* property has the following function:

FUNCTION	DESCRIPTION
call	Executes the DLL call and returns a value (if the return value type is not "none").

❖ *NOTE: If your DLL object allocates resources external to Rapid, such as opening a file or hooking to the mouse, make sure that you include exit activities (e.g., in the root mode) that free those resources.*

DLL OBJECT EXAMPLES

This section sets out two examples of how to use Rapid DLL objects.

- In the first example, Rapid uses two functions from a Windows DLL to change and display the position of the mouse cursor.
- In the second example, Rapid calls a function in a custom-made DLL that calculates the diagonal length of a cuboid of given dimensions.

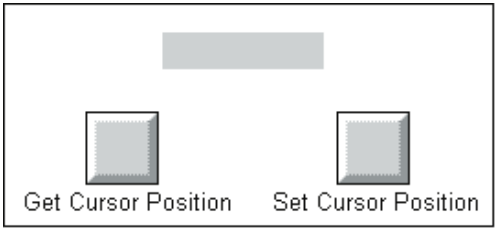
Example 1: Changing the Cursor Position

This example makes use of the USER32.DLL file found in your Windows system directory (or system32 directory in Windows NT). The example uses two functions:[1]

- *GetCursorPos*—retrieves the screen coordinates of the cursor's current position.
- *SetCursorPos*—sets the position, in screen coordinates, of the cursor.

The application needs two DLL objects, one to call each of the functions. As well as the DLL objects, the application should have a text display (for displaying the cursor's coordinates) and two toggle pushbuttons (one to activate each DLL call).

The application's layout should look something like:

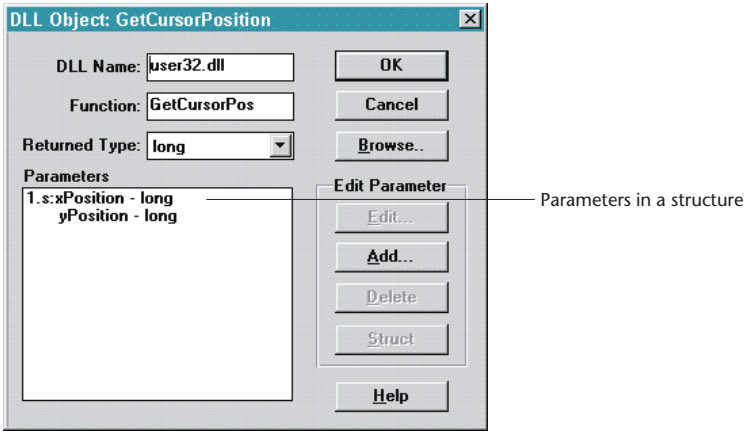


Defining the DLL object for GetCursorPos

From the documentation on the *GetCursorPos* function, we know that it has a structure (POINT) made up of two long types. The first long type receives the mouse cursor's x-coordinate and the second receives the y-coordinate.

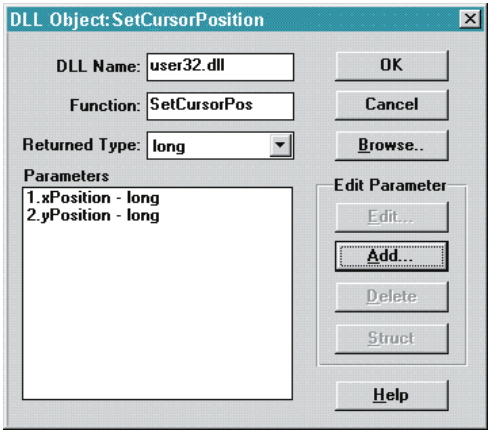
1 All information on the GetCursorPos and SetCursorPos functions is taken from *Microsoft® Windows™ SDK: Programmer's Reference, Volume 2*.

This function returns a non-zero value if it succeeds (BOOL = long), or zero if it fails. In this example, we ignore the return value. The dialog box for the first DLL object is shown below:



Defining the DLL object for SetCursorPos

From the documentation on the SetCursorPos function, we know that it has two long type parameters (not in a structure). The first long type specifies the mouse cursor's new x-coordinate and the second specifies the new y-coordinate. Again, this function returns a non-zero value if it succeeds, or zero if it fails and we ignore the return value. The dialog box for the second DLL object is shown below:

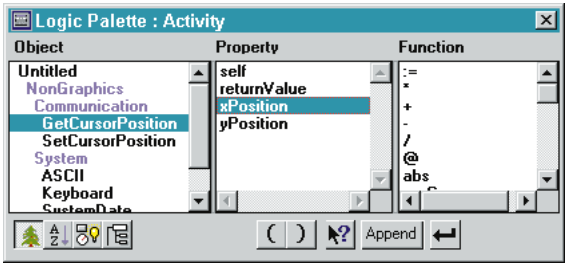


The application's logic

The DLL objects appear in the Logic Editor with four properties each:

- *self*
- *returnValue* (not used in this example)
- *xPosition* (the name of the first DLL parameter)
- *yPosition* (the name of the second DLL parameter)

The *self* property has the *call* function and the parameter properties have arithmetic functions, as shown in the Logic Palette below:



To get and display the cursor position:

Build a mode activity that continuously assigns the values of GetCursorPosition's x-coordinate and y-coordinate to the text display, i.e.,

Display.contents := GetCursorPosition.xPosition | ' ' | GetCursorPosition.yPosition

The empty string in-between the two DLL parameters serves to separate the two numbers in the text display.

To activate the DLL:

Build an internal transition that is triggered as long as one of the pushbuttons is in, i.e.,

Trigger: **& GetCursorButton is in**
Action: **GetCursorPosition call**

As long as the pushbutton is in, the DLL object continuously calls the function that, in response, continuously updates the x- and y-coordinates.

To set the cursor position:

You need Rapid to execute the following actions:

1. Get the current cursor position.

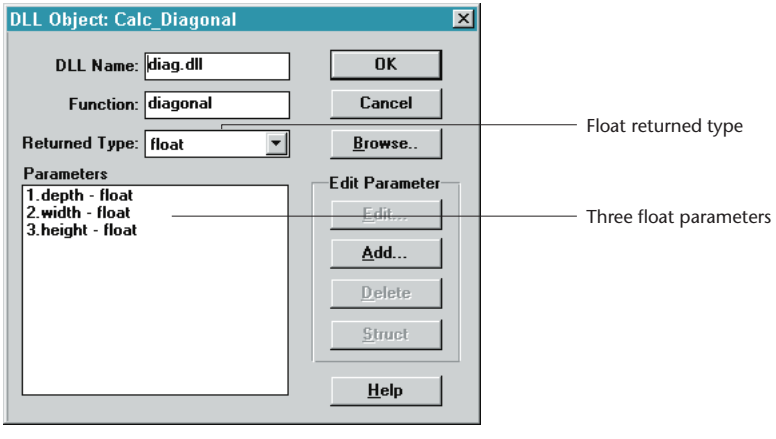
2. Assign the current cursor position's x-coordinate to the SetCursorPosition object's *xPosition* property.
3. Assign the current cursor position's y-coordinate to the SetCursorPosition object's *yPosition* property.
4. Change the value of the SetCursorPosition object's *xPosition* and *yPosition* properties.
5. Call the SetCursorPos function; the DLL function sets the cursor's position to the values of the DLL object's properties.

The following logic carries out the above steps:

Trigger: **& SetCursorButton is in**
Actions: **GetCursorPosition call**
SetCursorPosition.xPosition := GetCursorPosition.xPosition
SetCursorPosition.yPosition := GetCursorPosition.yPosition
SetCursorPosition.xPosition changeBy: 2
SetCursorPosition.yPosition changeBy: 2
SetCursorPosition call

Example 2: Performing a Calculation

This example uses a theoretical custom DLL that calculates the diagonal length of a cuboid of given dimensions. The DLL requires three parameters of the float type; each parameter gives one dimension of the cuboid. This DLL has a return value, also of the float type. The DLL object's dialog box looks like:



The DLL object has four properties, *depth*, *width* and *height* (the three DLL parameters) and *returnValue*. The application also needs to have a text display (where the result of the calculation is displayed) and three variables (number objects) that will assign values to the DLL object's parameters.

To receive the result of the calculation:

You need Rapid to execute the following activities (they can all be entry activities of a mode):

1. Assign a value to the DLL object's *depth* parameter.
2. Assign a value to the DLL object's *width* parameter.
3. Assign a value to the DLL object's *height* parameter.
4. Call the DLL function; the DLL function calculates the diagonal and returns a value.
5. Display the returned value.

The following logic carries out the above steps:

```
Calc_Diagonal.depth := Number1
Calc_Diagonal.height := Number2
Calc_Diagonal.width := Number3
Calc_Diagonal call
Display.contents := Calc_Diagonal.returnValue
```



COMMLINK OBJECT

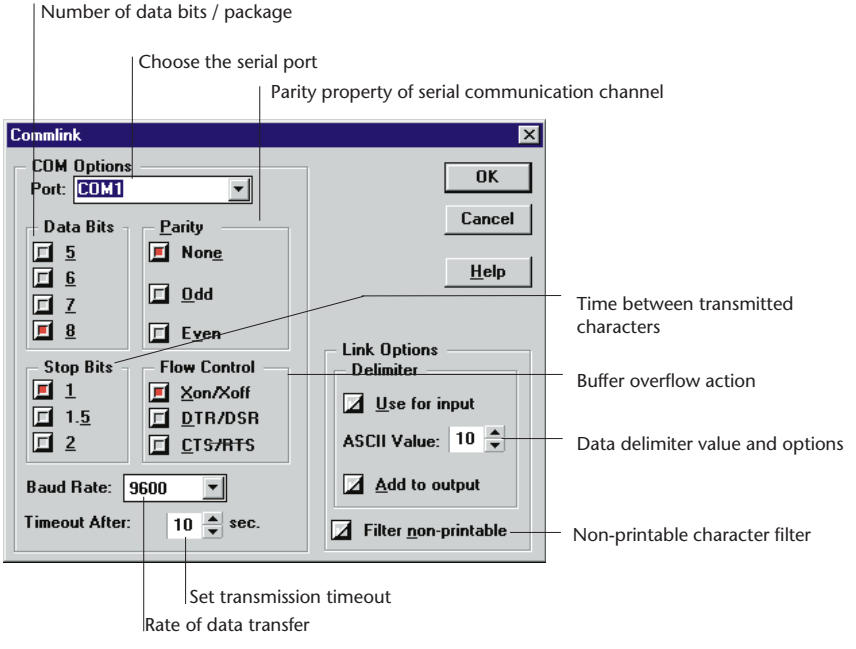
The Commlink object allows a Rapid application to communicate with external devices via one of the computer's serial ports (COM1, COM2, etc.).

Each Commlink object can connect to only one port, but an application can contain several Commlink objects.

The external device might be:

- A second computer running Windows 95/NT's HyperTerminal.
- A second computer running a Rapid application that uses a Commlink object.
- Any device that is able to receive data from a serial port via a serial cable.

When you click the More button in a Commlink object's parameter pane, the following dialog box opens:



PARAMETER	COMMENTS
*Port**	Opens a drop-down list of default serial port names. You can add a name to the list.
*Data Bits**	Determines the number of data bits in each data packet. Most characters are transmitted in 7 or 8 data bits, though some computers require 5 or 6 bits.
*Stop Bits**	Stop bits are not actually bits – they are the timing units between characters.
*Parity**	If the data bits is set to 8, set the parity to none.
*Flow Control**	Determines the action to be taken if the buffer becomes too full to receive more data from the remote system. Xon/Xoff causes Commlink to pause when the input buffer fills. This is the standard method for most systems. DTR/DSR specifies the DTR (data-terminal-ready) flow control. CTS/RTS specifies the RTS (request-to-send) flow control.
*Baud Rate**	Determines the rate of data transfer, given in bits-per-second (bps). Choose from a list of standard baud rates.
*Timeout**	Determines the number of seconds between transmission of any two data bits, after which the transfer stops and an error code is generated.
Delimiter: Use for input	If checked, Rapid recognizes an incoming delimiter as signalling the end of a package of data.
Delimiter: Add to output	If checked, Rapid automatically adds a delimiter to every string it sends.
*Delimiter: ASCII Value**	Determines the ASCII value for the delimiter.

PARAMETER	COMMENTS
Filter non-printable	If checked, Rapid filters out non-printable characters (i.e., ASCII values less than 32) from data.

* These parameters can be changed during runtime.

Using the Commlink Object

To communicate between two computers, you need the following hardware:

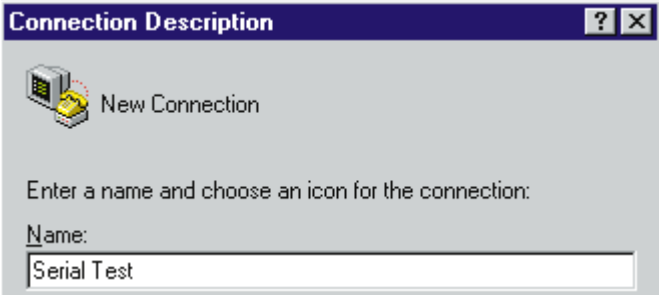
- An available serial port.
- A crossed cable to connect between the two serial ports.

❖ *NOTE: When setting up a Commlink, you must ensure that the various communication parameters are identical to those of the external device.*

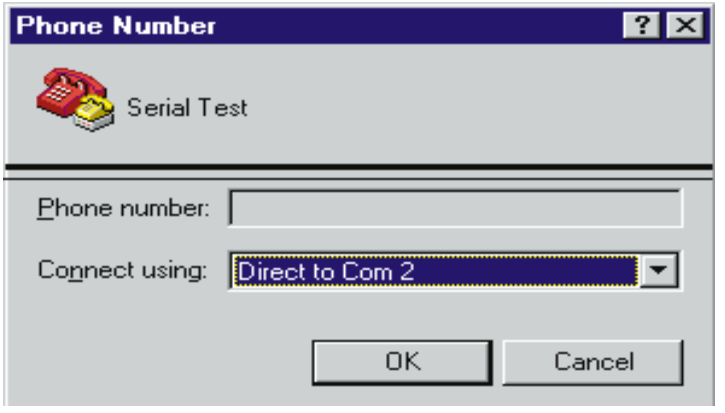
Testing the Link (optional)

The procedures below are carried out on each machine.

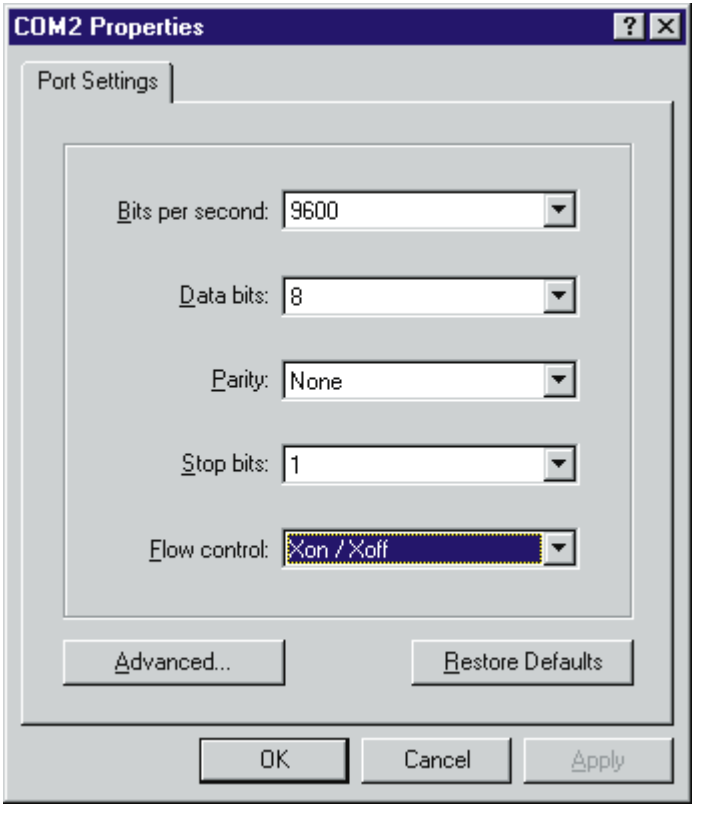
1. From the Windows 95 Start menu, select Programs|Accessories|HyperTerminal.
2. Launch *Hypertrm.exe*.
3. In the Connection Description dialog box, enter the name 'Serial Test' and click *OK*.



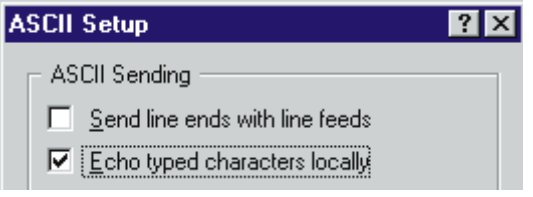
4. In the Phone Number dialog box choose the appropriate COM port connection in the 'Connect using' combo box and click *OK*.



5. Set the port properties as shown and click *OK*.



6. In the HyperTerminal dialog box, select File | Properties, click the Settings tab and click the *ASCII Setup* button. Check the 'Echo typed characters locally' option and click *OK*.



7. Test the serial link by typing text on each keyboard. You should see the correct text appear on the other screen.
8. Close HyperTerminal.

❖ ***NOTE: It is very important to close HyperTerminal. Otherwise, Commlink will not work.***

Communicating via Commlink

In this example, both computers are running Rapid applications that contain Commlink objects. See the *Object Reference* manual for an explanation of all Commlink properties, functions and events.

1. Make sure that the values of the various Commlink communication properties (that is, baud rate, data bits, stop bits and flow control) are the same on both computers. You can do this either through the initial settings in the Object Layout, or through entry activities that are executed at runtime.
2. Execute the *connect* function on the first computer as an activity or action:

   Commlink1 connect
3. Execute the *connect* function on the second computer.
4. If the connection is successful (i.e., the *is connected* condition is true), you can receive data using the *input* property, and send data using the *output* property (see the sections immediately below).
   If the connection is not successful, the *openError* event is generated.
5. If either side executes the *disconnect* function, the connection is ended.

❖ *NOTES: Executing the connect function while the connection is already open causes reconnection, i.e., disconnect followed by connect with the current property values.*
Changing property values while the connection is already open does not influence the current connection.

Getting Input Data

1. Use the *dataArrived* or *delimiterArrived* event to detect when data has been received from the connected port.
2. Get the data (a string) by using it, such as by assigning it to a string object or to a text display's contents:

   String1 := Commlink1.input

   or

   Display1.contents := Commlink1.input
3. Getting the string clears the *input* property.

Thus, a typical series of logic statements would be:

Transition	Event Trigger	Action
Internal	Commlink1 delimiterArrived	String1 := Commlink1.input

❖ *NOTE: You can also use the* **clear** *function to remove the data from the input property.*

Sending Output Data

To send data to the port, simply assign a string to the *output* property:

Commlink1.output := 'Hello there!'

The output buffer is limited to 32,500 characters.

Using Data Delimiters

Either in the Object Layout or during runtime, you can designate the ASCII value of a character which Rapid uses to signal the end of a data transmission.

❖ *NOTE: If you are passing data in a "double-byte" language such as Japanese or Korean, avoid using a delimiter greater that ASCII 127.*

Add to Output

When the Add to Output checkbox is checked in the Commlink More dialog, Rapid automatically adds a delimiter to every string it sends. If the Add to Output checkbox is not checked, Rapid sends the string without adding the delimiter.

Examples:

Case 1: Add to Output is checked

RAPID ACTION	STRING SENT
Commlink.output := 12345	12345<delimiter>
Commlink.output := 678 | <delimiter>	678<delimiter><delimiter>

Case 2: Add to Output is not checked

RAPID ACTION	STRING SENT
Commlink.output := 12345	12345
Commlink.output := 678 | <delimiter>	678<delimiter>

Use for Input

When the Use for Input checkbox is checked, Rapid recognizes an incoming delimiter as signalling the end of a data package.

If the Use for Input checkbox is not checked, Rapid ignores the delimiter and treats it like any other data character. Also, if a delimiter is not used, all the received data is kept until either the *clear* function is performed, or the data is removed from the *input* property by assigning it to a string object.

Examples:

Case 1: Use for Input is checked on the receiving machine

1. The string 12345<delimiter> arrives from the sending machine.
2. The string 678<delimiter> arrives from the sending machine.
3. The action **Display1.contents := Commlink1.input** is implemented on the receiving machine.
4. The resulting display is 12345.
5. The action **Display1.contents := Commlink1.input** is again implemented on the receiving machine.

6. The resulting display is 678.

Case 2: Use for Input is not checked

1. The string 1234510<delimiter> arrives from the sending machine.
2. The string 678 arrives from the sending machine.
3. The action **Display1.contents := Commlink1.input** is implemented on the receiving machine.
4. The resulting display is 12345<delimiter>678.

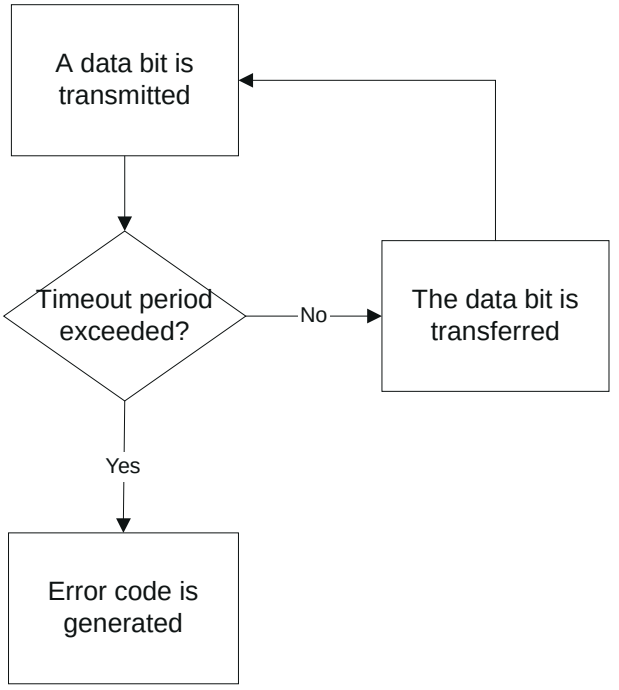
Commlink Object Logic

Refer to the *Object Reference* for a summary of the properties, functions, and events of the Commlink object.

❖ *NOTE: Any change to Commlink's settings during runtime are effective only following a new connect action.*

Error Codes

During data transfer, if the time between any two data bits exceeds the timeout value (set in the Object Layout as explained on p. 7-26, or through the logic), Rapid assumes that an error has occurred.



Schematic representation of the *timeout* property's effect on data transmission

When an error occurs, Rapid generates one of the following events:

EVENT	DESCRIPTION
openError	Generated when a timeout error occurs during an attempt to connect to a specified port.
readError	Generated when a timeout error occurs during data reception.
writeError	Generated when a timeout error occurs during an attempt to transmit data.

When an error occurs, Rapid also assigns an integer to the *errorCode* property. The value of this read-only integer allows you to determine the nature of the error.

You can use the *errorCode* property in your logic, for example, to display a message in a text display following a certain error:

Condition: **& Commlink1.errorCode = 256**
Action: **Display1.contents := 'Output buffer full.'**

The table below summarizes the error codes, including the event each one generates:

ERROR CODE	EVENT	MEANING
1	readError	An input buffer overflow has occurred. There is either no room in the input buffer, or a character was received after the end-of-file (EOF) character. Input (hardware: DTR/DSR and CTS/RTS) flow control is neccessary to completely alleviate this problem.
2	readError	A character-buffer overrun has occurred. The next character is lost. This error occurs when new data arrives before the communications driver completely receives old data. This can occur when the transmission speed is too high for the type of communications hardware or CPU. It can also occur when the operating system is not free to service the communications hardware. The only way to alleviate this problem is to apply some combination of: • Decreasing the transmission speed • Replacing the communications hardware • Increasing the CPU speed Sometimes third-party hardware drivers that are not very efficient with CPU resources cause this error.
4	openError	The hardware detected a parity error.
8	openError	The hardware detected a framing error.
16	openError	The hardware detected a break condition.
256	writeError	The application tried to transmit a character, but the output buffer was full.

ERROR CODE	EVENT	MEANING
512	readError or writeError	Windows 95 only: A time-out occurred on a parallel device.
1,024	readError or writeError	An I/O error occurred during communication with the device.
2,048	readError or writeError	Windows 95 only: A parallel device is not selected.
4,096	readError or writeError	Windows 95 only: A parallel device signaled that it is out of paper.
32,768	openError	The requested mode is not supported, or the hFile parameter is invalid. If this value is specified, it is the only valid error.

COMMLINK OBJECT EXAMPLE

This example shows how to pass data between the same Rapid application which is running simultaneously on two separate machines connected at their serial ports by an RS232 (null-modem) cable. It is assumed that the serial connection has been tested.

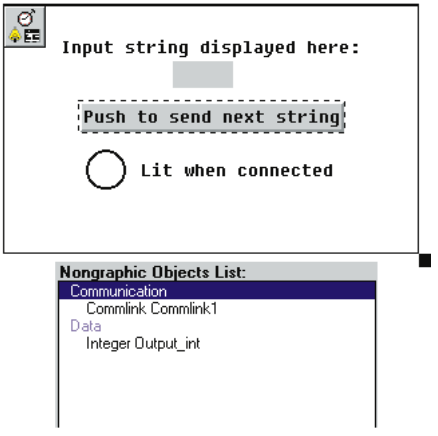
The application has a Commlink object with the following (default) settings:

- Port: COM1
- Data Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: Xon/Xoff
- Baud Rate: 9600
- Timeout After: 10 seconds
- Delimiter: For input and output

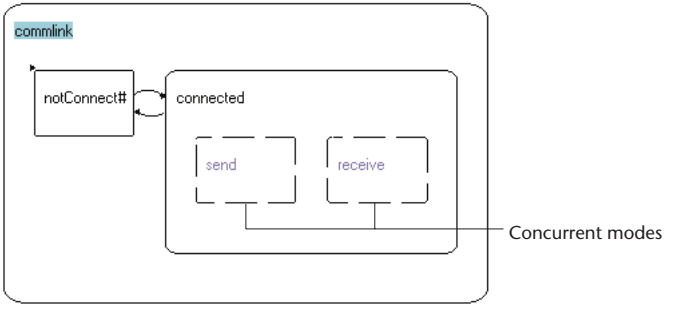
The application also contains:

- a text display (3 characters in length): for displaying input from the remote application
- an integer: for providing the output data
- a pushbutton: for sending the data and incrementing the integer
- a lamp which is lit while a connection exists

The Object Layout and list of nongraphic objects would look something like this:



The modes and transitions in the example application are represented in the following state chart:



The logic is implemented as follows:

1. The connection is established by the entry activity of ‘notConnected’ mode:

   Commlink1 connect

2. The transition from ‘notConnected’ to ‘connected’ is the condition:

   & Commlink1 is connected

   The transition from ‘connected’ to ‘notConnected’ is the condition:

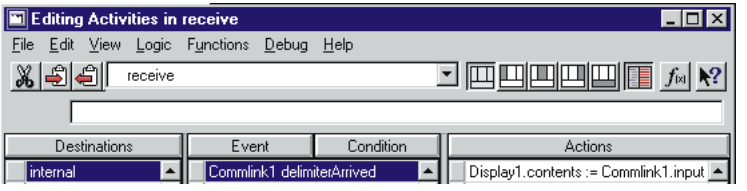
   & not Commlink1 is connected

3. The lamp is lit while there is a connection (and turned off when there is not a connection) by the entry and exit activities of ‘connected’:

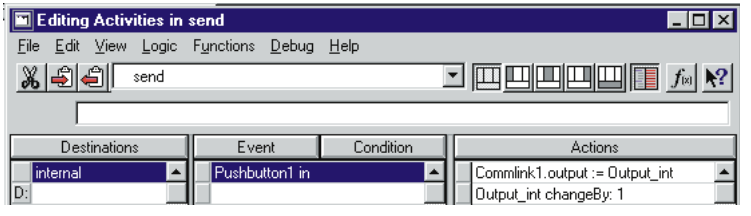
   Lamp1 on

   Lamp1 off

4. The receiving and displaying of data are implemented in the concurrent mode ‘receive’ as follows:



5. The sending of data (and incrementing the integer) are implemented in the concurrent mode ‘send’ as follows:



Troubleshooting

If you encounter communication difficulties (i.e., data is not being transmitted and/or received), you should add another text display in the Object Layout and the following mode activity in the root mode:

Display2.contents := Commlink1.errorCode

Use the error code summary on p. 7-34 to pinpoint the cause of the problem.



SIMULATION MANAGER OBJECT

The simulation manager object enables you to establish a connection to a student Rapid, that is a Reviewer or Prototyper set to student mode,[1] through a direct DDE link. Using the simulation manager object you can control the behavior of the student Rapid with your application. The simulation manager thus acts as a substitute for an authoring tool. This section explains:

- When to use the simulation manager.
- How to set up the simulation manager in the Object Layout.
- How the simulation manager interacts with a Rapid simulation.
- How to build logic with the simulation manager that will allow you to control a Rapid simulation.

❖ *NOTE: See Appendix B: "CBT Interface Message System" for information on how to pass messages directly between Rapid and any external application that supports DDE communications—without the need for an authoring tool or the simulation manager.*

The Simulation Manager: When?

In general, you use the simulation manager when you are building a Rapid simulation that is to be controlled by an external application, such as an authoring tool, as part of computer-based training (CBT) courseware. The simulation manager allows you to test how the simulation responds when it is controlled by the external application.

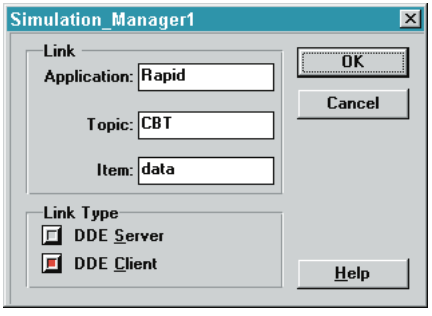
You run two Rapid applications simultaneously:

- The simulation.
- The application (containing the simulation manager) that controls the simulation.

1 To establish a connection to the Prototyper, it has to be in student mode—see Chapter 20: "The Prototyper."

Defining the Simulation Manager

The Simulation Manager dialog box is shown below.



The application, topic and item names that you enter depend on whether the simulation manager is to control a Reviewer or a Prototyper.

❖ *NOTE: For details on DDE and application, topic and item names, see “DDE Basics” on p. 7–2.*

If the simulation manager is to control a Reviewer:

The application, topic and item names must be identical to the options that appear in the Reviewer’s command line. If the command line does not explicitly give application, topic or item names, their default values are “Rapid”, “CBT” and “data”, respectively.

For details on the Reviewer’s command line options, see p. , “Using the Reviewer’s Command Line Options”.

If the simulation manager is to control a Prototyper

When you intend to link to a Rapid Prototyper, the simulation manager’s item name must be identical to the item name as defined in the simulation Prototyper’s CBT Channel Setup dialog box (see p. 20–15).

You can usually accept the default values of the Application and Topic fields (“Rapid” and “CBT” respectively) to establish a link with a simulation.

The simulation manager as a DDE server

If you define the simulation manager as a DDE server, then when you start the Prototyper, Rapid opens the communication channel and establishes the

connection with the simulation. When you stop the Prototyper, Rapid closes the channel and the connection.

The simulation manager as a DDE client

If you define the simulation manager as a DDE client, then when you start the Prototyper, Rapid polls for the server automatically. When the simulation (as a server) is opened, Rapid opens the channel and the *studentOpened* event is triggered.

How the Simulation Manager Interacts with Rapid

The simulation manager can control a Rapid simulation on two levels:

- The Rapid window
- The simulation (application) itself

Controlling the Rapid window

The simulation manager can send commands to the Rapid simulation's window. The commands give it total control over the window's size, position and appearance, as well as basic control over the simulation (start, stop, pause, etc.)

Controlling the application

The simulation manager can control the behavior of the simulation itself. This has far-reaching CBT consequences: you can provide a student with detailed instructions, demonstrate procedures and examine his or her progress. This section explains in detail how the simulation manager controls Rapid.

The key to understanding how the simulation manager can control the behavior of a Rapid simulation lies in understanding the concept of **event flow**. In a running simulation, there are certain objects that generate events when the user clicks them. For example, when the user clicks a switch, or a pushbutton, an event is generated. These events determine the flow of the simulation's behavior.

Normal event flow

When Rapid runs normally, the flow of the events is completely internal without any link to an external program. This situation is illustrated below.



Flow of events in independent Rapid

When you switch a Rapid simulation to student mode, a DDE link, known as the CBT channel, opens between the simulation and the simulation manager.

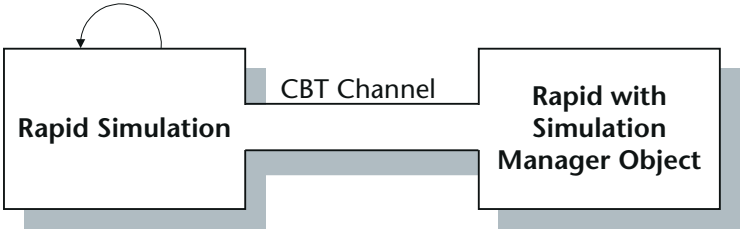
The simulation manager can instruct the simulation to be in any of three different event flow modes, each one suitable for different CBT situations:

- Normal mode
- Monitor mode
- Reroute mode

Normal mode

In normal mode, the flow of events within Rapid is normal—i.e., completely internal. The simulation manager receives no information about the student's actions (the events generated in the simulation), though it **can** receive certain information about the simulation's status if it requires.

This mode is suitable for free-play sessions where you want to give the student maximum independence without severing the link.



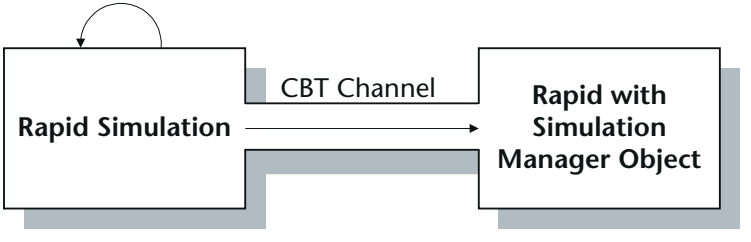
Flow of events in normal mode

Monitor mode

In monitor mode, the events flow both internally within the simulation **and** externally to the simulation manager. The simulation continues to behave as normal but the simulation manager is continuously updated about the student's actions.

This mode is suitable for situations where you want to keep a record of the student's actions, say for a test or examination. In monitor mode you can also watch the student's progress and send messages of encouragement or advice without interfering in the simulation's behavior.

For example, if you want to send an event to the simulation, to demonstrate a certain procedure, the simulation manager first has to send the *enableExternalEvents* function.



Flow of events in monitor mode

Causing Rapid accepts external events

Once the simulation accepts external events, the simulation manager can send an event, or a series of events, to the simulation causing it to behave

in a certain way. You can then use the application that contains the simulation manager object to demonstrate to the student how to perform particular procedures.

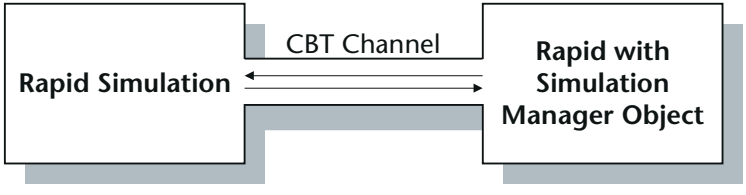
You can cause a simulation to accept external events any time it is in student mode. This means that the simulation manager can send events when the simulation is in normal, monitor or reroute mode.

❖ *NOTE: The enableExternalEvents function is only necessary for events initiated by the simulation manager using the sendEvent: function.*

Reroute mode

Reroute mode allows direct interaction between the simulation manager and the simulation. All events are routed to the simulation manager, and the simulation does not respond to the student's actions until the simulation manager returns the event. This way you can follow the student's actions one at a time and be selective about which of the actions you confirm.

This mode is suitable for instruction sessions where you want to demonstrate a procedure and then have the student repeat it step by step. If the student makes a mistake, do not return the event to Rapid—send a message and/or send the correct event as a reminder.



Flow of events in reroute mode

The Simulation Manager's Properties

The simulation manager object has the following properties:

PROPERTY	DESCRIPTION
self	Has most of the simulation manager's major functions, such as those that control the simulation's window, start the simulation and send events.
applicationName	Contains the application name, as a read-only string value, following the sending of the *queryApplicationName* function.
eventObject	Contains, as a read-only string value, the name of the Rapid object that last sent an event to the simulation manager. For example, if the last event received by the simulation manager was *Switch.position2 make &*, the *eventObject* property would contain the string "Switch".
eventProperty	Contains, as a read-only string value, the name of the object property that appeared in the last event that the simulation manager received. For example, if the last event was *Switch.position2 make &*, the *eventProperty* property would contain the string "position2".
eventSymbol	Contains, as a read-only string value, the symbol of the last event that the simulation manager received (as it appears in Rapid recorder files). For example, if the last event was *Switch.position2 make &*, the *eventSymbol* property would contain the string "make".
recorderStatus	Contains, as a read-only string value, the current recorder status (*open*, *close*, *stop*, *play*, *cue*, *record*, *rewind* or *step*).
simulationStatus	Contains, as a read-only string value, the current status of the simulation (*start*, *stop*, *pause* or *resume*).
studentName	Contains, as a read-only string value, the simulation's student name following the sending of the *queryStudentName* function.

PROPERTY	DESCRIPTION
timeStamp	Contains, as a read-only string value, the time when the last event was received. The time, measured in seconds, is the period that has elapsed since the simulation started.
windowPosition	Contains the upper-left position of the student simulation window, as point data (x@y), following the *queryPosition* function.
windowSize	Contains the size of the student simulation window, as point data (x@y), following the *queryPosition* function.

Controlling the Simulation Window

The *self* property has several functions that enable control over the simulation's window. These include the self-explanatory functions of *maximizeWindow*, *minimizeWindow*, *normalWindow* and *hideWindow*. In addition, there are some less familiar functions that are listed here:

FUNCTION	DESCRIPTION
fillScreen	The simulation window fills the student's screen. The screen appears without the title or menu bars, but with scroll bars if necessary. The student is unable to move or resize the window.
slaveWindow	The simulation window appears without the title and menu bars and the student is unable to move or resize it. This function is cancelled by *releaseSlave*.
stayOnTop	The simulation window always remains on top, even if an overlapping window is active. This function is cancelled by *cancelStayOnTop*.

Getting Information about the Simulation

The simulation manager object can send various queries to the simulation to obtain certain information about its condition. When the simulation manager sends a query through the CBT channel, the following series of operations occurs:

1. The simulation sends its reply back to Rapid.
2. When the the reply arrives, the simulation manager generates an event.
3. The contents of the reply are assigned to the contents of the relevant property, until another reply of the same type arrives.

The following information can be obtained by sending queries:

- Whether the student Rapid is open.
- Name of the application running in the student Rapid.
- Name of the student.
- Position of the simulation window.
- Size of the simulation window.

Example:

The following example sends a query to the simulation to obtain its application name:

```
SimMan queryApplicationName
```

When the reply to this query arrives, Rapid generates the following event:

```
SimMan applicationNameArrived &
```

The name of the student's application is assigned to the *applicationName* property. You can use this information as any other read-only string value; for example you can display it in a text display:

```
Display1.contents := SimMan.applicationName
```

❖ *NOTE: The position and size properties contain their information in the form of point data (x@y) and not as strings.*

Controlling the Simulation's Response

The simulation manager object allows you not just to monitor and control the student Rapid, but also to control the simulation's response to the student's actions.

To be able to completely control the student's application, you need to send two commands via the CBT channel: the first tells the student Rapid to respond to any events it receives from an external source. This command is the *self* property's *enableExternalEvents* function.

The second command tells the student Rapid to reroute any event that the simulation generates to the simulation manager, and only to respond to events that it receives **from the simulation manager**. This command is the *self* property's *rerouteMode* function.

The result of these two functions is: when the student performs an action (e.g., clicks a switch), the simulation does not respond until Rapid returns the event using the simulation manager object.

To enable the student Rapid to accept external events:

- In the Logic Editor, build the following action or activity:

  SimMan enableExternalEvents

  It is usual to place this logic as an entry activity of the mode in which the application controls the simulation. If you want to be able to control the simulation at any time during the application, place it in the root mode.

To tell the simulation to reroute its events:

- Build the following action or activity:

  SimMan rerouteMode

  It is usual to place this logic together with the *enableExternalEvents* activity.

Receiving Events from the Simulation

When the simulation is in monitor or reroute mode, it sends its events to the simulation manager. The events are sent as DDE strings. When the simulation manager receives an event via the CBT channel, Rapid triggers the *eventArrived* event. Rapid breaks down the DDE string's components into its separate pieces of information and stores them in the *eventObject*, *eventProperty*, *eventSymbol* and *timeStamp* properties.

For example, the following string describes a switch named “Switch1” being connected to its “up” position, 6.37 seconds after the Prototyper was started:

Switch1#up#make# 6.37

When the simulation sends this string to Rapid, the *eventObject* property receives the value “Switch1”, the *eventProperty* property receives the value “up”, the *eventSymbol* property receives the value “make” and the *timeStamp* property receives the value “6.37”.

❖ *NOTE: The timeStamp property’s value is a string not a numeric value. To convert it to a numeric value, use the following action:*
Number := SimMan.timeStamp asNumber

Sending Events to the Simulation

You can send events to the simulation with the simulation manager using the *self* property’s *sendEvent:* function. This allows you to directly control the behavior of the simulation. (The simulation must be in the accept external events mode.) This function must take the following format:

SimMan sendEvent: ‘eventObject#eventProperty#eventSymbol’

where:

- *eventObject* is a unique object name.
- *eventProperty* is the object’s property that you want to change.
- *eventSymbol* is the name of the event, as it appears in a recorder file.

For the “Switch1” example given above, the syntax would be:

SimMan sendEvent: ‘Switch1#up#make’

Note that the string which the simulation manager sends is case-sensitive. For example, if the object name part of the event was “switch1” instead of “Switch1,” the event would not have any effect.

❖ *NOTE: The simulation manager can send events to the simulation at any time (irrespective of whether the simulation is in normal, monitor or reroute mode) as long as the simulation is accepting external events.*

Automatically Returning Events

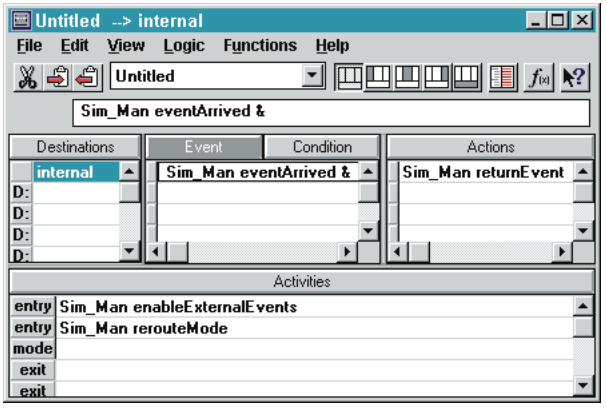
When the simulation is in reroute mode, you can watch what the student is doing while returning most events to the simulation. Occasionally you might want to send a particular event of your own. In this situation, use the

self property's *returnEvent* function in conjunction with the *eventArrived* event. This will automatically return all the simulation's events.

Every time the simulation sends an event to the simulation manager, the *eventArrived* event is generated in the simulation manager's application. Use this event to trigger an internal transition, with *returnEvent* as the action.

With this logic, each time the student generates an event in the simulation (e.g., by clicking a pushbutton), the simulation manager automatically "bounces it back."

The logic that you build would look like:



Checking for a Particular Event

In many cases, you will want to let the student carry out a certain procedure while checking that he or she is following the correct steps. If the student makes a mistake (e.g., presses the wrong button), you might want to send a message or demonstrate the correct action.

Using a condition, you can check each event that the simulation sends and, if the simulation manager detects that the wrong event has arrived, it will trigger a transition. The simulation must be in monitor or reroute mode.

For example, if the student is supposed to connect Switch4 to position3, you would use the following trigger to check for the event:

SimMan eventArrived & not (SimMan.eventObject = 'Switch4' and SimMan.eventProperty = 'position3' and SimMan.eventSymbol = 'make')

This condition tests for when the incoming event is **not** the correct switch being connected to the correct position. If the condition is true, the student has not performed the right action.

It is not always necessary to check the values of all three properties (*eventObject*, *eventProperty* and *eventSymbol*). For example, you might only need to check whether the student has used a particular object (irrespective of the actual event).

❖ *TIP: A long condition such as the above is an ideal candidate for being converted into a multi-line, user-defined function (see p. 18–3).*

Setting the Simulation's Size and Position

To set the simulation window's position and size:

1. Assign (or change) the value of the simulation manager object's *windowPosition* property.
2. Assign (or change) the value of the simulation manager object's *windowSize* property.
3. Use the *self* property's *setPosition* function.

The window position and size values are given as points (x@y). An example of the series of actions/activities required is:

```
SimMan.windowPosition := 300 @ 600
SimMan.windowSize := 400 @ 500
SimMan setPosition
```

The *windowSize* and *windowPosition* properties have several numeric functions that allow you to take the current values and manipulate them to receive new values.

Sending Messages to the Student

The simulation manager *self* property's *displayMessage:* function allows you to send a message window that opens on the student's screen. If you are monitoring the student's events, you can use this function to inform the student when he has made a mistake or when it is time to take a break.

To send a message:

- Build an action or activity with the following form:

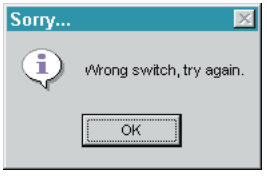
```
SimMan displayMessage: 'title text#message text'
```

where “title text” is the text that is to appear in the message window’s title bar and “message text” is the message text.

❖ *NOTE: You cannot use exclamation marks (!) in your title or message text.*

The message window shown below is the result of the following activity:

SimMan displayMessage: ‘Sorry...#Wrong switch, try again.’



CHAPTER 8

Object Libraries

You can store Rapid objects in special files, called libraries. Once an object is stored in a library, you can re-use the object in several applications, without needing to rebuild it each time. You can add an object to a library from the Object Layout and from the Object Editor.

For example, you could construct a 10-position rotary switch (with specific start and stop values, step angle, etc.) and then save it to a library. The next time you need that type of rotary switch, you could add it to the application from the library instead of recustomizing a standard rotary switch.

You could also create your own pushbutton in the Object Editor, using customized bitmaps, and store it in a library.

To manage libraries and the objects they contain, you use the Librarian.

In this chapter you will read how to:

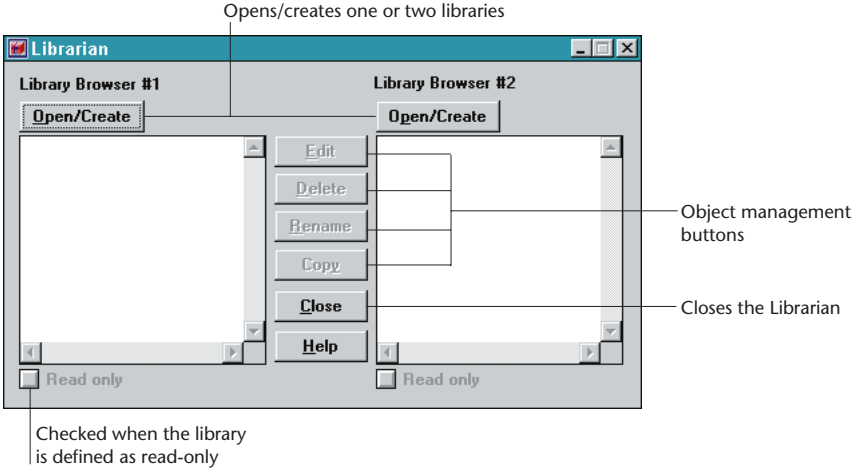
- Open and create object libraries.
- Manage objects within libraries.
- Copy objects from one library to another.
- Add a library object to the Object Layout.

For information about the Object Editor, see Chapter 9: “The Object Editor.”

THE LIBRARIAN

To open the Librarian from the Application Manager:

- Choose File|Librarian.



You can display two libraries simultaneously. You open (or create) object libraries using the Open/Create buttons and then manage the library objects using the Object Management buttons.

WORKING WITH LIBRARIES

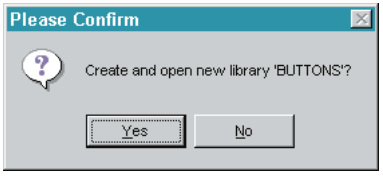
Creating a Library

Library objects are stored in library files (*.rld*). A good practice is to create libraries according to object type or class, or by application name.

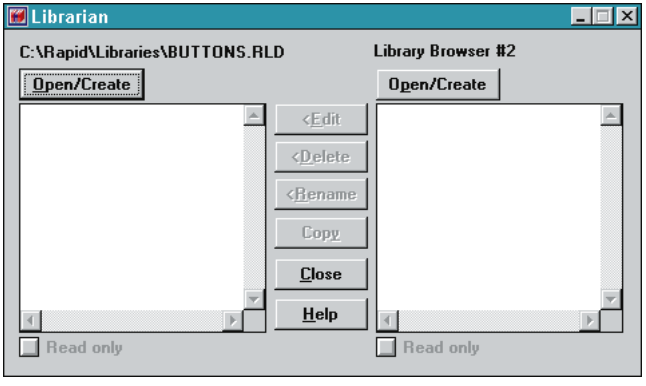
To create a library:

1. In the Librarian, click one of the Open/Create buttons.
2. In the Open/Create Library dialog box, select the destination for the library file (*.rld*), and type a file name for the library. (You may want to make a special folder for library files.)

3. Click OK; a confirmation dialog box opens in which you confirm the library name you typed:



If you choose No, the create library operation is canceled. If you choose Yes, the library is created and is opened in the Librarian.



❖ *NOTE: To add objects to the library, see Saving an Object to a Library on p. 8–5.*

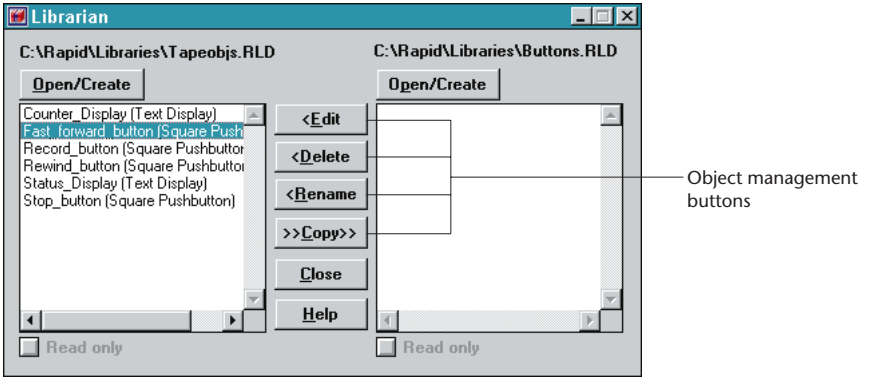
To open an existing library:

1. Click the Open/Create button.
2. Select a library, then click OK. The library is opened in the Librarian and its objects are displayed.

❖ *NOTE: When you open a new application or close the Object Layout, the Librarian is cleared.*

Managing Library Objects

All objects contained in a library are listed in the Librarian. The illustration below shows the Librarian with two open libraries. One of the libraries contains objects, the other does not:



When two libraries are open, the arrows on the object management buttons point towards the active library, that is, the list in which an object is selected or, in the case of the Copy button, the list into which the selected object will be copied. The object management buttons are only enabled when a library object is selected. The buttons are described in the following table:

CLICK	TO
Edit	Edit the selected object in the Object Editor.
Delete	Delete the selected object from the library. You are prompted to confirm the deletion.
Rename	Open a dialog box in which you edit the object name.
Copy	Copy the selected object to the other library. This button is only enabled when two libraries are open.

USING LIBRARY OBJECTS

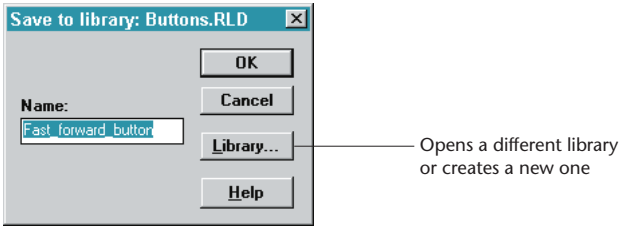
You can save an object to a library from either the Object Editor or the Object Layout. Once an object is stored in a library, you can add it to the Object Layout of any application.

Saving an Object to a Library

You can save any graphic object to a new or existing library.

To save an object in the Object Layout to a library:

1. Select the object on the layout.
2. Choose Objects|Save to Library; if a library has already been opened in the Librarian, the Save to Library dialog box opens:



   If no library is currently open, the Open/Create dialog box opens for selecting or creating a library.

3. The name that appears is the object's name in the application. You can save it under that name or rename it. Library object names do not have the limitations of application object names.

❖ *NOTE: If an object of that name already exists in the library, you are prompted to confirm overwriting the existing object.*

To save an object in the Object Editor to a library:

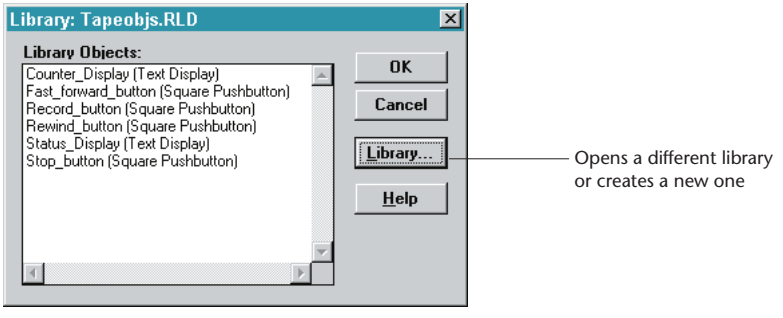
- Refer to the section "Saving New Objects" on p. 9–5.

Adding a Library Object to the Object Layout



To add a library object to the Object layout:

1. Choose Objects|Add from Library, or click the Library button in the Object Palette; the Open/Create Library dialog box opens for selecting a library.
2. Select a library from the list, then click OK; the Library dialog box opens listing all of the objects stored in the selected library:



3. Select a library object, then click OK; the cursor changes to a crosshairs, allowing you to position the object on the layout.

❖ *NOTE: Once you have opened a library, each time you choose Objects|Add from Library, the Library dialog box opens listing that library's objects.*

CHAPTER 9

The Object Editor

The Object Editor is used to construct and edit customized objects of the following classes: indicator, lamp, potentiometer, pushbutton, switch, tank and valve. These objects can be constructed from scratch or be based on existing Rapid objects. You can also use the Object Editor to create completely new bitmaps.

- For all the object types listed above, you can define the appearance for each of its possible states in the Prototyper. For example, you design two bitmaps for a pushbutton—one for its “in” state and one for its “out” state.
- For objects such as switches and pushbuttons, you can designate the active areas. For example, you can specify the active area for each position of a switch.
- For objects that have a pointer, you can change the pointer’s appearance and position.

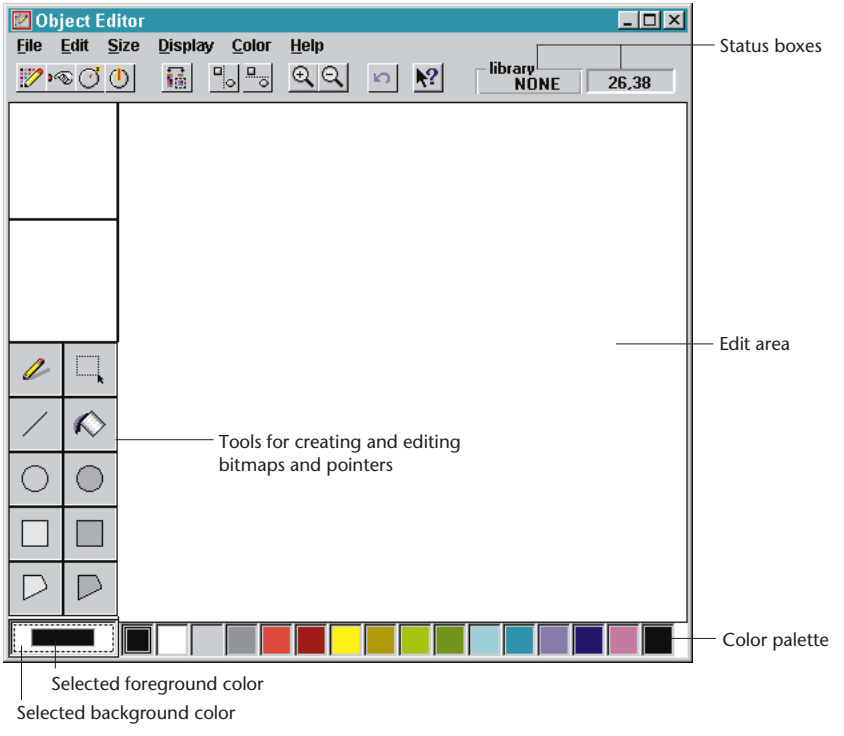
THE OBJECT EDITOR WINDOW



To open the Object Editor:

- Choose Tools|Object Editor in the Application Manager, or click the Object Editor button.

When you first open the Object Editor, its window is set up as shown below:

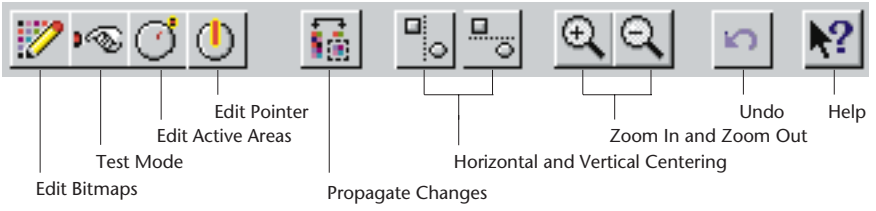


Status boxes

The first status box shows the library in which the object is found. Since there is no object present in the Object Editor shown above, “NONE” appears. The second status box shows the cursor coordinates. In Test mode, the word “TEST” appears.

Toolbar

The buttons on the Object Editor's toolbar are:



The tools are described later in the chapter.

❖ *NOTE: You can open up to ten Object Editor windows at a time.*

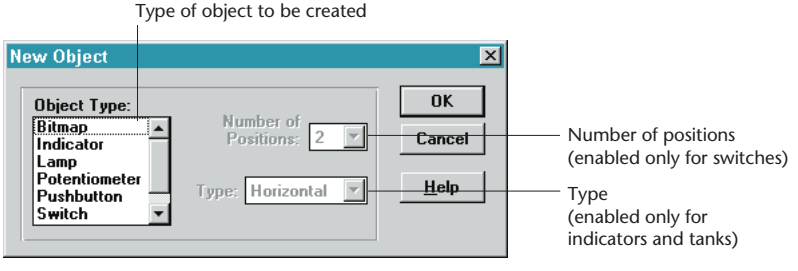
CREATING NEW OBJECTS

You can create new, customized objects of the following classes: indicator, lamp, potentiometer, pushbutton, switch and tank.

To create a new object:


Ctrl+N

1. Choose File|New Object to open the New Object dialog box:

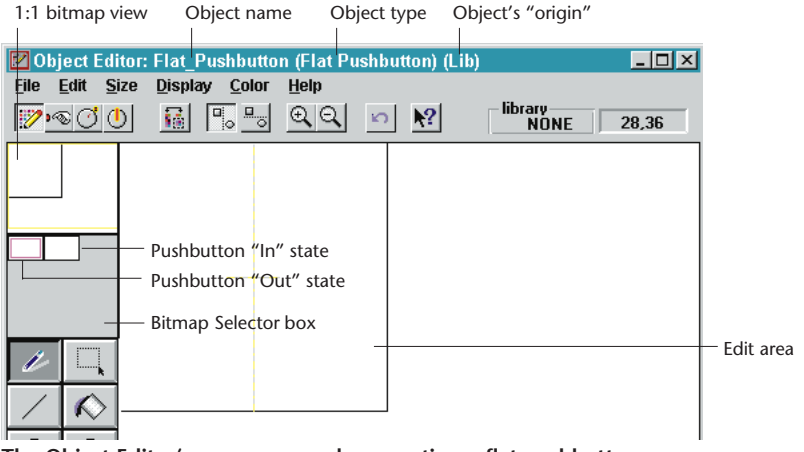


2. Select an object type.

   If you selected an indicator, specify the type: horizontal, rotary or vertical. If you selected a tank, specify the type: horizontal or vertical.

   If you selected a switch, specify the number of positions: 2–12.

3. Click OK. In the illustration below, a pushbutton has been selected:



The Object Editor's appearance when creating a flat pushbutton

Title bar

In the title bar you can see the:

- Object's name (for a new object, Rapid gives a default name)
- Object type
- Object's "origin." If you are editing an application object, the origin is given as "OL" (Object Layout). If you are editing a new object, or an object from a library, the origin is given as "Lib."

1:1 bitmap view

This view displays the bitmap being edited. It is updated as changes are made to the bitmap in the edit area.

Bitmap selector box

The number of empty boxes in the Bitmap Selector box is determined by the object type you selected. For example, if you are creating a lamp, there are two bitmaps, one for its On state and one for its Off state. If you are creating a switch with six positions, there are six bitmaps.

The selected bitmap is highlighted by a magenta frame. This bitmap appears in the 1:1 bitmap view and in the edit area. To edit one of the object's other bitmaps, click it in the selector box.

Edit area

You draw and edit the selected bitmap in this area. Instructions for using the editing modes, commands and toolbox appear later in this chapter.

Saving New Objects

After you have created a new object, you can save it in a library.

To save a new object:

1. Choose File|Save to Library. If no library is currently open, the Open/Create Library dialog box opens. Select or create a library, then click OK.
2. The Save to Library dialog box opens. Either accept the default name or overtype it. Click OK to add the object to the library.

❖ *NOTE: A new switch is saved as a "custom switch."*

EDITING EXISTING OBJECTS

You can edit existing objects from either the Object Layout or a library. Objects that appear on the layout are referred to in the Object Editor as application objects. Objects stored in libraries are referred to as library objects.

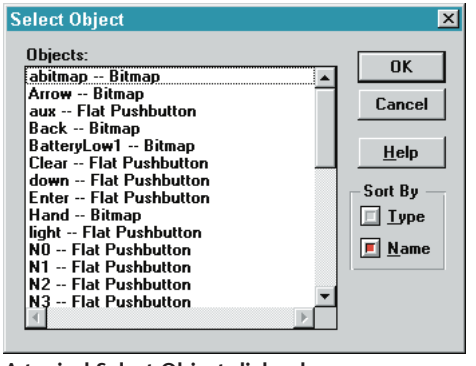
Editing Application Objects

There are two methods for selecting an application object to edit: directly in the Object Editor or from the Object Layout.

To select an application object in the Object Editor:


Ctrl+O

1. Choose File|Open Application Object to see a list of the application's graphic objects.



A typical Select Object dialog box

2. The application objects appear in alphabetical order. Select an object and click OK. If you would like to see the application objects according to object type, click the Type Sort radio button.
3. The selected object appears in the Object Editor.

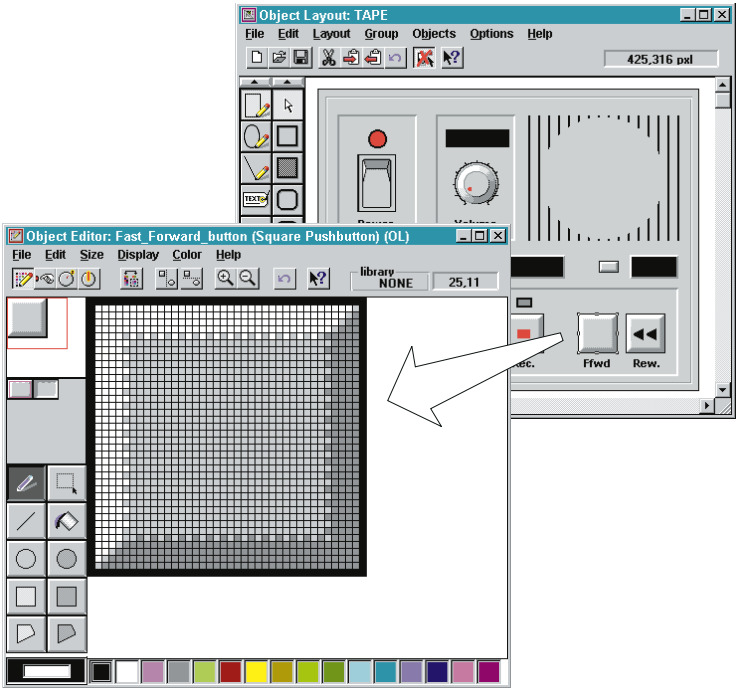
   The title bar displays the object's name, type and origin (i.e., OL—from the Object Layout).

To select an application object from the Object Layout:

1. Select an object on the layout.
2. Choose Objects|Edit Object (or Edit Object in the Object Editor's popup menu) to open the Object Editor with the selected object displayed.



Ctrl+E



An object brought into the Object Editor from the Object Layout

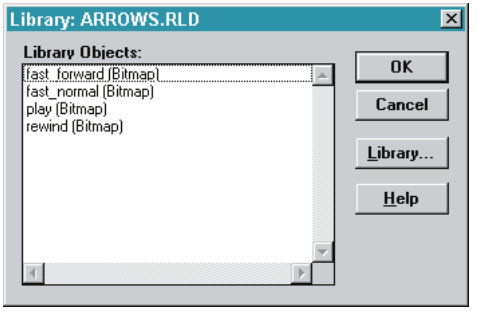
❖ *NOTE: If you edit a dynamic primitive object, the object will lose its color properties (lineColor and fillColor) when you save it.*

Editing Library Objects

There are two methods for selecting a library object to edit: directly in the Object Editor or from the Librarian.

To select a library object in the Object Editor:

1. Choose File|Open Library Object; the Open/Create Library dialog box opens.
2. Select a library from the list, then click OK; the Library dialog box opens listing all the objects stored in the selected library:



A sample list of library objects

3. Select a library object, then click OK.

   If you want to select an object from a different library, click the Library button to reopen the Open/Create Library dialog box.

4. The selected object appears in the Object Editor.

   The title bar presents the object's name, type and origin (i.e., Lib—from a library).

❖ *NOTE: Once you have opened a library, each time you choose File|Open Library Object, the Library dialog box opens listing that library's objects.*

To select a library object from the Librarian:

(See Chapter 8: "Object Libraries" for information about libraries.)

1. Open a library.
2. Select an object from the library, then click the Edit button. The Object Editor opens with the selected library object displayed.

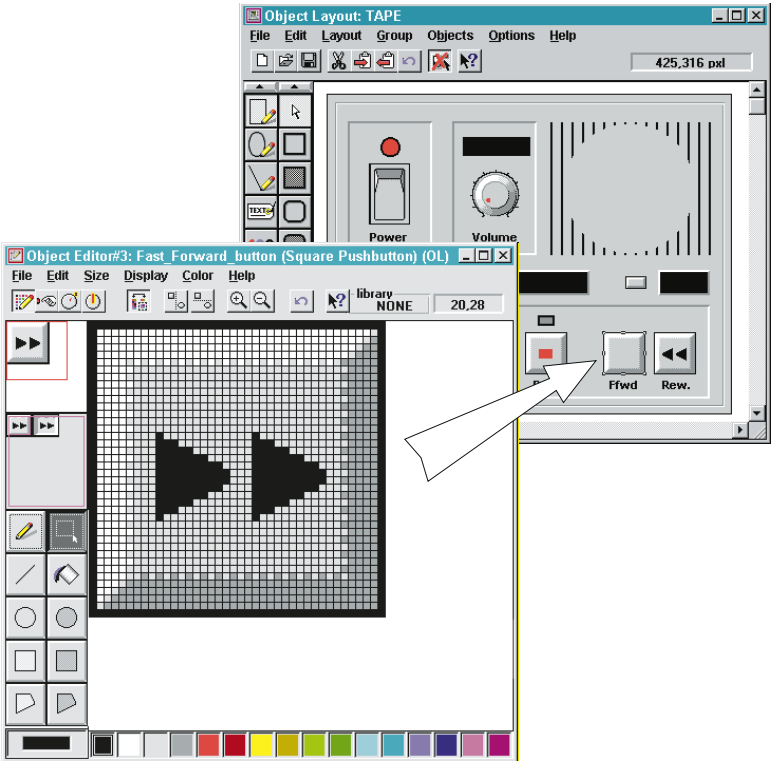
Saving Edited Objects

Application objects can be saved to both the Object Layout and the library. Library objects can only be saved to a library. (To add a library object to the Object Layout, see Chapter 8: "Object Libraries.")

To save an application object to the Object Layout:



- Choose File|Save to Application. The edited object replaces the original object in the Object Layout without closing the Object Editor window.



To save an object to a library:

- See “Saving New Objects” on p. 9–5.

When an edited object is saved, the resulting object behaves like an imported bitmap in certain ways:

- The object’s colors can no longer be edited in the Object Layout (using the Edit|Colors command), although you can still use the Colors Edit dialog box to make one of the object’s colors transparent.
- Resizing the object affects the resolution of the object’s bitmap. If you resize an object once, there usually is little change, but if you resize it several times, the effect can be more noticeable.

❖ *NOTE: Since an object’s bitmap is affected by the resizing operation, it is recommended that you resize objects before you edit them in the Object Editor.*

WORKING IN THE OBJECT EDITOR

The work you perform in the Object Editor affects different aspects of an object: its bitmaps and active areas or pointer. Each of these aspects is edited in a different editing mode. The editing modes are:

MODE	FUNCTION
Bitmap Editing	For creating and editing bitmaps.
Active Areas Editing	For redefining the active areas of objects that have active areas (e.g., rotary switch).
Pointer Editing	For editing the pointer on objects that require a pointer (e.g., indicators).

There is also an additional mode, **Test**, that is used for running a visual test on objects that are mouse-operated (e.g., pushbuttons). Test mode only applies to objects which have active areas.

The modes are described later in this chapter.

Object Editor options

There are various options that allow you to:

- Resize the object's bitmaps.
- Enable a pixel grid.
- Set the line width.
- Enable vertical and/or horizontal centering.
- Propagate changes (that is, for objects that have more than one bitmap, to include changes in all bitmaps).
- Zoom in/zoom out.
- Change colors using the color palette.
- Restore the bitmap/active areas/pointer to its pre-editing state.
- Undo operations.

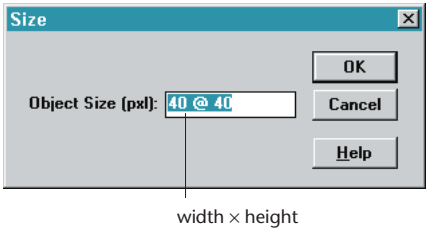
Changing the Object's Size

When you resize an object's bitmaps, you change its actual size (as opposed to zooming in or out to view the object in greater or lesser detail).

This option is only available in bitmap-editing mode.

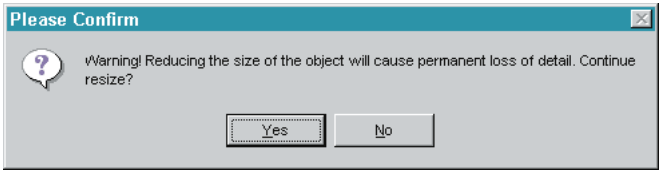
To resize an object's bitmap(s):

1. Choose Size|Set Size to open the Size dialog box:



2. Change the width and/or height of the object. If you enter only one number, Rapid assumes square dimensions. If there is more than one bitmap, the size of all the bitmaps will change.
3. Click OK.

   If you decreased the object's size, the following warning appears:



   Choose Yes to decrease the object's size. Remember that once you reduce the object, the lost detail cannot be recovered (unless you use the Undo command immediately). Choose No to cancel the resize operation.

Using the Pixel Grid

The Object Editor has a visible pixel grid that can be superimposed over the edit area. The grid is useful when you are pixel editing or aligning graphic elements.

There are two type of pixel grids: normal (black on white) and inverse (white on black). The inverse grid is particularly useful when you are editing dark-colored areas where the normal pixel grid is difficult or impossible to see.

This option is available in all modes.

To select a normal grid:

- Choose Display|Grid|Normal.

To select an inverse grid:

- Choose Display|Grid|Inverse.

To deselect the grid:

- Choose Display|Grid|Off.

Setting the Line Width

The line width determines the thickness of lines, rectangles and circles that you draw. The line width thickness range is 1–5 pixels.

This option is available in bitmap-editing and pointer-editing modes.

To set the line width

- Choose Display|Line Width|1, 2, 3, 4 or 5.

Centering New Graphic Elements

You can use the centering commands to automatically center new graphic elements in the vertical and/or horizontal plane.

These options are available in bitmap-editing and pointer-editing modes.

To activate vertical centering:

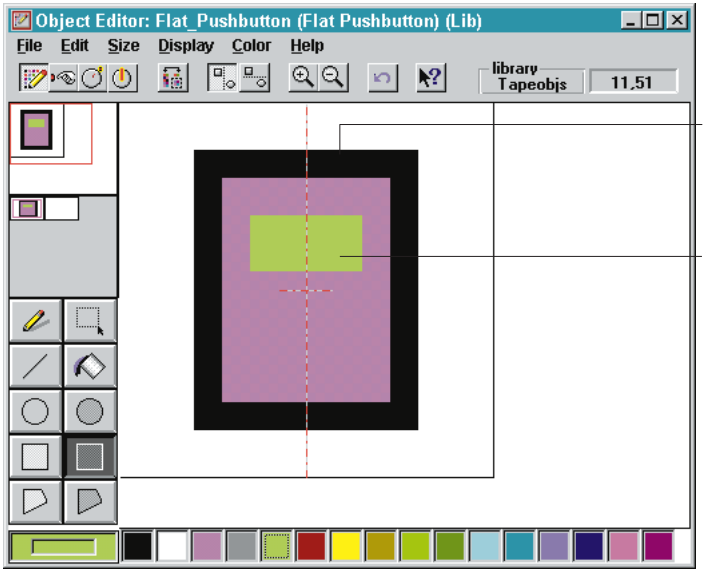


- Choose Display|Centering|Vertical, or click the Vertical Centering button. A y-axis appears.

To activate horizontal centering:



- Choose Display|Centering|Horizontal, or click the Horizontal Centering button. An x-axis appears.



A pushbutton created using the centering options

Propagating Changes

If you want the changes you make in the selected bitmap to appear in all the object's bitmaps, use the Propagate Changes command. It can save a great deal of time and promote consistency when you are working on graphic elements that are common to all the bitmaps.

This option is available in all modes, but only for objects that have more than one bitmap.

To activate Propagate Changes:



- Choose Display|Propagate Changes, or click the Propagate Changes button.

All the bitmaps displayed in the Bitmap Selector box are enclosed in a magenta frame.

❖ *NOTE: Make sure that Propagate Changes is off when you are making changes specific to a single bitmap.*

Zooming

If your bitmap is larger than the Object Editor window, you can display the entire bitmap by using the Zoom Out command. When performing intricate work, you can magnify the bitmap by using the Zoom In command. This option is available in all modes.

To magnify a bitmap:



- Choose Display|Zoom|In, or click the Zoom In button.

To reduce the bitmap view:



- Choose Display|Zoom|Out, or click the Zoom Out button.

❖ *NOTE: The Zoom in/Zoom out range is 15. If you are using a grid and zoom out to the lowest two views, the grid is deactivated.*

Selecting Colors

The Object Editor has a basic palette of sixteen colors. If your display supports 256 or more colors, you can open the appropriate dialog box to select a non-basic color (see "Customizing Colors," below).

❖ *NOTE: If your display is in 16 color mode, none of the commands in the Colors menu (except the Switch Fore/Back Colors command) are available.*

Selecting colors using the palette

To change the foreground color:

- Click a color on the palette.

To change the background color:

- Right-click a color on the palette.

To switch the foreground or background colors:

- Either
  Choose Color|Switch Fore/Back Colors.

  Or:
  Shift-click the foreground or background color.

Selecting colors directly from a bitmap

Instead of searching the palette, you can select colors directly from a bitmap. This procedure is useful for selecting colors that were previously used.

To select the colors directly from a bitmap:

1. Position the cursor over a pixel whose color you want to select.
2. For the foreground color, Shift-click with the left mouse button. For the background color, Ctrl-click with the left mouse button.

Customizing Colors

The Object Editor has various features that allow you to customize its basic colors. You can:

- Customize an individual color within the basic palette.
- Customize the foreground or background color.

To customize an individual color within the palette:

1. Click a color, then choose Color|Edit Color.

   SHORTCUT:
   Double-click a color.



2. (If your display is in 256-color mode, the extended palette opens; if your display supports more than 256 colors, the Color dialog box opens.)

   Select a color and close the dialog box.

❖ *NOTE: For instructions on using the Color dialog box, see "Editing Objects' Colors, Choosing from a wider range of colors" in Chapter 2.*

To customize the foreground or background color:

1. Double-click either the foreground or background color or choose the Color|Edit Foreground Color or Edit Background Color command.
2. Select a color and close the dialog box.

Restoring the default color palette

To restore the default color palette:

- Choose Color|Restore Default Colors.

Restoring the Bitmap/Active Areas/Pointer

The restore operation that occurs depends on the current editing mode.

Restore bitmap(s)

Returns the bitmap to its original state. This command is enabled when editing existing objects only. If Propagate Changes is enabled when the restore operation takes place, all the bitmaps are restored.

If you change the size of an object (via the Set Size command) and then choose Restore Bitmaps, you will receive a message informing you that the

object will be restored to its original state and size. Unless previously saved, changes made in any of the editing modes will be undone. You can choose to continue the restore or cancel it.

Restore active areas

Returns the active areas to their original state. If Propagate Changes is enabled when the restore active areas operation takes place, all the bitmap's active areas are restored.

Restore pointer

Returns the pointer to its original state.

Undoing

The Undo command allows you to delete the last-made change from the current bitmap or to cancel the previous operation.

This option is available in all editing modes.

To undo a graphic change or an editing operation:



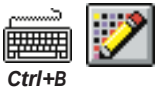
- Choose Edit|Undo, or click the Undo button.

BITMAP EDITING

The Object Editor's major editing mode is bitmap-editing. In this mode you determine what an object looks like in each of its states; for example, a lamp when it is on and off, a pushbutton when it is in and out.

If an object has more than one bitmap (e.g., a toggle switch with three positions), all the bitmaps appear in the Bitmap Selector box. The bitmap which is currently being edited appears in the 1:1 view and the edit area, and is highlighted by a magenta frame in the Bitmap Selector box.

To switch to bitmap-editing mode:



- Choose Display|Mode|Edit Bitmaps, or click the Edit Bitmaps button.

To select a bitmap for editing:

- Click it in the Bitmap Selector box.

The Editing Tools

The Object Editor has ten graphic editing tools for creating and modifying the bitmaps and pointers. The sections below describe the tools and how to use them.



Pixel painter

Paints a square area with the selected foreground or background color. The size of the painted area depends on the current line width.

To use the pixel painter tool:

- To paint a pixel in the foreground color, click it.
- To paint a pixel in the background color, right-click it.

❖ *NOTE: If the line width is larger than 1×1 pixel, the pixel you clicked on becomes the center of the painted area.*



Line

Draws a line in the selected foreground color with the current drawing width.



Circle

Draws an empty circle. Its outline is in the selected foreground color with the current drawing width.



Filled circle

Draws a filled circle. The outline is the selected background color with the current line width, and the fill is the selected foreground color.



Frame

Draws an empty rectangle. Its outline is in the selected foreground color with the current drawing width.



Filled frame

Draws a filled rectangle. The outline is in the selected background color with the current line width, and the fill is the selected foreground color.

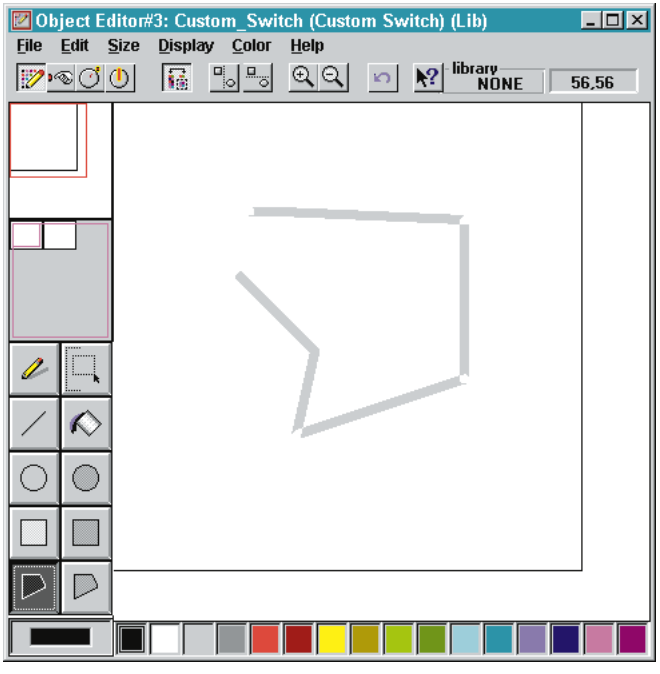


Polygon

Draws an empty polygon of any shape. Its outline is in the selected foreground color with the current line width.

1. Position the crosshairs where you want to anchor one of the polygon's corners and click.
2. Position the crosshairs over the next polygon corner and click again. A gray segment is automatically drawn between the two points.

❖ *NOTE: You can also click and drag in step 1, in which case a segment tracks the movement of the mouse as you drag to the second corner.*



3. Repeat step 2 until you close the polygon and it is drawn on the bitmap.

❖ *NOTES: If you designate a corner which is within four pixels of a polygon segment already drawn, Rapid automatically joins the two points and closes the polygon.*

When clicking-and-dragging, the start point of the next segment must be the end point of the previous segment. Thus, no matter where you click-and-drag the mouse in step 1, Rapid automatically references the segment starting point to the pixel where you last released the mouse button.



Filled polygon

Draws a filled polygon of any shape. The outline is in the selected background color with the current drawing width, and the fill is the selected foreground color.



Fill

Paints an enclosed area using the selected foreground or background color.

To use the Fill tool:

- To paint an area in the foreground color, position the fill cursor within the enclosed area and click.
- To paint an area in the background color, position the fill cursor within the enclosed area and right-click.



Marquee

Marks sections of the bitmap to be cut or copied using the Cut and Copy commands in the Edit menu. You can also mark areas to be moved.

To mark an area:

1. Position the crosshairs in a corner of the area to be marked.
2. Click and drag the cursor until the rectangular frame encloses the area you want to mark.

To move a marked area:

1. After an area has been marked, move the crosshairs away from the marked area; the hand cursor appears.
2. Place the hand over the marked area; click and drag the hand, moving the marked area with it.

3. When you are satisfied with the new position of the marked area, release the mouse button.
4. Move the hand out of the marked area and click; the area becomes "unmarked" and becomes part of the bitmap. The space vacated by the marked area is painted in the background color.

Cutting, Copying and Pasting Sections of a Bitmap

You can cut or copy sections of the bitmap to the Windows Clipboard. These operations are useful, for example, if you prefer to work on the bitmap in another Windows graphics program. When you have finished editing, you copy it back to the Clipboard, then paste it into the Object Editor.

❖ *NOTE: If Propagate Changes is enabled, the cut, copy and paste operations will affect all the object's bitmaps.*

Cutting

Cutting a section from a bitmap removes it from the Object Editor and places it on the Clipboard.

To cut a piece of the bitmap:


Ctrl+X

- Mark an area using the marquee tool, then choose Edit|Cut.

Copying

Copying a section from a bitmap copies it to the Clipboard without removing it from the Object Editor.

To copy a piece of the bitmap:


Ctrl+C

- Mark an area using the marquee tool, then choose Edit|Copy.

Pasting

You can paste bitmap contents of the Clipboard into the current bitmap.

To paste a bitmap in the Object Editor:


Ctrl+V

1. Choose Edit|Paste. The bitmap appears in the upper left corner of the edit area for the current bitmap.

2. Click and drag the pasted bitmap anywhere in the edit area. Release the mouse button to fix the position.
3. Click once more to end the paste operation.

Clearing and Refreshing the Bitmap

Clearing the bitmap

You can clear the currently-selected bitmap to create a new work area.

To clear the bitmap:

- Click Edit|Clear Bitmap.

❖ *NOTE: Make sure anything you have been working on has been saved before using this command.*

Refreshing the bitmap display

You can rebuild and redraw the window and remove any accumulated smears from the current bitmap.

To refresh the bitmap display:



- Click Edit|Refresh Display.

Manipulating Columns and Rows

The Size menu contains commands that allow you to copy, insert and delete rows or columns from the bitmaps. When using these commands, you should activate Display|Propagate Changes, otherwise, the row/column change will be made only to the selected bitmap, and blank rows/columns will be inserted in the remaining bitmaps.

❖ *NOTE: These commands do not reposition active areas. You must go to active area-editing mode to check active area positions after using these commands.*

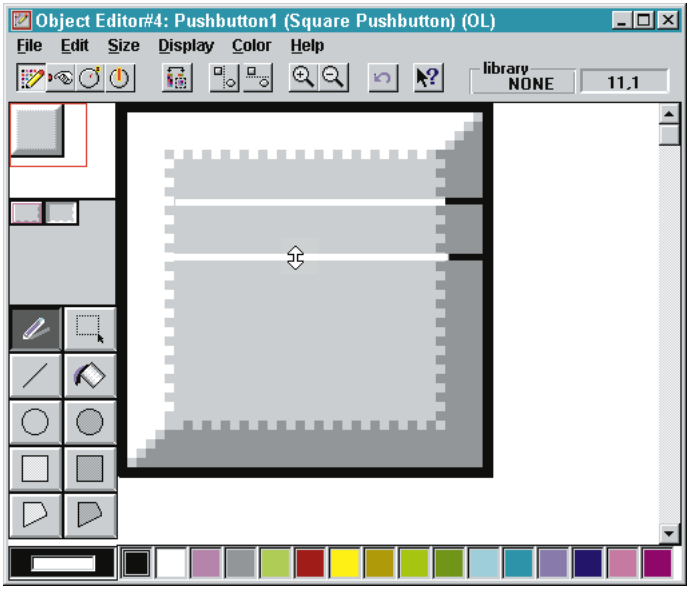
Copying rows/columns

To copy a row or column:

1. Choose Size|Row/Column|Copy.
2. The mouse pointer appears as a two-headed arrow. For rows, the arrow is accompanied by a horizontal line; for columns, the arrow is accompanied by a vertical line.

   For rows:

   Position the arrow+line in the highest or lowest row you want to copy. Click and drag downwards/upwards until the rows are enclosed within a rectangle.



   For columns:

   Position the arrow+line in the left-most or right-most column you want to copy. Click and drag to the right/left until the columns are enclosed within a rectangle.

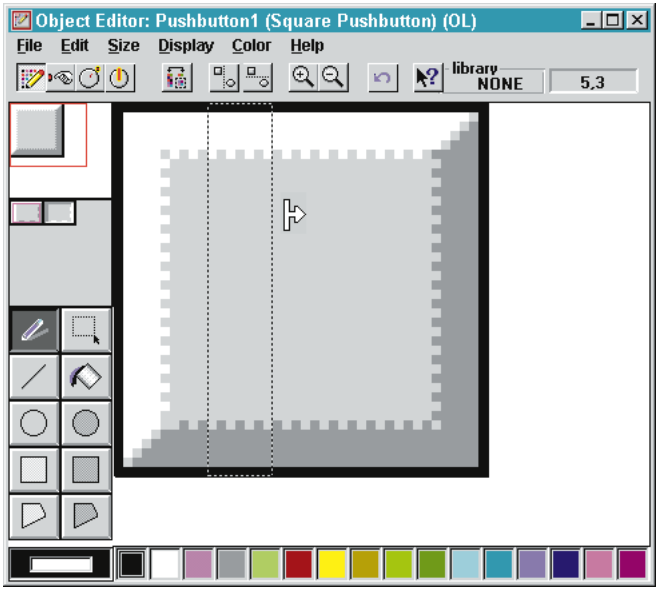
3. Release the mouse button.

   For rows:

   The two-headed arrow changes to a downwards pointing arrow. Position the arrow at the point you want the copy and click. The copy appears below the marked area.

   For columns:

   The two-headed arrow changes to a right-pointing arrow (see the following illustration). Position the arrow at the point you want the copy and click. The copy appears to the right of the marked area.



❖ *NOTE: The bitmap width is incremented by one each time you duplicate a column, and its height is incremented by one each time you duplicate a row.*

Inserting rows/columns

To insert a row/column:

1. Choose Size|Row/Column|Insert.
2. A dialog box appears for entering the number of rows/columns (1-100) you want to insert into the bitmap. Enter a number and click OK. The mouse pointer becomes a downwards pointing arrow (for rows)/right pointing arrow (for columns). Move the arrow to the desired insertion point and click.

Deleting rows/columns

To delete a row or column:

1. Choose Size|Row/Column|Delete.
2. Click on the row or column that you want to delete. If you are deleting more than one row/column:

   For rows:

   Position the arrow+horizontal line in the highest or lowest row you want to delete. Click and drag downwards/upwards until the rows are enclosed within a rectangle.

   For columns:

   Position the arrow+vertical line in the left or right-most column you want to delete. Click and drag to the right/left until the columns are enclosed within a rectangle.
3. Release the mouse button.

❖ *NOTE: The bitmap width is decreased by one each time you delete a column, and the height is decreased by one each time you delete a row.*

Showing the Pointer

- Choose Display|Show Pointer to display the pointer in the 1:1 view.

ACTIVE AREAS EDITING

Many objects have active areas. The table below describes each object's active areas:

OBJECT	ACTIVE AREAS
Switches (general)	One for each hot spot (switch positions).
Stepper switch	One for each hot spot (arrow buttons) and one containing the object's text display.
Pushbutton	One (the hot spot).
Tank	One (the area that fills and empties.)
Valves	One for each valve position

When you are designing any of these objects, the Object Editor provides default active areas. You can redefine any or all of these areas.

To switch to active area editing mode:



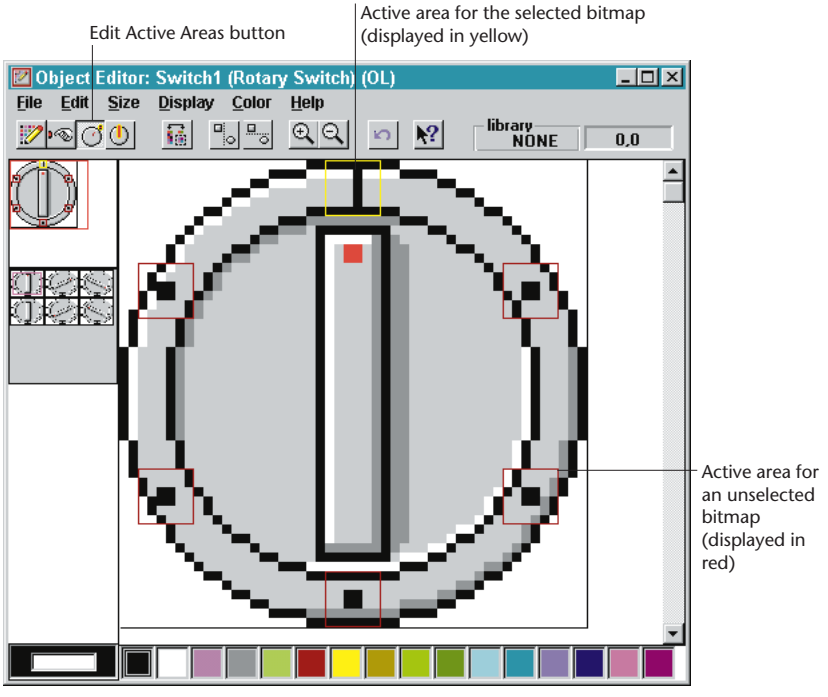
- Choose Display|Mode|Edit Active Areas, or click the Edit Active Areas button.

The active areas are displayed in red; the current bitmap's active area is displayed in yellow.

To edit an active area:

1. Position the crosshairs at the point where you want to anchor one corner of the active area frame.
2. Click and drag the gray frame (within which the active area will be drawn) that appears on the screen.

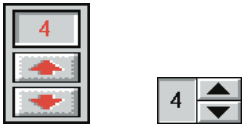
❖ *NOTE: Each time you redraw, you overwrite the previously-drawn active area.*



3. For each bitmap, select it in the Bitmap Selector box and repeat steps 1–2.

A special case: the stepper switch

As well as active areas for each of the buttons, the stepper switch has one in which its number is displayed. You can change the size and position of the text display area. Shown below are two examples of edited stepper switches:



POINTER EDITING

Potentiometers and indicators require a pointer to indicate their current setting or reading. When you create these objects in the Object Editor, their pointers are automatically created as well. The pointers can be edited in pointer-editing mode.

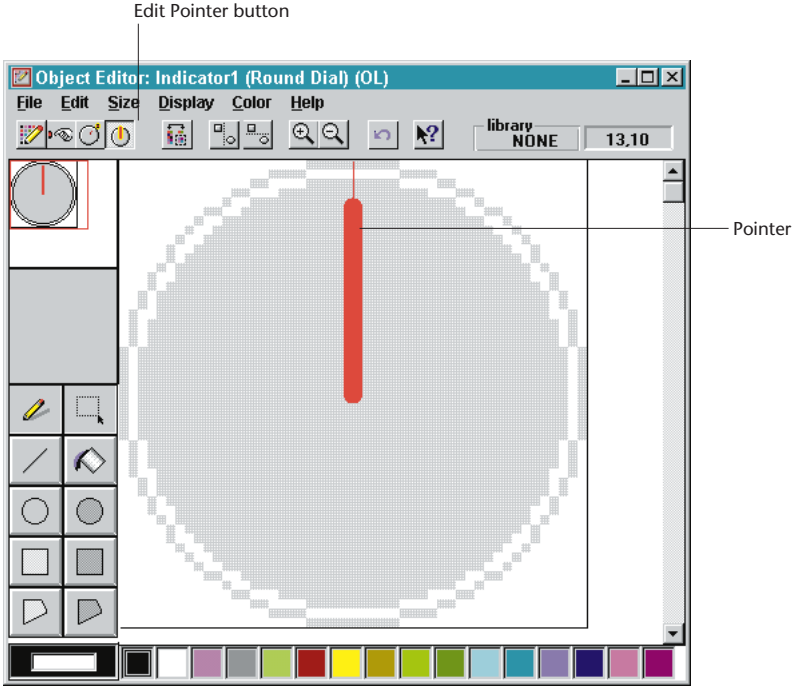
❖ *NOTE: Pointer-editing mode is disabled for objects that do not have a pointer.*

To switch to pointer-edit mode:



- Choose Display|Mode|Edit Pointer, or click the Edit Pointer button.

The bitmap in the edit area becomes half-toned, and an overlaid red line indicates the pointer's axis and point of reference.



❖ *HINT: Before editing the pointer, activate automatic horizontal and/or vertical centering to accurately center the pointer. When activated, the newly-generated pointer is positioned over the center line of the object's vertical and/or horizontal planes.*

To edit the pointer:

1. Select a drawing tool from the toolbox. The pointer must be a single graphic entity, i.e., a line, a circle or a polygon. The pixel painter is disabled in pointer-editing mode.
2. Position the crosshairs at the desired pointer location. Click and drag the corner of the rectangular frame to the desired size.

   The pointer's outline is drawn in the foreground color with the current line width. If filled, the outline is the background color and the fill is the foreground color.

❖ *NOTE: Each time you redraw, you overwrite the previously-drawn pointer.*

To change the pointer's color(s):

1. Select a different color(s) from the palette, then select the Fill tool.
2. Click anywhere within the edit area. The pointer is repainted with the new color(s).

To change the pointer's line width:

- Choose Display|Line Width|1, 2, 3, 4 or 5.

To return to bitmap-editing mode:

- Switch to bitmap-editing mode. The pointer is only visible on the bitmap if you choose Display|Show Pointer.

TEST MODE

In test mode, you can run a simple visual test on objects that are mouse-operated. For example, you can click a switch to observe it change position.

To test an object:

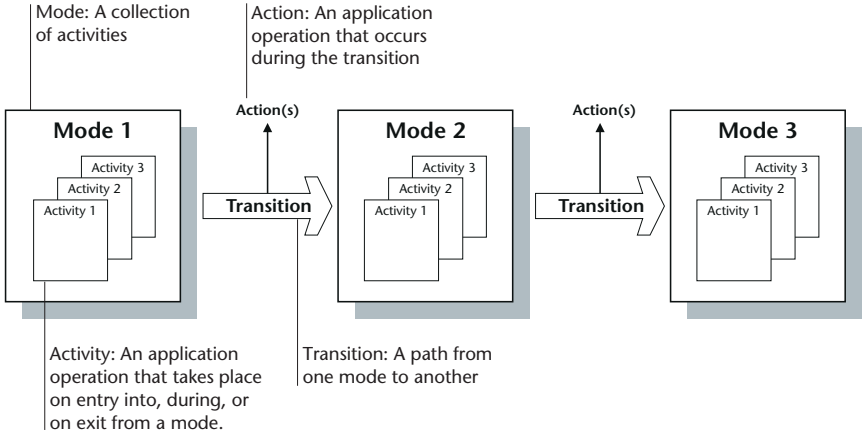


1. Choose Display|Mode|Test, or click the Test button.
2. When the cursor is positioned over an active area, a hand appears. Click the active area to test it.

CHAPTER 10

Rapid Logic: The Basics

The diagram below summarizes the major elements that make up a Rapid application's logic.



Logic flow in a Rapid application

This chapter explains in detail the meaning of modes, transitions and all the other logic elements that govern the behavior of a Rapid application. It also explains the different options that are available, and their significance for your application's behavior.

MODES

What is a Mode?

A Rapid application is a description of a complete system, such as a mobile telephone or an aircraft hydraulic system. The system's overall behavior is broken down into individual **modes**. For example, a tape recorder can be *on* or *off*. These are two distinct modes.

A tape recorder actually has many other modes besides *on* and *off*. While the recorder is on, there are several possible modes that you could describe: play, record, display counter, etc. The tape recorder, as a system, can be in one or several of these modes at any given time.

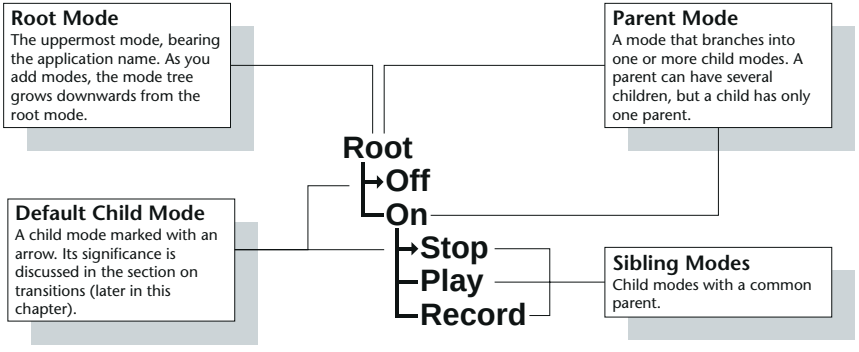
A mode can be as brief as a fraction of a second, or extend to several seconds, minutes or hours. Each mode, no matter how brief, is a distinct unit within the system's life cycle. Rapid allows you to depict all the modes of a system, as they occur in real-life.

Each mode is a collection of activities which Rapid performs on the objects when the mode is active. Each activity can take place either upon entry into the mode, upon exit from the mode or for the entire time that the mode is active (see "Activities" on p. 10–13).

As your application moves from mode to mode, sometimes activating one set of modes, sometimes activating several sets of parallel modes, Rapid performs all the relevant activities on the objects.

Hierarchy

When developing a Rapid application, you have to represent its behavior as a hierarchical arrangement of modes. The modes are linked as shown below:

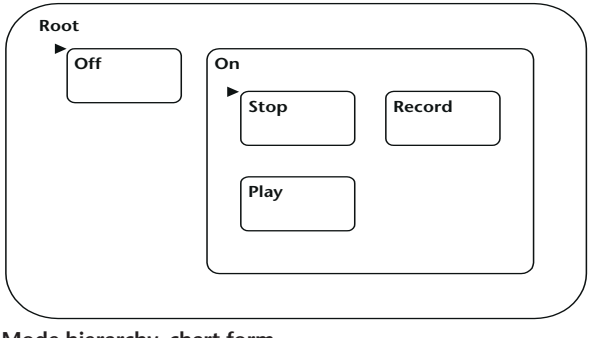


Mode hierarchy–tree form

The Rapid mode hierarchy is based on a parent-child relationship, with the arrangement of the tree branches depicting the relationship of one mode to another. The root mode breaks down into child modes, one of which (On) is itself a parent to three children, referred to as **sibling modes**.

This hierarchical organization means that when a transition is triggered from Off to On, Rapid deactivates Off and its children (if any) and activates On and one or more of its children (if any).

Another way to represent the same hierarchy is in nested chart form. In this format, the hierarchy among modes is shown by nesting child modes within their parent mode.



Mode hierarchy–chart form

Types of Modes

There are two types of modes; when you create a new mode, you must choose its type from one of the two options:

- Exclusive
- Concurrent

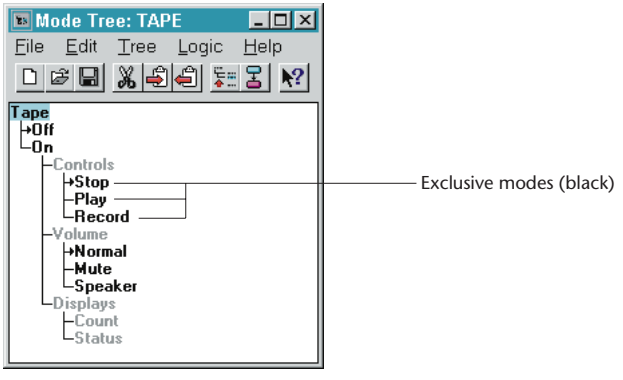
Exclusive modes

If sibling modes are exclusive, only one of them can be active while their parent mode is active. Use exclusive modes for grouping activities that cannot run simultaneously.

A tape recorder, for example, cannot play and record at the same time; a Rapid application of a tape recorder would include these activities within different, exclusive modes.

Example:

The mode tree for a tape recorder application is shown below. The Stop, Play and Record modes, are siblings that have been defined as exclusive modes. Only one of them can be active while their parent mode, Controls, is active.



❖ *NOTE: By definition, exclusive modes can only have exclusive siblings.*

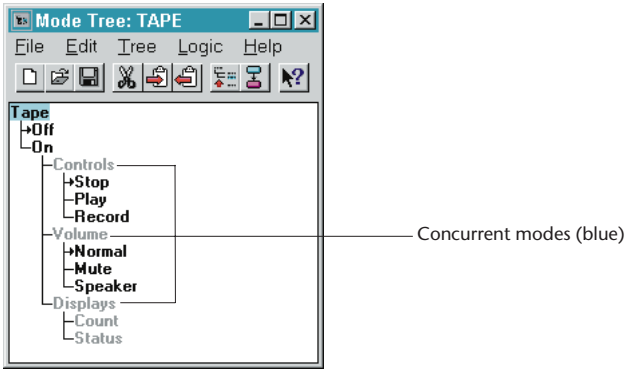
Concurrent modes

If sibling modes are concurrent, then they are all active when their parent is active. Use concurrent modes for parts of the system that are functionally independent of each other and can run simultaneously.

For example, a tape recorder can display the counter and change the volume at the same time; a Rapid application would include those activities within different, concurrent modes.

Example:

The tape recorder application has the modes Controls, Volume and Displays. They are siblings that have been defined as concurrent modes. They are all active while their parent mode, On, is active.



Two points to remember about concurrent modes:

- A concurrent mode can have exclusive children.

  This means that although its children cannot be active at the same time as each other, one of them can be active at the same time as a child of a different concurrent mode. For example, Play cannot be active at the same time as Record, but it can be active at the same time as Mute (they are children of different concurrent modes).

- A concurrent mode can have concurrent children.

  This means that when the mode is active, all of its children are active. For example, when Displays is active, Count and Status are both active. Logically, Displays could contain all the activities that take place in Count and Status and you could dispense with those two modes. Dividing the activities into two separate modes, however, helps you keep your application organized and more intuitive.

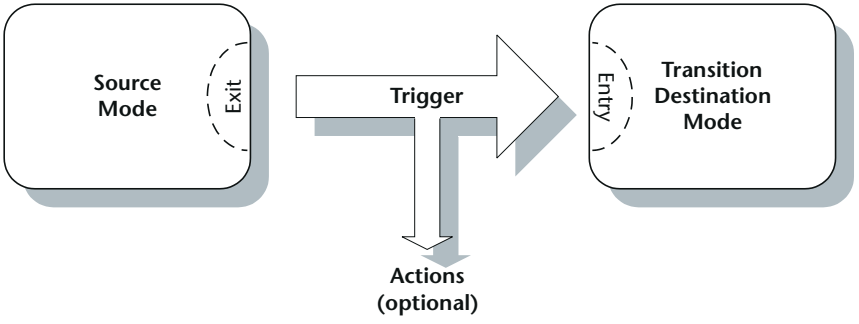
TRANSITIONS

What is a Transition?

An application's behavior is defined by its modes and the transitions between the modes. A transition is comprised of:

- A destination—the mode which Rapid activates when the transition takes place.
- A trigger—a certain condition within the application that initiates the transition.

In addition, a transition can also include one or more actions—operations that take place when the transition is triggered.



Parts of a transition

If a transition destination is a mode with exclusive children, Rapid will activate **one** of the child modes when the transition is triggered. The specific child mode which Rapid activates is determined by the type of transition that you specify when you define the transition destination.

❖ *NOTE: If the destination mode has concurrent children, they will **all** become active. If the destination mode has no children, the transition type is irrelevant.*

Transition Types

There are four types of transitions:

- Default
- History
- Deep History
- Internal

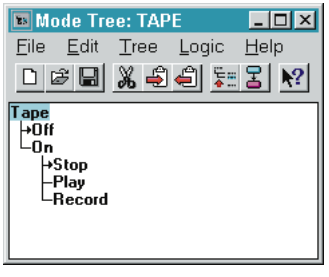
This section discusses the first three types of transitions. (Internal transitions are described on p. 10–10.)

Default transition

A default transition is the most common type of transition. When a default transition is made, the destination's default child mode is activated. Default child modes are marked with an arrow in the mode tree.

Example:

Shown below is the Mode Tree of an earlier version of the tape recorder application. If the transition from Off to On is a default transition, then when On becomes active, Stop (On's default child) also becomes active.



Transitions to concurrent modes

If a mode which has concurrent mode children is made active, all its children become active (the definition of concurrent modes). Similarly, when a concurrent mode becomes active, all its sibling modes become active. A concurrent mode, therefore, cannot be a transition destination. If you want Rapid to activate a concurrent mode, you must make a transition to its parent.

The type of transition (i.e., default, history, or deep history) to the concurrent mode's parent applies to the concurrent mode's **children** (i.e., the concurrent mode is "transparent" as far as the type of transition to its parent is concerned).

Example:

In a later version of the tape recorder application, shown on the next page, the transition from Off to On is a default transition. When the transition is triggered, On becomes active and also Stop and Normal (default child modes of concurrent parents).[1]

History transition

When a history transition is made to a mode, its most recently active child is activated. In other words, Rapid remembers which child was active the last time the parent mode was exited. The next time the transition is made, it makes that mode the active child (even if it is not the default child).

❖ *NOTE: The first time that the parent mode is made active (i.e., before a history has been created) the default child mode becomes active with the parent.*

Example:

Let us say that in our tape recorder application the transition from Off to On is defined as a history transition. Whenever this transition occurs[2], Rapid remembers which of the siblings (Stop, Play or Record) was most recently active and that child becomes active again. (When the tape recorder is off, pressing any of the buttons has no effect.)

This behavior resembles the behavior of a real tape recorder. When you first switch it on, it is stopped (Stop, the default child mode). When you press the play button the recorder plays (Play mode). If you then turn the recorder

1 Displays, Count and Status also become active.

2 Assuming it is not the first time that On becomes active.

off—without first stopping it—and turn it back on again, it will continue to play (i.e., the Play mode is activated, not the default Stop mode).

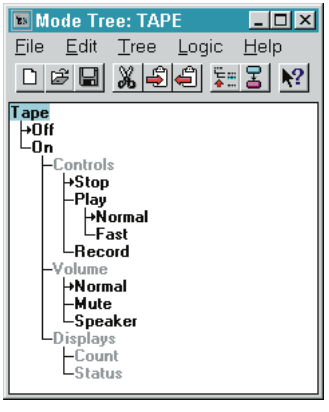
Deep history transition

When a deep history transition is made to a mode with several generations of descendants, the most recently active descendant at each generational level becomes active. In other words, Rapid remembers which child, grandchild, great-grandchild, etc., were active the last time the parent mode was exited and makes those modes active (even if they are not default modes).

❖ *NOTE: The first time that the parent mode is made active (i.e., before a deep history has been created), the default child descendants become active with the parent.*

Example:

Our tape recorder application has two kinds of play modes: normal and fast. They are exclusive child modes of Play, with Normal as the default child.



If the transition from Off to On is a deep history transition, then when this transition occurs, Rapid activates the most recently active mode in all the generations below On. If the most recently active child was, for example, Stop, then the chain ends there because Stop does not have any descendants. (In this case, the deep history entry type is effectively the same as the history entry type.) If, however, the most recently active child was Play, Rapid also activates the most recently active of Play's children.

This means that if the recorder was playing at its fast speed, and you turned it off—without stopping it first—and then on again, it would continue to play fast. If the transition was a history transition and not deep history, the recorder would play, but only at normal speed.

Internal Transitions

Often you will want Rapid to perform operations without exiting a mode, but only under particular conditions, and not continuously. In this case, use an internal transition.

Internal transitions allow execution of actions that are triggered by conditions and/or events, without leaving the mode. A mode can have several internal triggers, each with its own associated action.

Example:

Our tape recorder has a button that resets the tape counter; when you press the button, the counter display resets to zero. In the Rapid application, we do not want to exit the Counter mode when the button is pressed. The solution: add an internal transition to the mode.

The internal transition could be triggered by a condition or an event (see "Triggers," below).

❖ *NOTE: If the condition of an internal transition is true again, Rapid repeats the transition. That is, an internal action is like a conditional mode activity.*

As you build your application, you will find that internal transitions are a very common aspect of a mode's behavior.

Internal transitions or transitions from a mode to itself

An internal transition allows Rapid to execute an action or actions without leaving the mode. Only the action(s) associated with the internal transition trigger are executed. This is quite different from what happens when a transition from a mode **back to itself** is triggered.

If a transition from a mode back to itself is triggered, several things happen:

- The mode and all its descendents are exited and Rapid executes all their exit activities (if any).
- Any actions associated with the transition are executed.
- The mode is then re-entered, along with children and other descendents—depending on the transition type and the mode types of the descendents. When the mode is re-entered, its entry and mode activities (and that of its active descendents) are executed.

The order in which Rapid executes activities and actions is discussed in more detail in Chapter 16: "Rapid Logic: Hints and Tips."

TRIGGERS

The set of circumstances which cause a transition to take place is known as a trigger. A trigger is made up of one, or both, of the following elements:

- Conditions
- Events

Conditions

A condition tests some property of an object and returns a value of true or false. If true, the transition is triggered. In Rapid syntax, typical object conditions would look like the following examples:

& Switch1.position2 is connected

In this case[1], Rapid tests the property *position2* of the object "Switch1;" the condition is true if *position2* is connected, otherwise the condition is false.

& NumberObject > 10

In this case, Rapid tests the *self* property (i.e., the object's value — implicit in the syntax) of NumberObject; the condition is true if NumberObject is greater than 10, otherwise the condition is false.

Conditions on modes

There is a special type of condition that tests the state of a particular mode and returns either true (if the mode is active) or false (if not active), or vice versa, depending on the function used to test the state of the mode. In Rapid syntax, a typical mode condition trigger would look like:

& mode/Off is inactive

Rapid triggers the transition if it finds that the Off mode is inactive.

Events

Rapid generates an event at the moment that some defined change in an object's property occurs. In Rapid syntax, a typical event would look like:

Button1 in &

In this case, Rapid generates the event when the pushbutton goes in.

1 Rapid logic syntax is explained in Chapter 12: "Building Logic".

Multiple and Compound Triggers

A trigger can be any number or combination of conditions and/or events.

Multiple trigger

A multiple trigger is one that is made up of two or more of the same type of triggers, for example, two events or two conditions.

In Rapid syntax, a typical multiple event trigger would look like:

Button1 in ! Timer1 tick &

In this case, Rapid triggers the transition at the moment that the user pushes in the pushbutton object "Button1" **or** when the timer object "Timer1" reaches zero. Note that events are separated by an exclamation mark.

❖ *NOTE: Events **cannot** occur simultaneously.*

A typical multiple condition trigger would look like:

& Integer = 5 and Switch.position2 is connected

Here, Rapid only triggers the transition when the integer object is equal to 5 **and** the switch object is connected to position 2.

❖ *NOTE: Conditions* **can** *be true simultaneously.*

Compound trigger

A compound trigger is made up of a combination of trigger types, for example, one event and one condition.

A typical compound trigger would look like this:

Button1 in & Switch1.position2 is connected

In this case, the transition will be triggered only when "Button1" is pushed in **and** "Switch1" is in position2. Note that events and conditions within a trigger are separated by an ampersand (&).

Refer to the "Object Reference" manual for a description of the various events that objects can generate and the conditions that can be used to test each object's properties. It is important to note, however, that not all objects can generate events.

ACTIVITIES

What is an Activity?

Activities are application operations performed on objects when a specific mode is activated. There are three types of activities, that relate to the time frame in which the activity is performed:

- Entry activity: Occurs when the mode becomes active.
- Exit activity: Occurs when the mode becomes inactive.
- Mode activity: Occurs continuously while the mode is active.

It is important to note that activities always occur, irrespective of the route that Rapid takes to enter the mode, and the route Rapid takes after exiting the mode. That is, an exit activity takes place whenever a mode is exited—regardless of which transition was triggered to cause the exit from that mode.

❖ *NOTE: When the application is started in the Prototyper, Rapid executes the entry activities of all modes that are initially active. When the application is stopped, all the active modes' exit activities are executed.*

Syntax of Activities

You build activities with the application's objects that you define in the Object Layout, using each object's pre-defined properties and functions.

For example, a 2-position rocker switch has, among others, the following pre-defined properties and functions:

PROPERTY	FUNCTIONS
self (the entire rocker switch)	hide (suppresses the display of the entire rocker switch) show (redisplays the entire rocker switch)
up	connect (moves the switch to the "up" position)
down	connect (moves the switch to the "down" position)

Thus, you could construct an entry activity that would move a rocker switch to its "up" position when you enter a specific mode. In Rapid syntax, it would look like:

```
Switch.up connect
```

where "Switch" is the object, *up* is the property and *connect* is the function.

Some object properties have functions that require an argument in order to be complete. In our tape recorder application, for example, a typical mode activity could be that during the Count mode, the application displays a string on a text display object, named Display. In Rapid syntax, you do this by assigning the string to the display. This activity would look like:

```
Display.contents := NumberCount
```

where "Display" is the object, *contents* is the object's property, *:=* is the property's function and "NumberCount" is the argument (an integer object containing a variable that is to be displayed).

Some functions require two or three arguments. For example, the function *deleteFrom:to:* that deletes a section of a string:

```
String deleteFrom: 5 to: 7
```

This activity deletes the 5th, 6th and 7th characters of "String." If the string was "Emultek", the result would be "Emul".

Mode Activities

While entry and exit activities only occur once (when a mode is entered or exited), mode activities can take place **continuously** the entire time that the mode is active.

Mode activities are different from entry/exit activities in two major respects:

- They are continuously applied to the object.
- If they are dependent on an argument, then a change in the argument results in a change in the object.

Mode activities are continuous

You can use some functions as entry/exit activities and as mode activities. The table below demonstrates the differences between using the Lamp's *on* function as an entry/exit activity and as a mode activity.

ACTIVITY TYPE	RESULT
Entry Activity	When the mode is entered, the lamp is switched on. It can be switched off at any time.
Mode Activity	As long as the mode is active, the lamp is switched on. If it is switched off in another mode that is also active, the mode activity immediately switches it back on.
Exit Activity	When the mode is exited, the lamp is switched on. It can be switched off at any time.

Some functions can only be used as entry/exit activities and never as mode activities. The stopwatch object's *restart* function, for example; it makes no sense for a stopwatch to be continually restarting as long as a mode is active.

Mode activities reflect changes in their argument(s)

Our tape recorder application has a display that shows the current tape count. In the Count mode the application could have the following mode activity:

Display.contents := NumberCount

where NumberCount is an integer object. If the value of NumberCount changes, the activity will take place again (i.e., the display will be updated).

The "Object Reference" manual details all the properties and functions that can be used by each object to build mode and entry/exit activities.

Timing of mode activities

Although mode activities are continuous, if the timing of when the activity occurs is important, it is better to use an internal transition triggered by an event. That is, an internal action (see below).

ACTIONS

Actions are similar to activities, but they take place only during transitions. That is, activities occur "within" modes, actions occur "in-between" modes.

The basic order of operations when a transition is triggered is:

1. Source mode's exit activities.
2. Transition's actions.
3. Destination mode's entry activities.

Going back to our tape recorder example, a typical operation that could be performed during the transition from the Off mode to the On mode might be switching on an indicator lamp. This operation is accomplished by attaching an action to the Off-to-On transition. In Rapid syntax, the action would look like:

```
Lamp on
```

where "Lamp" is the object and "on" is the function. The result of this action is that when the tape recorder is switched from off to on, the lamp lights.

Refer to the "Object Reference" manual for a complete listing, by object, of properties and functions that can be used to construct actions.

When should you use an entry activity and when an action?

When several transitions (each from a different mode) to one destination mode all need the same operation to take place, it would be more efficient to define an entry activity for the destination mode.

For example, when the tape recorder is recording, you want an LED indicator to blink. The Record mode can be entered from two modes (Stop and Play). In this case, it is better to have *Lamp blink* as an entry activity for the Record mode.

If, however, you do not wish the same operation to be associated with all the transitions, you must attach specific actions to the relevant transitions.

Internal Actions

An internal action is one that is triggered by an internal transition. That is, it takes place within the mode, while the mode is active. Because an internal action only occurs if its trigger is true, it can be likened to a conditional mode activity.

CHAPTER 11

The Mode Tree

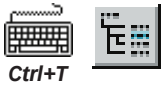
The behavior of Rapid applications is governed by the flow from one mode to another, with different actions and activities being performed in each mode.

In this chapter you will read about:

- Adding a new mode to the tree.
- Editing a mode's name.
- Designating a mode as a default mode.
- Hiding and showing modes in the tree.
- Cutting, copying and pasting modes.
- Restructuring the tree by moving and inserting modes.
- Defining transition destinations from the modes.

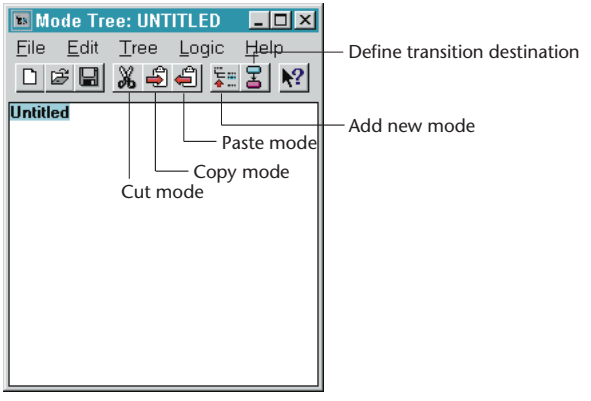
If you want information on how to build transition triggers, actions or activities, see Chapter 12: "Building Logic."

THE MODE TREE WINDOW



To open the Mode Tree:

- Choose Tools|Mode Tree in the Application Manager, or click the Mode Tree button.



The Mode Tree is the window in which you add, edit and order the application's hierarchy of modes. Once the mode names are listed here, you can select them for use in the Logic Editor and the State Chart.

It is often convenient to use the State Chart in conjunction with the Mode Tree. The State Chart provides a graphic, bird's-eye view of the modes and transitions in a nested chart format. See Chapter 17: "The State Chart" for detailed instructions on how to operate the State Chart.

DEVELOPING THE MODE TREE

The first mode in a mode tree is the root mode, which bears the application's name. The Tree menu provides the commands necessary to add, edit and order modes as you develop an application's mode tree.

There is also a popup menu which contains identical commands and additional commands that allow you to build a selected mode's logic.

To select a mode in the Mode Tree:

- Either click it or highlight it using the up/down arrow keys.

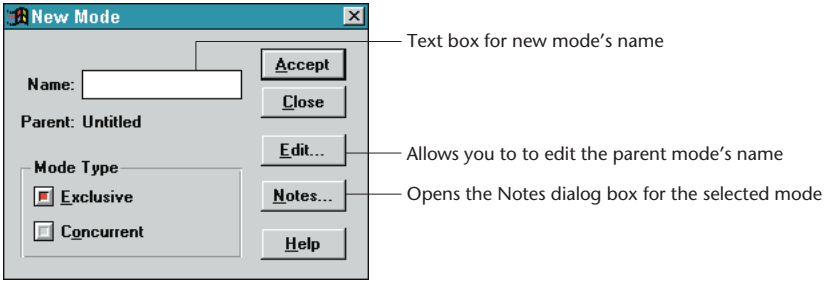
Adding a New Mode

To add a new mode:



1. Select a parent mode.
2. Choose Tree|New Mode, or click the New Mode button.

   The New Mode dialog box opens with the selected mode's name in the Parent field:



3. Type in the new mode's name in the Name text box. (Mode names must obey the same rules as object names, see p. 2–21.)
4. Click the appropriate radio button to define the type (exclusive or concurrent; exclusive is the default).

❖ *NOTE: A new mode must be the same type as any existing sibling modes—exclusive or concurrent.*

5. Click Accept; the dialog box remains open. The new mode appears below its parent in the tree (and inside its parent in the State Chart). Exclusive modes are displayed in black, concurrent modes in blue.

   If this is the first exclusive child mode to a parent (congratulations!), it will become the default mode (marked by an arrow). For the significance of default modes, see p. 10–7, "Default transition".
6. Repeat steps 3–5 to add additional modes; otherwise, click Close.

❖ *NOTE: Rapid checks for illegal mode names (such as the use of spaces or a number as the first character) and, if necessary, displays an appropriate error message.*

To continue developing the tree:

1. Make sure the parent mode is the correct one (select another mode to make it the parent, if necessary).
2. Type in the new mode name.
3. Select a mode type.
4. Click Accept to add the mode to the tree. If it is a sibling mode, it appears as the last mode in the branch.

Editing a Mode Name

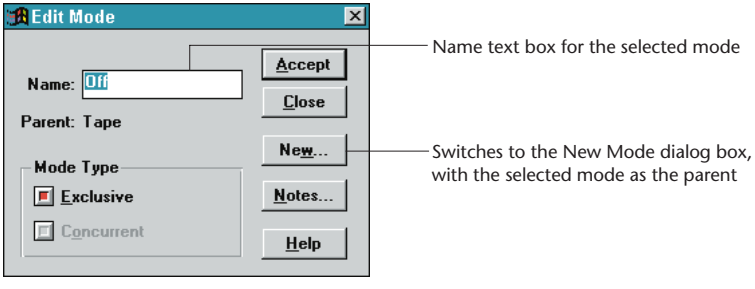
After you have added a new mode, you can change its name.

To modify a mode's name:

1. Select the mode.
2. Choose Tree|Edit Mode; the Edit Mode dialog box opens.

   Ctrl+E

   This dialog box allows you to edit the mode's name. It also provides information on the mode's type and parent.



A sample Edit Mode dialog box

3. Edit the name in the Name text box and then click Accept; the dialog box remains open and the name is changed in the Mode Tree (and in the Logic Chart).

To continue editing mode names:

1. Select another mode in the tree (or in the State Chart).
2. Edit its name in the dialog box.
3. Click Accept.

Changing the root mode's name

The root mode's name can be modified like any other mode. The only difference is that it is limited to eight characters, since the root mode name is also the application name.

When you change the root mode's name, Rapid changes the root mode **and** the application name. However, it does not save the application—this is **not** the same as the File|Save As command.

Alternating between creating and editing modes

While building the mode tree, you can alternate between creating new modes and editing mode names—without closing the dialog box.

If you have been editing a mode name, click the New button in the Edit Mode dialog box. The dialog box title changes to "New Mode" and the selected mode is transferred from the Name text box to the Parent field.

If you have been creating new modes, click the Edit button in the New Mode dialog box.

Reassigning the Default Mode

The first exclusive child mode that you add becomes the default mode and is marked by an arrow in the Mode Tree. Whenever a default transition is made to this mode's parent, the default child mode always becomes active as well. (For the significance of default modes, see p. 10–7, "Default transition".)

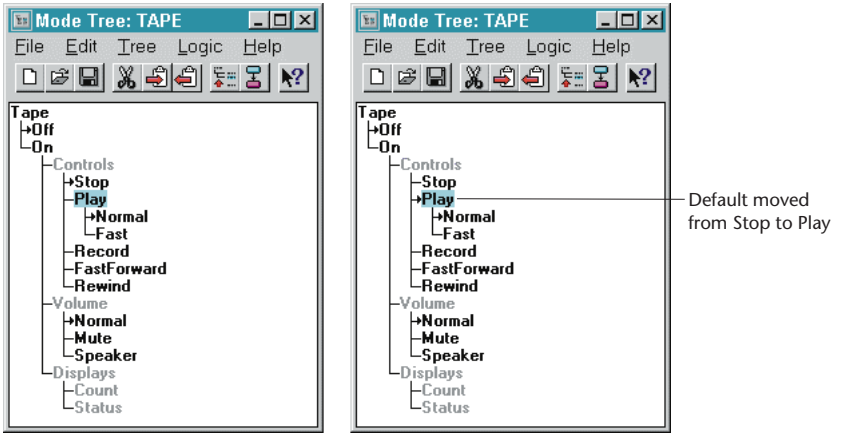
You can make a different sibling mode the default mode. Whenever a default transition is made to the parent mode, Rapid activates the new default child mode instead of the old one.

You can remove the default designation completely from a group of sibling modes, to achieve certain "special effects," (see "Removing the Default," below).

To make a mode the default mode:


Ctrl+D

1. Select the mode.
2. Choose Tree|Make Default; Rapid moves the arrow from the old default mode to the new default mode.



1. Mode selected

2. Mode made default

Removing the Default Designation

If you remove the default from a group of exclusive sibling modes, you create a situation that is potentially unstable. When a default transition is made to the parent, Rapid does not automatically know which child mode to activate. The result could be that Rapid stops running the application in the Prototyper.

If you plan carefully, though, you can use the remove default feature to build a "selector" routine, for example when switching from one TV channel to another. How to build a selector routine is discussed on p. 16–6.

To remove the default designation from a mode:



Ctrl+F

1. Select the default mode.
2. Choose Tree|Remove Default; Rapid removes the arrow from the mode.

Hiding and Showing Modes

In large applications that have many modes, the tree can often become long and unwieldy. You can make the Mode Tree easier to use by temporarily hiding descendant modes.

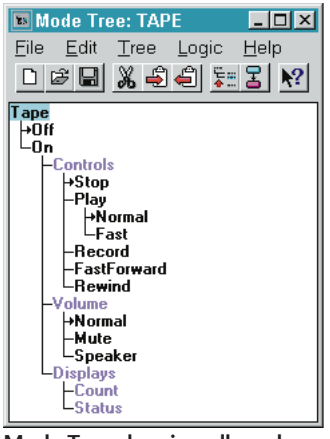
To hide a mode's descendants:



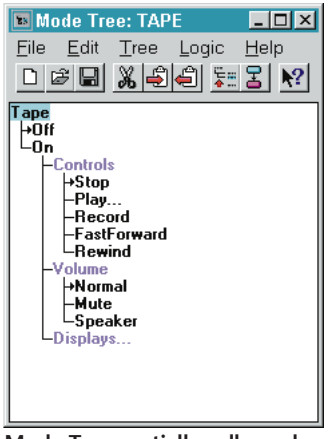
Ctrl+H

- Select the mode and choose Tree|Hide, or double-click the mode.

When its children are hidden, the parent name is followed by an ellipsis (...) The Hide command changes to Show.



Mode Tree showing all modes



Mode Tree partially collapsed

To expand the branch:

- Choose Tree|Show, or double-click the collapsed mode.

Cutting, Copying and Pasting Modes

You can cut, copy and paste modes from one part of the tree to another. Cutting or copying a mode also cuts or copies its descendants (that is, you cut or copy a subtree). This allows you to quickly create similar structures of modes at different places in the tree.

When you cut or copy a mode, Rapid includes the mode's logic. That is, you don't just cut or copy an empty shell (the mode), you also cut or copy the contents (the transitions between the subtree's modes and their activities).

❖ *NOTE: These operations are internal; the Windows Clipboard is not involved.*

Cutting modes

To cut a mode:

1. Select the mode.
2. Choose Tree|Cut, or click the Cut button; Rapid removes the mode (or subtree) and places it into the internal paste buffer.



❖ *NOTES: Any transitions between the cut modes and other modes outside the subtree are removed and cannot be recovered. Since the Mode Tree does not have an Undo function, cutting a mode that has external transitions is irreversible. You can paste it back, but you will need to redefine its external transitions.*

Copying modes

To copy a mode:

1. Select the mode.
2. Choose Tree|Copy, or click the Copy button; Rapid copies the mode (or subtree) to the internal paste buffer.



❖ *NOTE: When you copy a mode or a subtree, Rapid copies its internal transitions (including transitions between the subtree modes) and its activities; it does not copy its external transitions.*

Pasting Modes

To paste a mode:

1. Select the mode that you want to be the parent of the pasted mode.
2. Choose Tree|Paste, or click the Paste button; Rapid pastes the mode (or subtree) from the internal paste buffer. The pasted mode becomes the **last** child of the selected mode.



If you paste a mode that has the same name as one of its new siblings, the Clash Names dialog box opens. In this dialog box you can either change the pasted mode's name or provide it with a suffix or prefix.

Quickly duplicating modes

You can quickly duplicate a mode (and its descendants) to another part of the tree using the Ctrl key:



1. Ctrl-click the mode; the mode and its descendants are grayed and the cursor changes to the Duplicate Mode cursor.
2. Drag the cursor until the new parent mode is highlighted.
3. Release the mouse button; the duplicated mode and its descendants appear as children of the highlighted mode.

When you use this method of duplicating modes, Rapid does not copy the mode to the internal paste buffer.

Moving Modes

You can rearrange the logical order of your modes by moving a mode (or a subtree) to another mode (which becomes its parent mode). Unlike cutting and pasting modes, moving modes does not remove external transitions.

To move a mode:

1. Select the mode to be moved.
2. Choose Tree|Move; the selected mode and its descendants are grayed, and the Move command changes to Cancel Move.
3. Click the mode to which the modes should be attached. The grayed modes disappear, and then reappear as children of the selected mode.

❖ *NOTE: You cannot move modes to a place that would result in illegal transitions, invalid siblings, etc. (for example, you cannot move a concurrent mode so that it has exclusive siblings).*

If you change your mind before you complete the move operation, choose Tree|Cancel Move.

Quickly moving modes

You can quickly move a mode (or subtree) to another part of the tree by dragging it:



1. Click and drag the mode until the new parent mode is highlighted. As you drag, the mode and its descendants are grayed and the cursor changes to the Move Mode cursor.
2. Release the mouse button; the moved mode and its descendants appear as children of the highlighted mode.

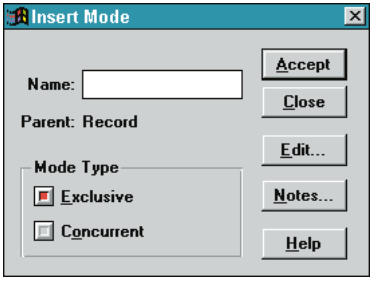
Inserting Modes

You can add a new generation of modes **in between** two current generations by inserting a mode as a child of a selected mode. The selected mode's children become children of the new mode.

To insert a mode:



1. Select the mode that you want to be the parent of the inserted mode.
2. Choose Tree|Insert; the Insert Mode dialog box opens:

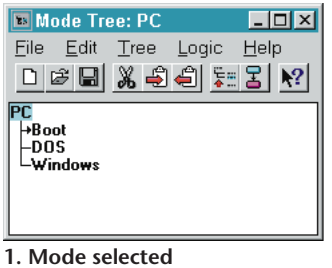


This dialog box is identical to the New Mode dialog box (see p. 11–3) except for the title.

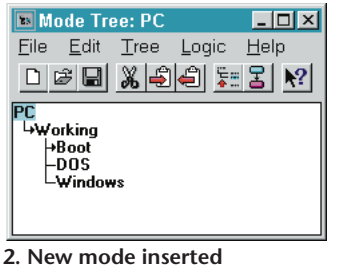
3. In the Name text box, type the name of the mode that you want to insert.
4. Select the appropriate mode type.
5. Click Accept; the mode is inserted as the default child of the selected mode. The selected mode remains selected and the dialog box changes to a New Mode dialog box.

Example:

This example shows an early stage of an application simulating a PC. The Boot, DOS and Windows modes are children of the root mode. A new mode, "Working," is inserted between PC and Boot:



1. Mode selected



2. New mode inserted

Removing Modes

You can permanently remove a mode from the mode tree. The children of a removed mode become children of its parent.

❖ *NOTE: Removing a mode is different from cutting it (if you cut a mode, you cut its descendants as well).*

If you remove a mode, Rapid deletes all transitions to that mode. Unlike cutting a mode, you cannot paste a removed mode.



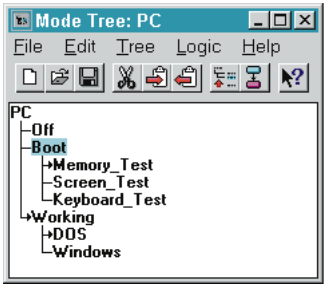
To remove a mode:

1. Select the mode that you want to remove.
2. Choose Tree|Remove.

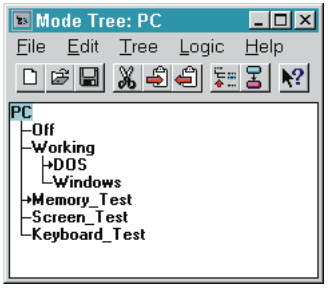
   Rapid checks that the selected mode's children can legally become children of its parent. If any rules are violated, an appropriate error message appears. If no rules are violated, Rapid asks you to confirm the removal.
3. Click Yes to confirm the removal of the selected mode or No to cancel the operation.

Example:

In this example, Boot is removed from the tree. Note that Boot's children become children of Boot's parent, at the foot of the tree.



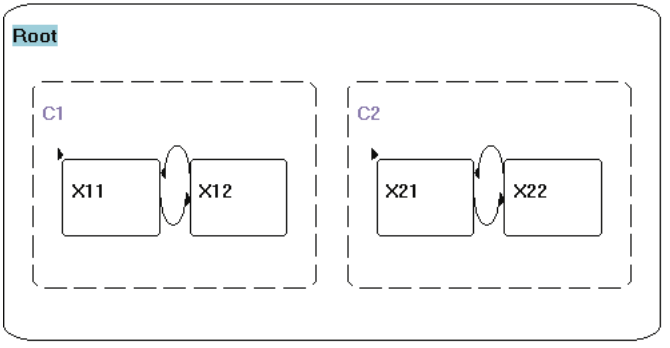
1. Mode selected



2. Mode removed

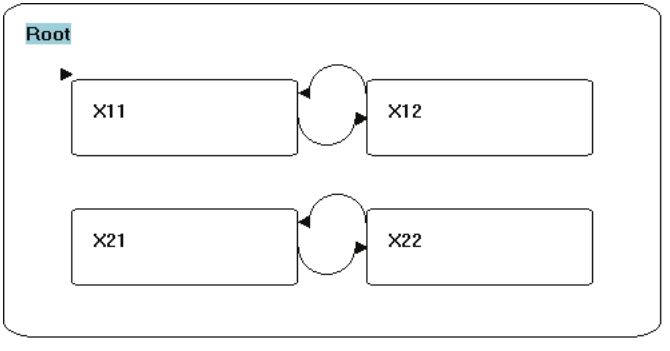
A practical example of inserting and removing modes

Combining insert and remove mode operations in a creative manner makes it possible to change an application's logical design in midstream. The state chart below shows an application in which C1 and C2 are two concurrent children of the root mode. Their exclusive children (X11/X12 and X21/X22, respectively) have transitions between them.



If you decided that you would rather have the four exclusive modes as children of the root mode, you could carry out the following series of move, insert and remove operations to modify the tree (without losing the transitions already defined):

1. Insert an exclusive mode (X0) between C2 and X21.
2. Move X0 (and its children) so that it is a child of C1, then remove X0. C1 now has four children.
3. Remove C2 and C1. The result is shown below:



CREATING TRANSITION DESTINATIONS

You can define transitions between an exclusive mode and any number of other exclusive modes. For example, if Mode A is the source mode for transitions to Modes B, C and D, the application can move from Mode A to **any** of the three destination modes.

- The transition trigger determines **which** transition the application takes.
- The transition type determines which of the destination mode's descendants become active.

❖ *NOTE: A mode can also have an internal transition. This transition is used for triggering actions without exiting the current mode (see p. 10–10). You can only define internal transitions in the Logic Editor (see p. 12-16).*

To define transition destinations, you need a list of the application's modes that enables you to define the source and destination of each transition. A number of Rapid's windows present you with a suitable mode list—the Mode Tree, the State Chart and the Logic Editor. You can define transitions in each of these windows, though each one is suitable in different situations.

Since the Mode Tree presents all the modes in a tree form and distinguishes between exclusive and concurrent modes, it is often the most convenient place to define several transitions. If you only want to define a few transitions, you may prefer the Logic Editor (see p. 12–14).

Creating a Transition Destination

To create a transition destination in the Mode Tree:

1. Select the source mode.
2. a) If the Logic Editor is not open:
   Choose Logic|Transitions or click the Transition button; the Logic Editor for the selected mode opens.

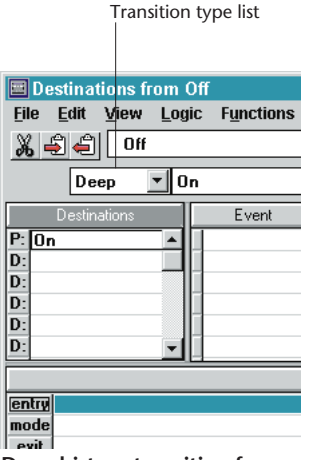
   b) If the Logic Editor is already open:
   Click in the Destinations column.
3. Alt-click the destination mode in the Mode Tree.

   The mode is highlighted in magenta in the Mode Tree and the name appears in the Logic Editor's Destinations column.

❖ *NOTE: You can also select the destination by Alt-clicking it in the State Chart.*

4. Select the appropriate transition type (default, history, or deep history) from the transition type list. For a detailed discussion of transition types, see p. 10–7.

   The transition type is indicated by a letter in front of the destination mode's name (D: for default, H: for history, P: for deep history).



Deep history transition from Off to On

To continue adding transition destinations:

1. Make sure the source mode is the correct one (if necessary, select another mode in the Mode Tree).
2. Alt-click the destination mode.
3. Make sure the transition type is the correct one (if necessary, choose another type from the list).

Editing a transition destination

You can modify a transition destination at any time— even after you have created a complete transition that includes triggers and actions.

To modify a transition destination:

1. Select the destination in the Logic Editor's Destinations column.
2. Modify the destination name and/or transition type as desired.

FINDING A MODE

In this chapter you have read about various procedures that can be performed on modes once you have added them to the mode tree. Examples of these procedures are editing, cutting and pasting, moving, and removing. There are, in fact, numerous logic-building situations in which you will want to select a particular mode in the mode tree.

It can sometimes be difficult to find a mode quickly, especially if your mode tree is large and/or branches have been collapsed (see "Hiding and Showing Modes" on p. 11-6). In such cases, you can use the Find Mode feature to help you go directly to a mode in the tree:

To find a mode in the mode tree:


Ctrl+F

1. Choose Edit|Find Mode.
2. A dialog box opens in which you are prompted to enter the mode name to find. Type in the mode name and click OK.

Rapid goes to (and selects) the specified mode in the tree. If the mode is in a collapsed branch, the branch is automatically expanded.

❖ *NOTES: The search is not case-sensitive.*

*You can use an asterisk (*) to designate wildcards when specifying the mode name. For example, typing in '*power*' (without the apostrophes) will generate another dialog box with a list of all modes that contain the string 'power' such as powerOn, powerOff, and batteryPower. You then double-click on a mode in the list to go to that mode in the tree.*

Accessing Find & Replace

Rapid's Find & Replace utility allows you to search for and replace text in the application logic. You can access the Find & Replace window from the Mode Tree by choosing Edit|Find/Replace. For a full discussion on the Find & Replace features, see Chapter 14, "Find and Replace".

CHAPTER 12

Building Logic

Once you have defined and organized your application's modes in the Mode Tree, you can move on to the Logic Editor—the module used to:

- Add transition destinations.
- Build triggers (i.e., events and conditions), actions and activities.
- Build user-defined functions.

(See Chapter 18 for more information about user-defined functions.)

The Logic Editor stores the logic in a table format that is organized according to the application's modes.

The Logic Palette (described later in this chapter) provides a palette of properties, functions, conditions and events. It can be used in conjunction with the Logic Editor to build logic.

At any point you can run a Verification Test (described at the end of this chapter) to check for modes which cannot be reached or exited, absent triggers and unreferenced objects.

THE LOGIC EDITOR WINDOW

Opening the Logic Editor

You can open the Logic Editor from the Application Manager or from the Mode Tree (when you create an activity or transition).

To open the Logic Editor from the Application Manager:

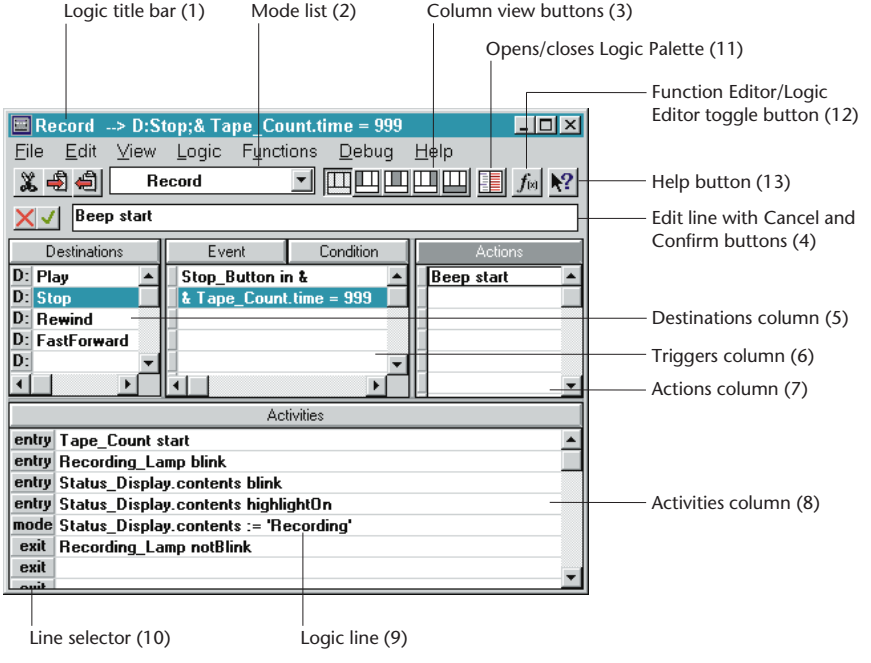


- Choose Tools|Logic Editor, or click the Logic Editor button.

To open the Logic Editor from the Mode Tree:

- Choose Logic|Activity (or Transition) or choose Activity (or Transition) from the popup menu.

A typical Logic Editor window is shown below:



The components of the Logic Editor are described below.

1. Logic title bar

The title bar displays the current mode's logic. In the above illustration:

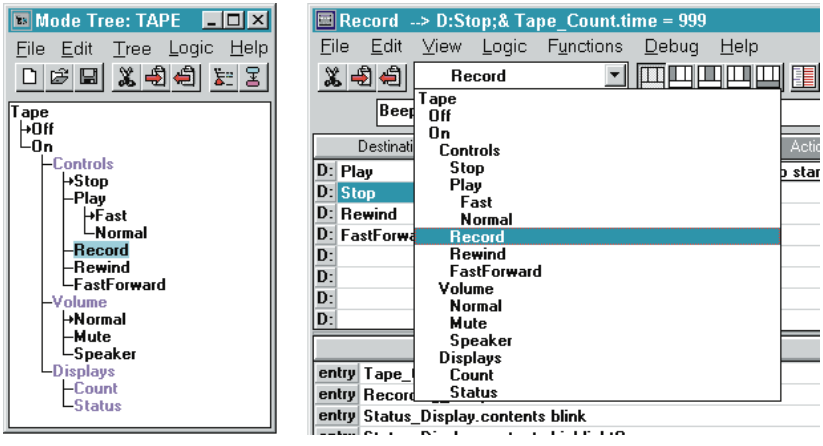
- The mode being edited is the Record mode.
- The Stop transition destination is selected (shown as Record —> D:Stop).
- A condition is selected (& Tape_Count.time = 999).

2. Mode list

The application's modes are displayed in the Logic Editor's mode list. To edit the logic of a mode, select it from the mode list.

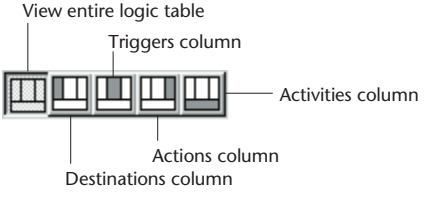
❖ *NOTE: Selecting a mode in the Mode Tree also makes that mode the Logic Editor's current mode.*

The mode list displays the modes in a tree form, similar to the Mode Tree, as shown below:

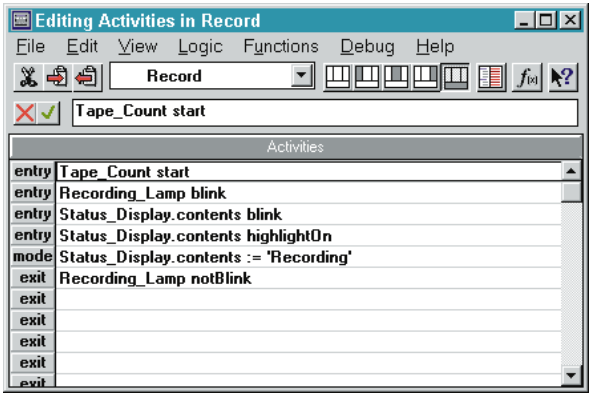


3. Column view buttons

Click these buttons to zoom in and out of the columns.



Zooming in to a column gives you more room for editing and allows you to concentrate on one aspect of the mode's logic. Shown below is the Logic Editor after zooming in to the activities column.



4. Edit line and its Confirm and Cancel Buttons

The edit line is the place where you build and edit logic. You build logic phrases by either typing them directly using the keyboard or constructing them in the Logic Palette and appending them to the edit line. Whatever is entered in the edit line appears in the active logic line.



Click the Confirm button to confirm a logic phrase (or right-click and choose Accept from the popup menu).



Click the Cancel button to cancel the logic phrase and restore the original contents of the active line (or right-click and choose Restore Previous from the popup menu).

When you edit a transition destination, the edit line is replaced by the Destination Mode and Transition Type list boxes (see To create a transition destination on p. 12-14).

The columns

The Logic Editor is set up like a table divided into two parts. The upper part contains logic relating to the mode's transitions and comprises three columns: destinations, triggers and actions. The lower part of the table displays the mode's activities.

Above each column is a title button. When you click a column's title button, the focus is moved to that column and the first empty line is activated.

❖ *NOTE: As a general rule, the order of items in a column determines the order in which Rapid executes the logic.*

5. Destinations column

This column displays all the transition destinations of the current mode. Each external destination's transition type is displayed in its line selector (e.g., D for default). The transition destinations appear in the order in which you create them, though you can change that order.

When you select a transition, all of its triggers are displayed in the Triggers column (see the following illustration).

6. Triggers column (events/conditions)

This column displays all the triggers of the selected transition destination. Each trigger is either a condition, an event or a combination of both. The triggers appear in the order in which you create them, though you can change that order.

You click the Event button to edit events; you click the Condition button to edit conditions.

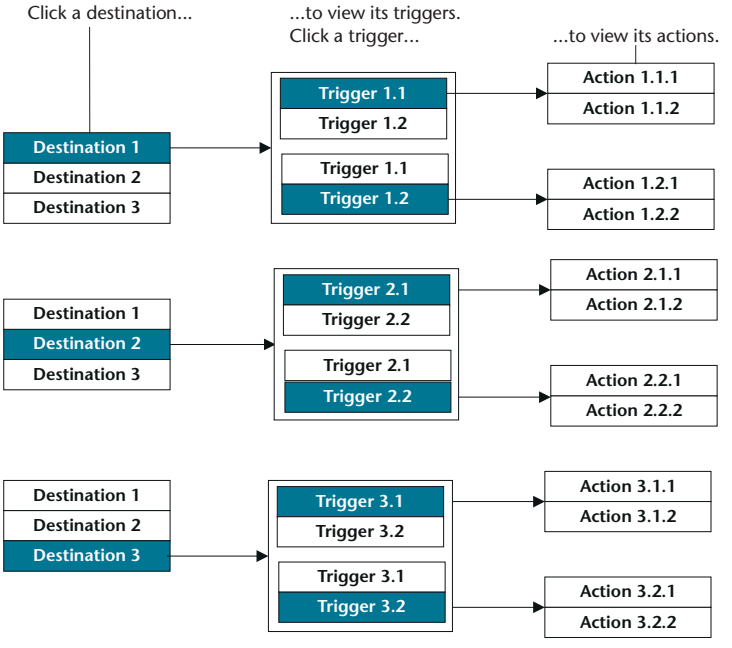
When you select a trigger, all of its actions (if any) are displayed in the Actions column (see the following illustration).

❖ *NOTE: This column is empty if the current mode does not have any transition destinations, or if no destination is selected.*

7. Actions column

This column displays all the actions of the selected trigger (see the following illustration). The actions appear in the order in which you create them, though you can change that order.

❖ *NOTE: This column is empty if no trigger is selected.*



8. Activities column

This column displays all the activities of the selected mode. The activities are grouped according to type: entry, mode and exit.

There are two ways to change an activity's type; they are described on p. 12-29.

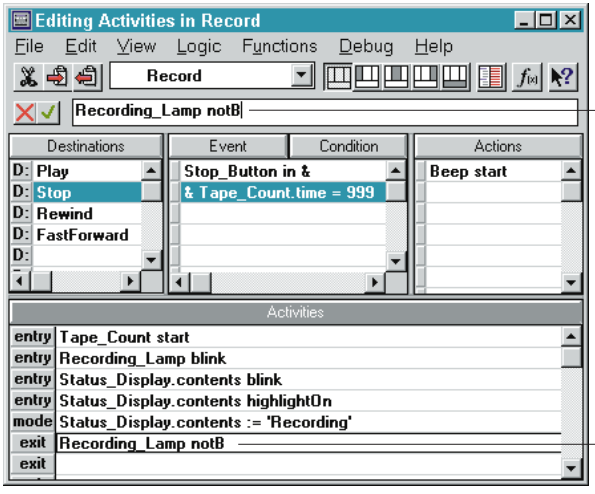
Using the title button

The Activities title button cycles through the activity types. When you click the button, the first empty line of the next type is selected. If there is no empty line, Rapid adds one.

For example, if the currently active line is an entry activity, clicking the button selects the first empty mode activity line. If there is no empty mode activity line, Rapid adds one.

9. Logic line

A mode's logic is made up of separate lines. Each line contains one complete phrase of logic. When you enter logic into the edit line, it simultaneously appears in the active line. Note that you cannot enter text directly into the lines.



10. Line selectors

Every line has its own selector located to the left of the line. You use line selectors to select multiple lines with the Ctrl or Shift keys.

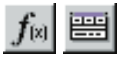
In the Destinations column, the line selector indicates the transition type (default, history or deep history) for external transitions; it is blank for internal transitions. In the Activities column, the line selector indicates whether the activity is an entry, mode or exit activity. In the Triggers and Actions columns, the line selector is blank.

11. Logic Palette toggle button



This button opens and closes the Logic Palette (equivalent to the View|Logic Palette command). The Logic Palette is used to construct the application's logic using a simple "point-and-click" method. Refer to p. 12–8 for a description of the Logic Palette.

12. Function Editor/Logic Editor toggle button



This button toggles between the Function Editor and the Logic Editor (equivalent to the View menu's Mode Logic and User Function commands). The Function Editor is used to build user-defined functions. It is described in Chapter 18: "User-Defined Functions."

13. Help button



For online Help about Logic Editor/Function Editor components, click this button, then click the component.

THE LOGIC PALETTE

The Logic Palette provides a list of all of the properties, functions, conditions and events for the application's objects. It is used to build events, conditions, actions and activities using a simple "point-and-click" method. Each time you build a logic phrase, you can append it to the Logic Editor's edit line.

Opening the Logic Palette

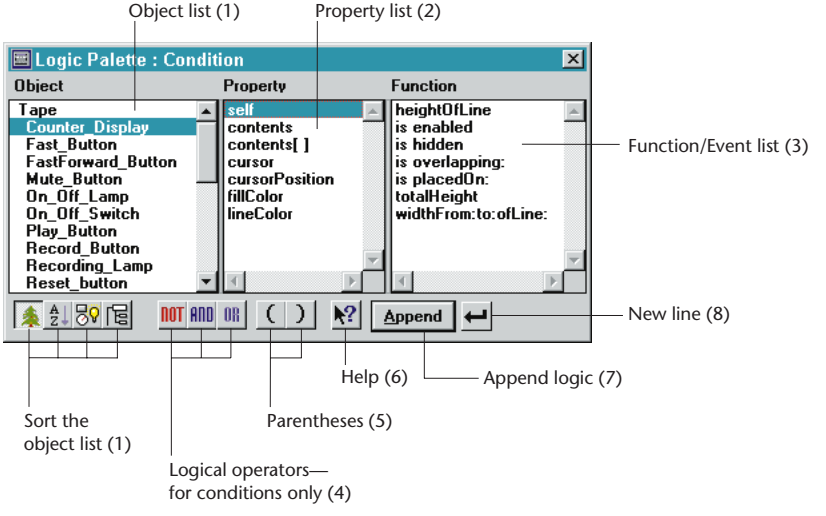
To open the Logic Palette:

Either:

- Choose View|Logic Palette (it opens in activity-editing mode).
- Click the Logic Palette tool button.



The Logic Palette shown below is in condition-editing mode.



The components of the Logic Palette are described below.

1. Object list and sort buttons

This list displays all the application's active and nongraphic objects. If you want to build a trigger based on the status of a mode (for example, to test whether a mode is active), you can also display the Mode Tree.

If an object is not relevant to the current logic-editing mode, it is grayed. For example, when the Logic Editor is in event-editing mode, those objects that

do not generate events are grayed. The buttons below the object list are used to select the way in which the objects are sorted:

	TYPE OF LIST	DESCRIPTION
	*Object Tree*	Rapid displays the objects in a hierarchical tree, beginning with the Root object. Each child object is displayed below its parent. The nongraphic objects are displayed at the foot of the tree.
	*Objects by Name*	Rapid displays the objects in an alphabetical list according to the objects' names.
	*Objects by Type*	Rapid displays the objects in groups, according to their class. All the switches appear together, all the potentiometers appear together, etc. The name of each class is displayed in blue. Double-clicking the class name hides or shows a class's objects; a collapsed class has an ellipsis (...) following its name.
	*Mode Tree*	Rapid displays the Mode Tree.

2. Property list

This list displays all the properties of the selected object.

3. Function/Event list

This list displays the functions of an object's selected property. The functions can be actions, activities, conditions or events—depending on the Logic Editor's current editing-mode.

Moving up and down the Logic Palette lists

You can select items in the lists like in standard Windows lists:

- Typing Home or End selects the first or last item, respectively.
- You can move up and down one item at a time with the arrow keys.
- Typing a letter selects the next item that begins with that letter.

4. Logical operators



The "not," "and," and "or" buttons are used to append a logical operator to the edit line. You use these operators when building certain types of conditions (the buttons only appear in condition-editing mode).

These buttons are discussed in more detail on p. 12-23.

5. Parentheses



The parentheses buttons append an opening or closing parenthesis to the edit line. These parentheses are useful for grouping mathematical expressions. These buttons do not appear in event-editing mode.

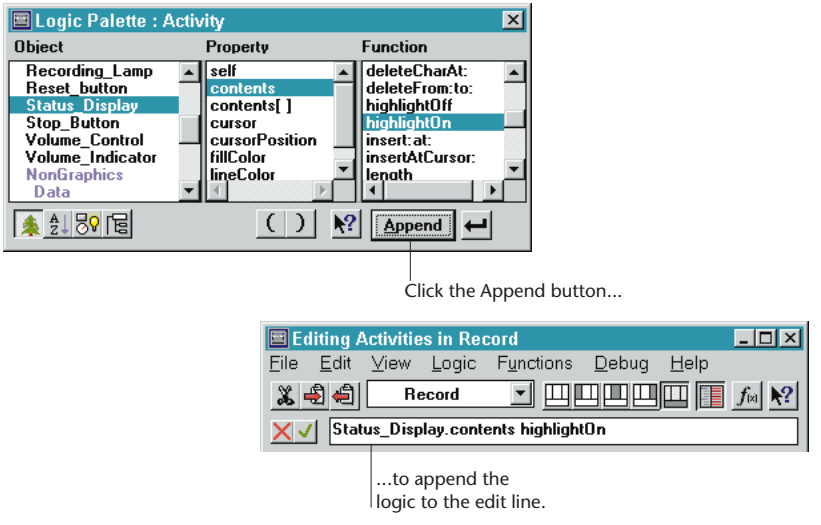
6. Help



For online Help about a component, function, or event in the Logic Palette window, click this button, then click the item.

7. Append button

The Append button transfers whatever is currently selected in the Object, Property and Function/Event lists to the edit line.



8. New line



The New Line button confirms the logic that is in the edit line and adds a new (blank) line below the active logic line. The new line becomes the active line.

Finding an Object in the Logic Palette

It can sometimes be difficult to find an object quickly, especially if your object list is long and/or object classes have been collapsed (in Objects by Type view—see p. 12-9). In such cases, you can use the Find Object feature to help you go directly to an object in the list:

To find an object in the object list:

1. Right-click anywhere in the object list and choose Find Object from the popup menu.
2. A dialog box opens in which you are prompted to enter the object name to find. Type in the object name and click OK.

Rapid goes to (and selects) the specified object in the list. If the object is in a collapsed class, the class is automatically expanded.

❖ *NOTES: The search is not case-sensitive.*

*You can use an asterisk (*) to designate wildcards when specifying the object name. For example, typing in '*power*' (without the apostrophes) will generate another dialog box with a list of all objects that contain the string 'power' such as power_pb, nuclearPower_sw, and manpower_disp. You then double-click on an object in the list to go to that object in the list.*

Accessing Find & Replace

Rapid's Find & Replace utility allows you to search for and replace text in the application logic. You can also focus your search on logic statements that reference a specific object and property.

You can access the Find & Replace window from the object list popup menu by choosing Find/Replace. For a full discussion on the Find & Replace features, see Chapter 14, "Find and Replace".

Inspecting an object

When the Prototyper is running (either started or paused), you can examine the state and contents of any application object in a special inspection window. For a full discussion of this feature, see "Inspecting the Contents of an Object", p. 15-8.

To specify an object to inspect via the object list of the Logic Palette:

1. Select the object in the list.
2. Right-click on the object and choose Inspect in the popup menu.

GENERAL EDITING IN THE LOGIC EDITOR

Using the Edit and Popup Menus

The Edit menu contains the commands that are common to all of the columns: Insert, Cut, Copy, Paste, Delete, Select All and Make Function. These commands are available for the active column.

Each of the columns has its own popup menu that contains all of the commands found in the Edit menu, plus commands particular to the column.

To open each column's popup menu:

- Right-click anywhere within the column.

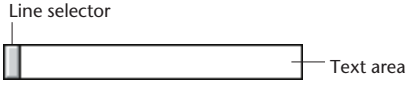
(The popup menu commands are described in the "Manipulating Lines of Logic" section starting on p. 12-31.)

Selecting Lines



The Logic Editor has two main mouse cursors—a "cross" cursor and the regular arrow cursor. While working in the body of the Logic Editor, the normal cursor is the cross cursor—used for selecting lines of logic. (The arrow cursor shows that you can drag selected lines, see p. 12-33.)

You select a line by clicking the line. To select multiple lines using the Ctrl or Shift keys, click the line selectors.



To select a line:

- Click anywhere in the line. When a line is selected, it is able to receive input.

To select a range of lines:

Either:

- Click the first line and drag down until all the lines in the range are selected.

Or:

- Click the first line's selector and Shift-click the selector of the last line in the range.

❖ *NOTE: Selecting more than one line is useful if you want to perform an operation on several lines, such as deleting, copying or dragging.*

To select non-adjacent lines:

1. Click the line selector of one of the lines.
2. Ctrl-click the line selectors of each additional line.

   (To deselect one of the lines, Ctrl-click it.)

To select all of the lines in a column:

1. Click a line in the column.
2. Choose Edit|Select All.

Editing a Line's Logic

To edit a line's logic:

1. Click the line; the logic appears in the edit line.
2. Modify it as necessary, then confirm the change.

Confirming changes to the logic

When you enter logic into the edit line, it also appears in the active line. To confirm the logic, you can carry out any of the following operations:

OPERATION	RESULT
Click the Confirm button.	Rapid accepts the logic, if there are no errors. The line remains active.
Select another line in the same or a different column.	Rapid accepts the logic, if there are no errors. The line that you clicked on becomes active.
Press Enter.	Rapid accepts the logic, if there are no errors, and inserts a new blank line below the active line. The new line becomes the active line.
Click the New Line button.	As above.



Changing a Column's Width



To change a column's width:

- Click and drag its boundaries.

Navigating in the Logic Editor with the arrow keys

If you prefer to use the keyboard while you are working in the Logic Editor, you might like to make use of the arrow keys:

- The up/down arrows move the focus from one logic line to another.
- The right/left arrows move the text cursor in the edit line.

❖ *NOTE: If NumLock is on, you need to press Ctrl together with the arrow keys to move up or down the logic lines.*

CREATING TRANSITION DESTINATIONS

In the Logic Editor, you can create both kinds of transitions:

- External (i.e., from one mode to another).
- Internal (i.e., within a mode).

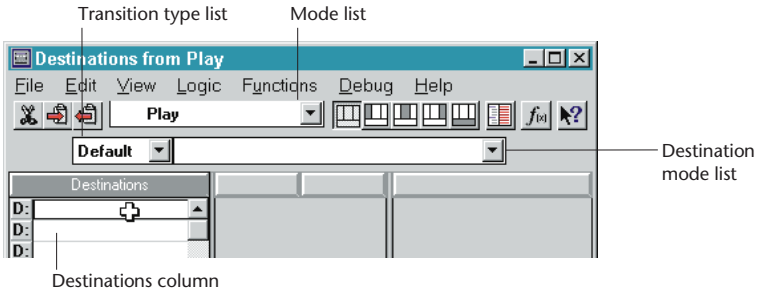
❖ *NOTE: You can also create external transitions in the Mode Tree (see the next page). Internal transitions can only be created in the Logic Editor.*

The Logic Editor's mode list displays all of the application's modes. The current mode is the source mode for any transitions you make.

Creating External Transitions

To create an external transition destination:

1. Make sure that the mode selected in the mode list is the correct source mode. If not, click the list and select a different mode.
2. Click a blank line in the Destinations column; the edit line changes to the destination mode list.



3. The default transition type is "Default." To change it, click the transition type list box and select either History or Deep History. The transition type appears in the line selector (D:, H: or P:).
4. Choose the destination by selecting a mode from the destination mode list. The mode is transferred to the line in the Destinations column.

   SHORTCUT:
   Type the first character of the destination's name to cycle through all modes that begin with that character.
5. Click the Confirm button or press Enter.

Using the Mode Tree to create transition destinations

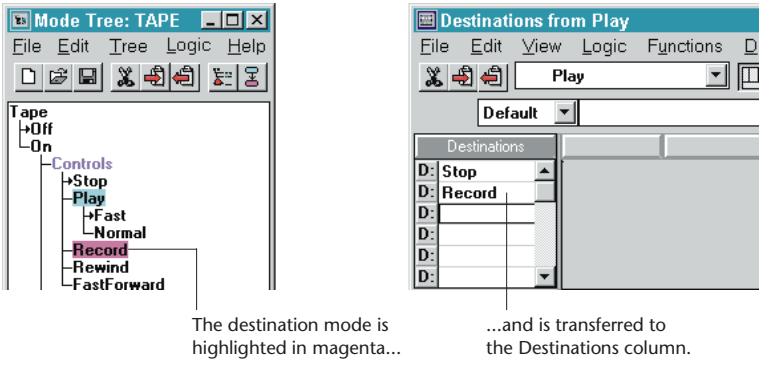
When the Mode Tree is open, you can use it to create transition destinations.

To create a transition destination:

1. Select the source mode in the Mode Tree; it becomes the Logic Editor's current mode.
2. Choose Logic|Transition or click the New Transition button. The focus moves to the Logic Editor, and the first empty line in the Destinations column is selected.



3. In the Mode Tree, Alt-click the destination mode. The destination mode is highlighted in magenta, is transferred to the Destinations column and is automatically confirmed:



Creating Internal Transitions

Internal transitions allow execution of actions that are triggered by conditions and/or events, without leaving the mode. A mode can have several internal triggers, each with its own associated action.

To create an internal transition:

1. Make sure that the mode selected in the mode list is the correct source mode. If not, select a different mode.
2. Click a blank line in the Destinations column; the edit line changes to the destination mode list.
3. Select “Internal” from the destination mode list.
4. Click the Confirm button.

SHORTCUT:
Right-click a blank destination line and choose Internal from the popup menu.

Changing the Transition Type

You can change a transition from being one type to another type.

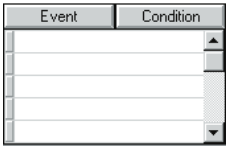
❖ *NOTE: You can change an external destination (default, history or deep history) to an internal destination, but not vice versa.*

To change the transition type:

1. Select the destination.
2. Right-click and choose a transition type from the popup menu.

BUILDING TRIGGERS

A transition occurs only when triggered by an event, condition, or a combination of the two (a compound trigger). To build an event or a condition, click either the Event button or the Condition button at the top of the Triggers column.



This section explains how to build the four types of triggers:

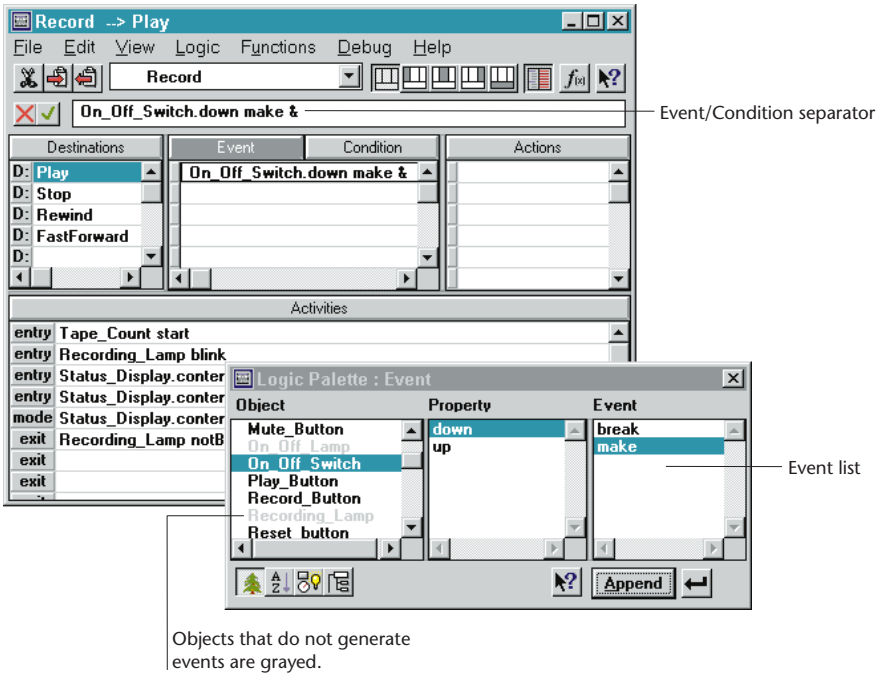
- Events
- Conditions
- Mode triggers
- Compound triggers

❖ *NOTE: You must select a destination before you can build a trigger.*

Building an Event Trigger

An event trigger means that the transition is triggered when the event is generated. You can also build a trigger that consists of several events; when just one of the events occurs, Rapid triggers the transition.

The Logic Editor and Logic Palette in event-editing mode are shown below:



Event syntax

The syntax for single events is:

<object name>.<property> <event> &

Note that the final event is always followed by an ampersand (&).

The use of the *&* symbol

The & symbol is used in Rapid expressions as a separator between events and conditions. It is not the address of a variable as used in the C programming language, nor is it the Boolean “and” operator. (The Boolean “and” operator in Rapid is the keyword “and”.)

The syntax for multiple events is:

<object name>.<property> <event> ! <object name>.<property> <event> ... &

Note that the two events are separated by an exclamation point. Below are examples of events:

EVENT	MEANING
On_Switch.down make &	A switch object called "On_Switch" is moved to the position called "down."
Rewind_but in &	A pushbutton called "Rewind_but" is pushed in. (The property is *self*, which doesn't explicitly appear.)
Timer tick &	A timer object reaches zero. (Again the property is *self*.)

The self property

If the logic involves an object's *self* property, the property does not explicitly appear in the logic. In this case, the period that separates an object from its property is unnecessary.

Building a single event trigger

To build an event trigger:

1. Select the destination of the transition that you want to trigger (in the Destinations column).
2. In the Triggers column, click the Event button. The first empty line is activated and the edit line appears.
3. In the Logic Palette's object list, select the object which will cause the event; the object's event-generating properties appear in the property list.

❖ *NOTE: Objects that cannot generate events are grayed in the object list.*

4. Select the property which will generate the event.
5. Select the event from the event list and click Append; the event is transferred to both the edit line and the line selected in the Triggers column.

   SHORTCUT:
   Double-click the event.

6. Either:
   Click the Confirm button to confirm the logic.

   Or:
   Click the New Line button to confirm the logic and enter a new line.

Building multiple event triggers

By definition, a multiple event is an “either-or” trigger. Two events cannot occur simultaneously.

To build a multiple event trigger:

1. Repeat steps (1) to (4), above, for each event that you want to add to the trigger. Each time you click Append, Rapid automatically adds an exclamation point before the new event to separate it from the previous event.
2. Click Confirm.

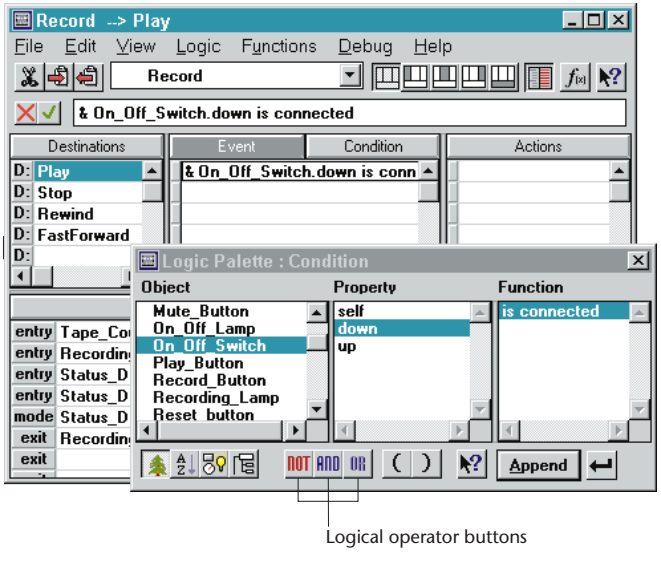
Below is an example of a multiple event trigger:

Tape_Count tick ! Stop_Button in &

The transition is triggered when either the timer object (”Tape_Count”) ticks, (i.e., reaches zero), or when the pushbutton object (”Stop_Button”) is pushed in—whichever happens first.

Building Condition Triggers

A condition trigger means that the transition is triggered when the condition is true. You can include logical operators (“and” and “or”) to build a trigger that consists of several conditions. The Logic Editor and Logic Palette in condition-editing mode are illustrated below:



Condition syntax

The syntax for a single condition is:

& <object name>.<property> <function> [argument]

where [argument] is an optional value against which the value of the object's property is compared; for example, to build a condition that tests whether an integer object equals 4.

Note that the condition is always preceded by an ampersand (&). Below are examples of conditions:

CONDITION	MEANING
& On_Switch.down is connected	Tests whether a switch object called "On_Switch" is connected to the position called "down."
& On_Off_Lamp is on	Tests whether the lamp is on. (The property is *self*, which doesn't explicitly appear.)
& Timer.time >= 60	Tests whether the timer's *time* property is greater than or equal to 60.

Building a single condition trigger

To build a single condition trigger:

1. Select the destination of the transition that you want to trigger (in the Destinations column).
2. In the Triggers column, click the Condition button. The first empty line is activated and the cursor appears in the edit line.
3. In the Logic Palette's object list, select the object whose status will trigger the transition; its properties appear in the property list.
4. Select a property.
5. Select a function and click Append; the condition is transferred to both the edit line and the line you selected in the Triggers column.

   SHORTCUT:
   Double-click the function.

   If the function requires an argument, Rapid prompts you to provide a value.



6. Either:
   Click the Confirm button to confirm the logic.



   Or:
   Click the New Line button to confirm the logic and enter a new line.

Adding a function's argument

Some functions require you to add an argument in the edit line. For example, when you build a condition that tests whether a potentiometer's *value* property has a certain value.

When you append such a function to the edit line, Rapid prompts you to add the argument, informing you of the type of value required:

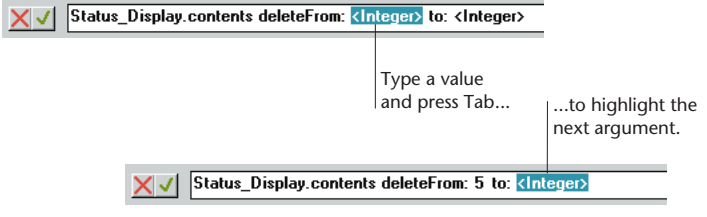


This prompt tells you that Rapid is expecting a number value. You must replace <Number> with either an actual number, or an object property that contains a number value.

For example, you could replace a <Number> prompt with, among others:

- **35.6**
- **NumberObject**
- **Potentiometer.value**

If you enter a function that requires two or more arguments, you can easily replace each prompt by typing a value and pressing the Tab key. Each time you press Tab, Rapid highlights the next argument.



Building negative condition triggers

You might want to construct a condition that triggers a transition when, for example, a number object does **not** have a certain value.

Negative condition syntax

The syntax for a negative condition is:

& not <object name>.<property> <function> [argument]

For example:

& not Switch1.position2 is connected

& not Potentiometer1.value = 45

To construct a negative condition trigger:



1. Before appending the object+property+function to the edit line, click the "not" logical operator button. The operator, *not*, appears in the edit line immediately after the ampersand.
2. Build the rest of the condition as explained for a single condition trigger.

Building multiple condition triggers

Multiple condition triggers consist of more than one condition; the conditions are linked by the logical operators "and" and "or."

In a tape recorder, for example, you might want the play mechanism to work only when the application senses the presence of a cassette. You can simulate the presence of a cassette by adding a pushbutton called "Cassette." When the button is pushed in, a cassette is present. You can then build a multiple condition:

& Play_but is in and Cassette is in

This condition means that Rapid triggers the transition only if the Play_but object is in **and** if the Cassette object is in (i.e., a cassette is present).

In the same application, you might want the recorder to stop when the stop button is in **or** when the cassette tape reaches the end. You can simulate a cassette advancing by adding a timer called "Tape_Count." The timer advances whenever the tape is playing, recording or "fast-forwarding," to a maximum of 999.

You can then build a multiple condition:

& Stop_but is in or Tape_Count.time >= 999

This condition means that Rapid triggers the transition when **either** the Stop_but object is in **or** when the Tape_Count object reaches (or passes) 999.

To construct a multiple condition trigger:

1. Build a condition, as explained for a single condition trigger.
2. Insert the argument's value (if any).
3. Click the "and" or "or" button, as necessary.
4. Build the second condition, including its argument value (if any).



Priority within multiple conditions

Rapid evaluates multiple conditions from left to right, without assigning any priority to the "and" or to the "or." For example, the condition below is only true if either of the first two sub-conditions is true at the same time that the third sub-condition is true.

& Integer = 5 or Button is in and Switch.position3 is connected

As soon as Rapid finds a sub-condition which is false, it immediately returns the whole multiple condition as false and ignores the rest of it. For example, in the above condition, if *Integer = 5* is false and *Button is in* is also false, Rapid ignores the condition on the switch.

If you want to change the priority in which Rapid evaluates a multiple condition, you can use parentheses. For example, if you add parentheses to the above condition, Rapid will evaluate two sub-conditions, not three.

& Integer = 5 or (Button is in and Switch.position3 is connected)

For the second sub-condition to be true, the Button object must be in and the Switch object must be connected to position 3.

Building Mode Triggers

You can trigger transitions based on the current status of a mode, i.e., is the mode active or not. Modes can be used in conditions and to generate events.

A mode event trigger means that the transition is triggered at the instant the mode is entered or exited. A mode condition, on the other hand, tests whether a mode is active or inactive.

Mode trigger syntax

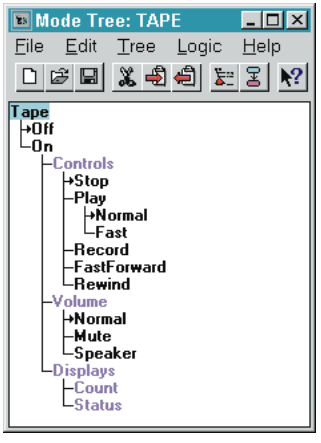
The syntax for a mode condition is:

& Mode/<mode name> <function>

The syntax for a mode event is:

Mode/<mode name> <function> &

Example:



In this application for a tape recorder, you want the counter display to blink when the tape recorder is recording. You can use the status of the Record mode to trigger an internal transition in the Count mode, whose action causes the display to blink.

The logic would take the form "when the Record mode is active, blink the text display's contents:"

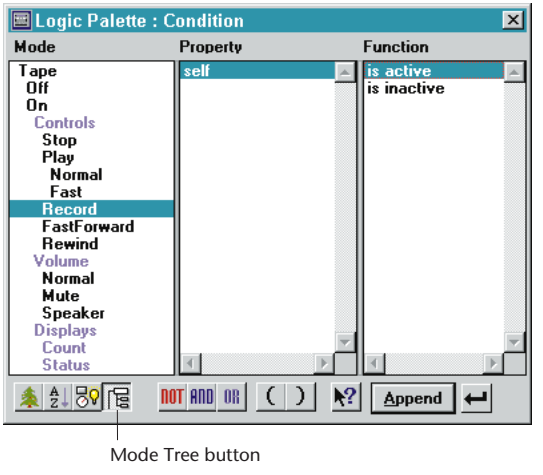
Trigger: **& Mode/Record is active**
Action: **TextDisplay.contents blink**

❖ *NOTE: When you build the trigger using the "point & click" method, Rapid automatically adds "Mode/" when you append the trigger to the edit line. If you build the trigger manually (using the keyboard), you need to add "Mode/" yourself.*

To build a trigger using a mode:

1. In the Triggers column, click the Event button. The first empty line is activated.

2. In the Logic Palette, click the Mode Tree button to display the application's modes in the object list:



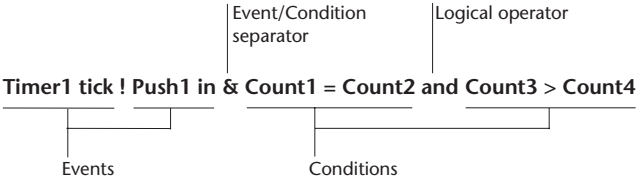
3. Build the condition or event using the mode, as with any object trigger.
4. If necessary, add other object or mode conditions (or events — see "Building Compound Triggers" below) to the trigger.
5. When the trigger is complete, click the Confirm or New Line button to confirm the logic.

Building Compound Triggers

Compound triggers are based on a combination of events and conditions. For example, you might wish to trigger a transition when several conditions and events occur:

- The value of number object "Count1" equals the value of number object "Count2"
- The value of number object "Count3" is greater than the value of number object "Count4"
- Countdown timer "Timer1" reaches 0, or pushbutton "Push1" goes in.

In Rapid syntax, this trigger is represented as:



To build a compound trigger:

1. Click the Event button.
2. Build the first event and append it to the edit line.
3. Build and append the trigger's other events (if any).
4. Click the Condition button.
5. Build and append the trigger's condition(s), separated by logical operands as necessary.
6. Click the Confirm or New Line button to confirm the logic.

❖ *NOTE: If you build the events and conditions in a different order, Rapid automatically "shuffles" them into the correct order (i.e., events followed by conditions).*

BUILDING ACTIONS AND ACTIVITIES

Activities are operations that take place on entry into, exit from or during a mode. Actions are operations that take place during transitions. Activities and actions are constructed in a similar manner.

Mode Activities versus Entry/Exit Activities

Entry and exit activities are performed **once only**, when the mode is entered or exited. For example, the entry activity

Display1.contents := Number1

would display the number object's value when the mode becomes active.

When a mode is entered, its mode activities are performed **continuously** as long as the mode is active. A mode activity will be updated if its function is binary (i.e., accepts an argument) and the value of its argument changes while the mode is active.

Examples:

Examine the following activity:

Display1.contents := Number1

If this is a **mode** activity, then every time Number1 changes, Rapid re-evaluates the := function and updates the text display object (Display1).

For another example of the difference between an entry and a mode activity, let's take the following activity:

Primitive_Object.position := (a@b)

As an entry activity, Rapid would update the position of "Primitive_Object" once when the mode is made active, i.e., the object would move once and then stop. As a mode activity, Rapid would update the position of Primitive_Object with every change of *a* or *b*, i.e., the object moves continuously while the mode is active (assuming that *a* and *b* are changing continuously).

Effect of pausing an application on mode activities

If you pause an application while running it in the Prototyper, and then resume, Rapid re-executes all of the mode activities of the active mode(s)—even those that would not normally be updated.

For example, an application has a mode with two mode activities:

IntegerObject := 25
IntegerObject changeBy: 5

When this mode is active, Rapid executes the first activity (i.e., assigns the value 25 to the integer object) and continuously executes the second activity (i.e., changes the value of the integer object by 5). If you pause the application and then resume it, Rapid executes **all** of the mode's mode activities. The result is that the integer object is once again assigned the value 25.

To avoid this result, you should define any activities that are not to be re-executed following a pause/resume as **entry** activities.

Most activities can be used as entry, exit and mode activities. There are some functions which cannot be used as mode activities. These functions do not appear in the Functions list when a mode logic line is selected.

Building Actions and Activities

To build an action or activity:

1. If you are building an **action**, select the trigger in the Triggers column, then select an empty line in the Actions column.

If you are building an **activity**, select an empty line in the Activities column in the appropriate section (i.e., in the Entry section if it will be an entry activity, etc.). If there are no empty lines in the section, add a line by either clicking the Activities title button or moving an empty line to the appropriate place.

2. In the Logic Palette's object list, select an object.
3. Select the property of the object that will be affected by the action or activity.
4. Select the function that is to be carried out by the object and click Append; the logic is transferred to both the edit line and the line you selected.

   SHORTCUT:
   Double-click the function.
5. If necessary, use the keyboard to add the function's argument (if any). Click the Confirm or New Line button to confirm the logic.

Changing an Activity's Type

You can change an activity's type to alter when Rapid executes it. For example, if you want an activity to occur continuously while the mode is active instead of just when the mode is entered, you can change it from an entry activity to a mode activity.

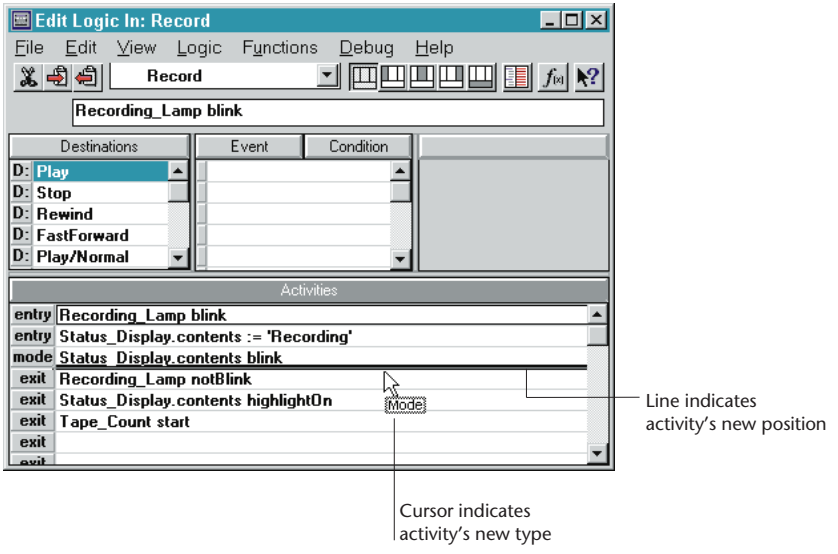
There are two ways to change an activity's type:

- With the Activities column popup menu.
- Dragging the activity to a different section (e.g., from the entry section to the mode section).

To change an activity's type:

1. Select the activity.
2. Either:
   Right-click and choose an activity type from the popup menu. The activity becomes the **last** activity of the chosen type.

   Or:
   Drag the activity to a new location. As you drag the activity, a line follows the cursor up and down the column indicating the activity's new position. In addition, the cursor itself indicates the activity's new type. Dragging lines is describes in more detail on p. 12-33.



Order of Execution of Actions and Activities

When an application is running in the Prototyper and a transition is triggered, Rapid performs the actions associated with the trigger in the sequence in which they appear in the Logic Editor (first Action 1, then Action 2, etc.)

In the same way, Rapid performs the activities of the active mode(s) in the sequence in which they appear in the Logic Editor:

- When a mode is entered, Rapid first performs the entry activities (in the order Entry Activity1, then Entry Activity 2, then Entry Activity 3, etc.)
- Rapid then performs the mode activities (in the order Mode Activity 1, then Mode Activity 2, then Mode Activity 3, etc.) If a mode activity has one or more arguments, the mode activity will be recalculated if any of the arguments changes while the mode is active.
- When the mode is exited, Rapid performs the exit activities.

The precise order of execution of transitions and activities is discussed in more detail in Chapter 16: "Rapid Logic: Hints and Tips."

MANIPULATING LINES OF LOGIC

This section provides step-by-step instructions for manipulating logic lines:

- Cutting, copying and pasting lines
- Moving (dragging) lines
- Inserting and deleting lines

Cutting, Copying and Pasting

You can cut and copy lines of logic (transitions, triggers, actions and activities) within a mode and paste them somewhere else in the same mode or in another mode. The cut, copy and paste functions allow you to:

- Change the order of logic within a mode.
- Move, or copy, logic from one mode to another.
- Make actions into activities and vice versa.

❖ *NOTE: To change the order of logic or to exchange actions and activities, within the same mode, it is often easier to move lines by dragging them (see p. 12-33).*

Cutting lines

Cutting a line removes it from its column and stores it in Rapid's internal memory. If you cut a transition destination, its triggers and actions are also cut. If you cut a trigger, its actions are cut as well. You cannot cut lines from more than one column at a time.

To cut a line:



1. Select the line. (See p. 12-12 for details on selecting multiple lines.)
2. Choose Edit|Cut.

SHORTCUT:
Right-click and choose Cut from the popup menu.

❖ *NOTE: If you mistakenly cut a line, you can recover it by pasting (if you have not subsequently cut or copied anything from that column).*

Copying lines

Copying a line copies it to Rapid's internal memory without removing it. If you copy a transition destination, its triggers and actions are also copied. If you copy a trigger, its actions are copied as well. You cannot copy lines from more than one column at a time.

To copy a line:


Ctrl+C

1. Select the line. (See p. 12-12 for details on selecting multiple lines.)
2. Choose Edit|Copy.

   SHORTCUT:
   Right-click and choose Copy from the popup menu.

Pasting lines

You can paste an item that has been copied to the Rapid internal memory. The pasted line replaces the selected line in the target column.

To paste a line:


Ctrl+V

1. Select the line.
2. Choose Edit|Paste.

   SHORTCUT:
   Right-click and choose Paste from the popup menu.

Pasting activities of different types

If you copied or cut more than one activity of different types (for example, three mode activities and two exit activities), you can paste them all while retaining their original types.

To paste an activity (which retains its type):

1. Select the lines.
2. Right-click and choose Paste by Type from the popup menu. (This menu command is not available in the Edit menu.)
3. The lines are pasted as the last activities of their respective types.

Cutting, copying and pasting in the Logic Editor

When copying, cutting and pasting logic lines, bear in mind the following points:

- Cutting/copying a line in one column does not affect a line that was cut/copied in a different column (i.e., each column has its own paste buffer). For example, you can copy a trigger and then copy an activity, without overwriting the copied trigger.
- The only columns whose contents are "interchangeable" are the Actions and Activities columns (i.e., they share the same paste buffer). For example, you can paste an action into the activities column, but you **cannot** paste a trigger into the actions column.
- Cutting or copying a transition destination also cuts/copies the associated triggers and actions, which will be pasted with the destination.
- Cutting or copying a trigger also cuts/copies the associated actions, which will be pasted with the trigger.
- If you cut a transition destination that has triggers and actions, you must paste it into the Destinations column. You cannot paste just the triggers or actions.

Moving Lines

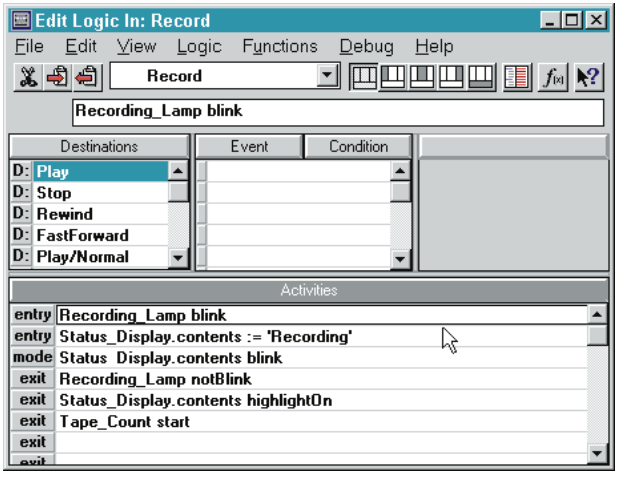
You can drag the lines of logic up and down the columns and, in some cases, from one column to another. This allows you to:

- Change the order of a column's logic.
- Quickly duplicate lines of logic.
- Make actions into activities and vice versa.

Dragging lines

To drag a line:

1. Select the line(s).
2. Move the mouse cursor to the top or bottom edge of one of the selected lines; the cursor changes from the cross to the regular arrow, indicating that you can drag the line:



3. Drag the line to a new location. As you drag, a bold line follows the cursor up and down the column indicating the logic line's new position. (When you drag activities, the cursor also indicates the activity's new type.)
4. When the line is positioned correctly, release the mouse button.

Duplicating lines

To duplicate a line:

- Press the Ctrl key while dragging the line to a new location.

Exchanging between activities and actions

The contents of the activities and actions columns are interchangeable. That is, you can drag an activity into the actions column, and vice versa.

Inserting and Deleting Lines

Inserting new lines

There are several situations in which you might want to insert new lines:

- When you want to insert a new line of logic between two existing lines.

- When you want to add a blank line between lines of logic in order to improve readability (e.g., a blank line between entry and mode activities).
- When you have used all the available lines in a column and want to add more.

To insert a new line:


Insert

- Either:
  Select a line, then choose Edit|Insert. A line is inserted **above** the selected line.

  Or:
  Click the column's title button.

Deleting logic lines

Deleting a line removes the logic from the application; the line is not placed in the internal memory. If you delete a transition destination, its triggers and actions are also deleted. If you delete a trigger, its actions are deleted as well.

To delete a line:


Delete

1. Select the line(s).
2. Choose Edit|Delete.

   Or:
   Right-click and choose Delete from the popup menu.

MANUALLY TYPING THE LOGIC

While building your application's triggers, actions and activities, there are two methods for entering the logic:

- "Point & Click"—the conventional way of selecting objects, properties and functions in the Logic Palette and appending them to the edit line.
- Manually typing the entire logic phrase into the edit line.

The typing method is suitable for those who are experienced with Rapid's logic and its syntax and feel that they can work more quickly with the keyboard rather than with the mouse.

To enter logic by typing:

1. Select a blank line in one of the columns.
2. Type the logic and press Enter. The logic is confirmed and a new line is inserted.

Adding Comments to Logic

You can add comments (”remarks”) to an application’s logic to help clarify its behavior. Comments can make your application’s logic more understandable and easier to maintain. A comment can be:

- Part of a line of logic (must be enclosed in double-quotes). For example,

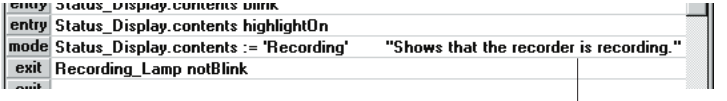
  Display2.contents := TapeCounter.value “Displays the current counter value”

- A whole line to itself (must begin with a double backslash). For example,

  These activities occur in play mode.

To add a comment to a line of logic:

1. Select the line.
2. In the edit line, add the comment to the logic. The comment must be enclosed in double-quotes:

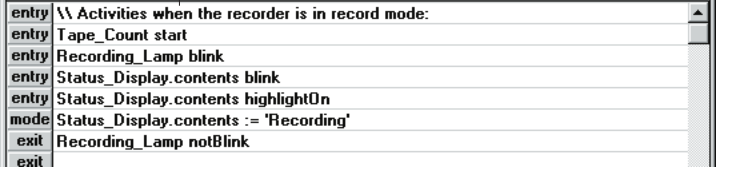


Comment added to the end of a line

To make a whole line a comment:

1. Select the line.
2. Type a double backslash (\\), followed by the comment:

Comment as a line to itself



Temporarily Removing Logic ("Remarking Out")

You can temporarily remove triggers, activities or actions from an application by adding a double backslash at the beginning of each logic line. By doing so, you are making the logic line a comment (see "Adding Comments to Logic" on previous page).

Temporarily removing logic is especially useful during debugging.

To temporarily remove logic:

1. Select the line.
2. In the edit line, insert the cursor at the start of the line and type "\\". Examples of "remarked out" logic are:

   \\lineObject.end setX: 15
   \\& Timer is running

To restore the logic:

1. Select the line.
2. In the edit line, delete "\\".

VERIFICATION TEST

The verification test allows you to perform a basic check on any of the following aspects of an application's logic design:

- Modes: checks for modes which cannot be reached or exited.
- Transitions: checks for absent triggers.
- Objects: checks for unreferenced objects.

At the end of the test, a list of detected logic problems are displayed and you can then debug them one by one.

Not all the generated warnings are an indication of error since Rapid cannot always distinguish between an actual problem and an intentional feature of the application design.

❖ *CAUTION! The verification test is not exhaustive and may miss potential problem areas, particularly in the case of a complex application.*

Running the Verification Test

You run the verification test, set its options and view the logic warnings in the Verification Test window.

To open the Verification Test window:

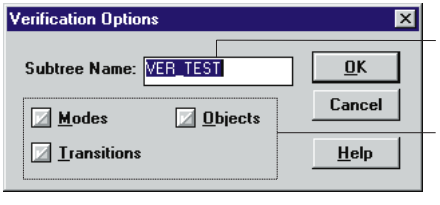
Ctrl+V

- In the Application Manager, choose Tools|Verification Test.

To set the options for the test:

Ctrl+O

1. Choose Options|Options to open the Verification Options dialog box.



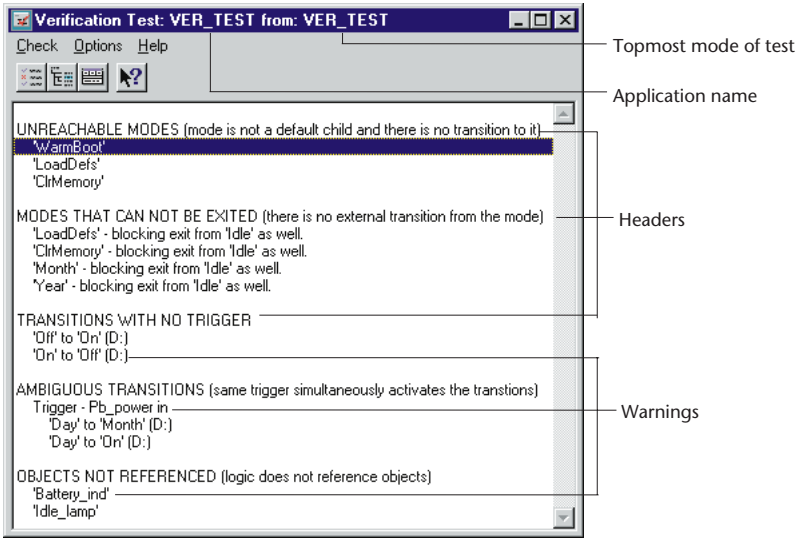
2. When the root mode is selected, Rapid checks the entire application. If you only want to check a subtree, select the mode in the mode tree or type the mode's name in the text box.
3. Click the appropriate check boxes to deselect any verification test categories.

To start the test:



- Choose Check|Check or click the Start Test button.

When Rapid has checked all selected categories, the logic warnings (if there are detected problems) appear in the Verification Test window, as shown below:



Examining Logic Errors

To examine errors:

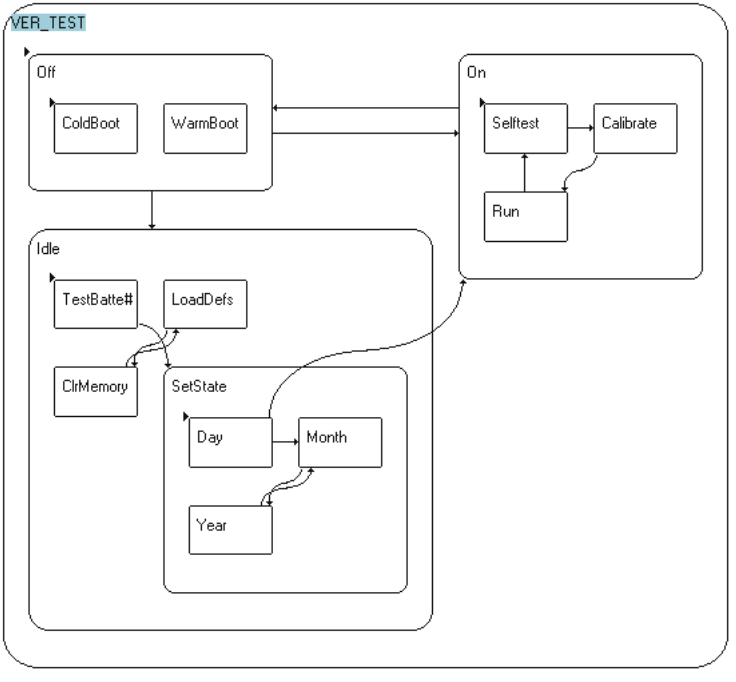
Select the warning in the window that you want to examine, then use the Check menu commands or toolbar buttons as follows:

COMMAND	DESCRIPTION
 *Edit (Ctrl+E)*	Selects (in the mode tree and the state chart) the mode where the error was detected. Opens the Logic Editor. Use this option when you want to investigate the warning in more detail. **SHORTCUT**: Double-click the warning in the window.

COMMAND	DESCRIPTION
Select (Ctrl+L)	Selects (in the mode tree and the state chart) the mode where the error was detected. Use this option if you just want to identify the referenced mode.

Warning Messages

This section describes the warning messages that can appear in the Verification Test window. The warnings are grouped according to their categories: mode, transition and object warnings. The State Chart shown below will help to explain some of the mode warnings. (All modes shown are exclusive.)



Modes

WARNING SYNTAX	DESCRIPTION OF CHECK
UNREACHABLE MODES '<mode>' '<mode>'	This check looks at the application's modes and transition destinations, and finds modes which cannot be reached. (That is, modes not referenced by an indirect or direct transition destination.) *NOTE: This check does not test triggers.* In the VER_TEST state chart shown above, modes which cannot be reached are: LoadDefs and ClrMemory.
MODES THAT CAN NOT BE EXITED '<mode>' '<mode>' - blocking exit from ' <parent mode>' as well.	This check tests the modes and transition destinations in order to find any mode which, once entered, cannot be exited. A mode is regarded as "exitable" if there is a transition from it (or from one of its ancestors or reachable descendants) to a destination that is **outside** the mode's own subtree. In the VER_TEST state chart, there is only one mode (Day, the default child of SetState) which has a transition external to the Idle subtree. Thus, the modes LoadDefs, ClrMemory, Month and Year, once entered, cannot be exited. ***NOTES**: Since TestBattery (the default child of Idle) has a valid transition to SetState (whose default child, Day, is exitable), it is not considered a mode which cannot be exited.* *If a child mode can not be exited, the warning takes note of the fact that the exit from the child's parent is also blocked.*

Transitions

WARNING SYNTAX	DESCRIPTION OF CHECK
TRANSITIONS WITH NO TRIGGER '<source mode>' to '<destination mode' (<transition type>)	This check tests all the transitions' triggers, and issues a warning for any transition that has no trigger (i.e., the transition can never take place.) In the VER_TEST application warnings, for example, you see that no triggers have been defined for the transitions between the Off and On modes. Although they show in the state chart, they cannot take place.
AMBIGUOUS TRANSITIONS Trigger - <trigger syntax> '<source mode1>' to '<destination mode1' (<transition type>) '<source mode2>' to '<destination mode2' (<transition type>)	This check tests all the transitions' triggers and issues a warning for any trigger which simultaneously activates more than one transition. In the VER_TEST application warnings, for example, you see that the same trigger activates both transitions out of Day mode. During runtime, when Day mode is active and Pb_power is pressed, Rapid activates the first transition in the list.

Objects

WARNING SYNTAX	DESCRIPTION OF CHECK
OBJECTS NOT REFERENCED <object> <object>	This check tests each object and issues a warning if an object is not referenced by any action, activity or trigger. In the VER_TEST application warnings, for example, you see that the objects Battery_ind and Idle_lamp are not referenced anywhere in the logic.

❖ *NOTE: When performing the mode or object checks, Rapid occasionally discovers internal inconsistencies in the mode or object tree structure that may cause display problems. In these cases, Rapid alerts you to the situation and requests permission to correct the problem, which involves some restructuring of the tree.*

CHAPTER 13

How to Use Object Logic

Objects in the Object Layout have no effect on your application's behavior unless they are involved in its logic. This chapter describes how to use specific properties and functions of certain objects.

For a description of every function of every object, you should refer to the *Object Reference Manual*. In this chapter, general functions that are common to many objects are explained, and the more complex objects are covered in greater detail.

In this chapter you will read about the following objects:

- Active objects (general functions common to all active objects)
- Dynamic objects' functions

NONGRAPHIC OBJECTS:

- String
- Number and integer
- Random number
- Data store
- Array (general)
- Holder and array of objects
- Time
- Date
- Event object
- Keyboard
- Mouse and cursor

GRAPHIC OBJECTS:

- Barrel Dial
- Pointer
- Sector Dial
- Switches
- Joystick
- Animated object
- Pipe, valve and tank objects

ACTIVE OBJECTS (GENERAL)

There are several functions that are common to all active objects (i.e., all named, graphic objects): *disable*, *enable*, *hide* and *show*. These functions allow you to disable (and enable) and hide (and show) an object. In addition, objects with children have the *hideWithoutChildren* and *showWithoutChildren* functions.

Several conditions test whether an object is hidden or enabled, or whether the object is in the vicinity of another object.

Disabling an Object

The *disable* function "neutralizes" the object (without hiding it) until it receives the *enable* function. During this time, the object does not respond to any click or logic. If it is drag 'n drop, you cannot drag it; if it is a switch, you cannot connect it to a new position, etc.

When the object is enabled, Rapid ignores any action or activity that was performed on the object while it was disabled. For example, in the following series of actions a switch is connected to one position, then it is disabled and connected to a new position:

Switch1.position2 connect
Switch1 disable
Switch1.position3 connect
Switch1 enable

After the switch is enabled, it does **not** move to position3.

❖ *NOTE: Conditions that were true before the object was disabled remain true while it is disabled.*

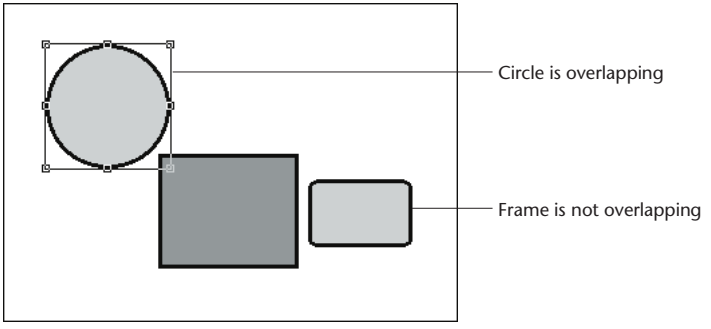
Testing an Object's Position in Relation to Another

Every active object, whether or not it is dynamic or drag 'n drop, has two conditions that test the object's position in relation to another active object:

is overlapping:

The *is overlapping:* condition tests whether the object's enclosing rectangle is overlapping another (specified) object's enclosing rectangle. The overlap can be as little as one pixel, and the condition is true whether the object is above or below the other object.

❖ *NOTE: Any non-rectangular object's enclosing rectangle extends beyond the object's bitmap.*



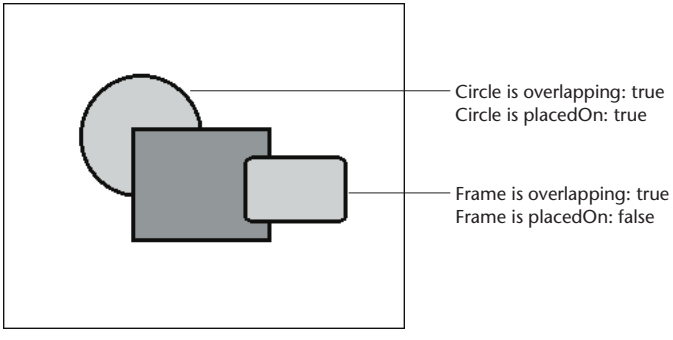
Examples of Syntax

& Circle is overlapping: Frame
& Switch is overlapping: Pushbutton

is placedOn:

The *is placedOn:* condition tests whether the **center** of the object is within another (specified) object's enclosing rectangle. The condition is true whether the object is above or below the other object.

From the definition of *is overlapping:*, you can see that whenever *is placedOn:* is true, *is overlapping:* is also true.



Examples of Syntax

& Circle is placedOn: Frame
& Switch is placedOn: Pushbutton

DYNAMIC FUNCTIONS

When you make an active object dynamic, it acquires properties that allow you to change its position in the application's logic.

When you make a primitive object dynamic, it acquires properties that allow you to change its size, position and color in the application's logic.

Changing an Object's Position or Size

Rapid gives an object's position in the form of a coordinate, where 0@0 is the upper-left corner of the Object Layout work area. Most dynamic objects' size and position properties are *size* and *position*; circular objects' size and position properties are *centerPosition* and *radius*, respectively. Dynamic lines have the *start* and *end* properties that govern the line's length and angle (see "Dynamic Lines" on p. 13–8).

When working with coordinates in the Logic Editor, keep the following points in mind:

- Rapid uses the @ sign to separate x and y coordinates.
- The x-coordinate increases towards the right and the y-coordinate increases towards the bottom, each with a maximum determined by the size of the layout you have defined in the Object Layout.
- Rapid coordinates are measured in the current application's scale units.
- If an expression (i.e., anything more complex than a single number) is used for the x- or y-coordinates, then the coordinates must be enclosed in parentheses. For example:

  (Integer+ 5) @ (Integer - 10)

Examples of using coordinates are given below.

Assigning values to an object's position or size

To assign values to individual coordinates:

Use the *setX:* or *setY:* functions. For example:

Circle.centerPosition setX: 25

You can also use the *x* and *y* functions for the same effect (you need to manually enter the *:=* function):

Circle.centerPosition x := 25

To assign coordinate values to other position/size properties:

Frame.position setX: Circle.centerPosition x

To move an object up by 20 pixels and to the right by 10 pixels:

Object.position := Object.position + (10 @ -20)

or

Object.position changeBy: 10 @ -20

The above := and *changeBy* statements perform identically. If used as an entry or exit activity, or as an action, the object will move once and then stop. Mode activities, however, are constantly re-evaluated, so the object will change its position continuously. In other words, the object will move up and to the right as long as the mode is active.

The following activities place "Object1" 50 units to the right and 30 units above "Object2":

Object1.position := Object2.position + (50 @ –30)

or

Object1.position changeXBy: 50 yBy: -30

or

Object1.position := (Object2.position x + 50) @ (Object2.position y – 30)

Changing an object's position or size with a potentiometer

You can use the value of an object's property to change another object's position, for example, a potentiometer's value:

Object.position := Pot1.value @ Pot1.value

This activity causes the object to move upwards or downwards at an angle of 45° as the potentiometer is adjusted. No parentheses are required here.

Circle.radius := Pot1.value

This activity causes the circle to shrink or grow as the potentiometer is adjusted.

To assign double the value of a potentiometer to the x-coordinate and the value of the potentiometer to the y-coordinate:

Object.position := Pot1.value * 2 @ Pot1.value

Conditions that Test Coordinates

When creating a trigger, you can use conditions that test the value of an object's position or size. The standard relational operators (i.e., =, >, <, >=, <> and <=) are available for use with coordinates.

When comparing one coordinate, X1 @ Y1 with another coordinate, X2 @ Y2, the relational operator separately compares X1 with X2 and Y1 with Y2. The result is true only if it is true for both parts of the coordinate. For example, the condition

& X1 @ Y1 <= X2 @ Y2

is exactly the same as the condition

& (X1 <= X2) and (Y1 <= Y2)

You can also test the x or y values separately. For example, to find whether "Object1" is above "Object2," use the condition:

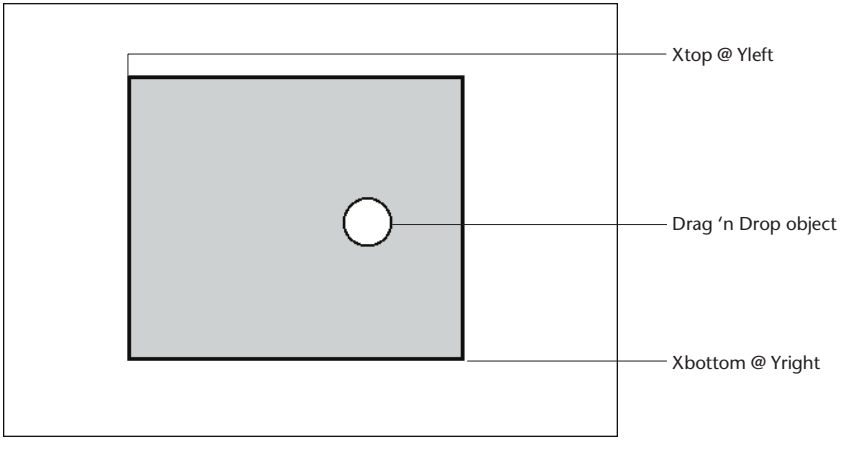
& Object1.position y < Object2.position y

Setting a boundary

The following condition checks whether an object is outside a certain area on the screen. The area is bounded by the points (Xtop @ Yleft) and (Xbottom @ Yright), where Xtop, Yleft, Xbottom and Yright are integer objects:

& Object1.position x < Xtop or Object1.position x > Xbottom or Object1.position y < Yleft or Object1.position y > Yright

This condition would trigger a transition if the object moved outside the area.



Changing a Primitive Object's Colors

A dynamic primitive object has properties that allow you to change two color elements in the application's logic:

PROPERTY	DESCRIPTION
fillColor	Has functions that allow you to change the object's fill color (if any). The fill color is the color of the main body of the object (usually light gray).
lineColor	Has functions that allow you to change the object's line color. The line color is the color of the object's outline (usually black).

Most of these properties' functions are simple commands that change the color element to a specific color.

Examples of Syntax

Circle.lineColor darkRed
Frame.fillColor yellow

Changing a color's RGB number

There is one function that allows you to set a specific color using the RGB (red-green-blue) color model: *setRed:green:blue:*. In the RGB color model, various brightness values of red, green, and blue light combine to form the colors on the screen. The range of colors in the visible spectrum is represented by controlling the intensities of the individual RGB components.

The range of values for each color channel is 0-255, giving a theoretical maximum of over 16 million colors (256^3). Not all video graphic cards enable the display of this many colors; some only enable the display of 16 colors, others enable 256. A 256-color monitor can display the colors that are present in the current Windows palette. If you assign an RGB value that corresponds to a color not present in the current palette, Rapid maps that value to the nearest corresponding color.

If your graphic card enables the display of 16 million colors ("true color"), Rapid displays the precise RGB color that you set. The syntax for setting an RGB color is

<object name>.<color property> setRed: <integer> green: <integer> blue: <integer>

For example:

Circle.fillColor setRed: 102 green: 204 blue: 153



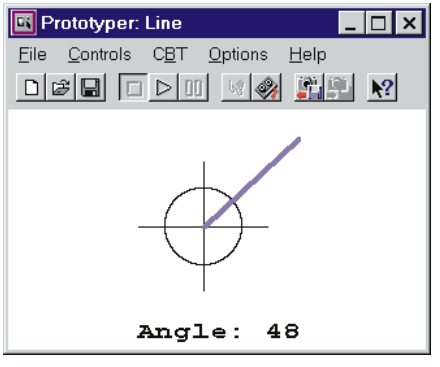
Dynamic Lines

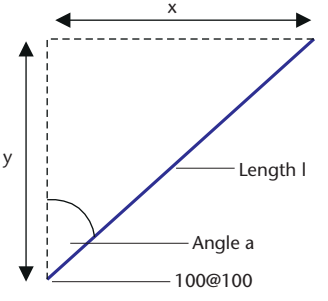
A dynamic line object has *start* and *end* properties that enable you to control the position of their "start" and "end" points. These properties allow each end of the line to move independently. You can use these properties to provide a form of animation. When defining a line in the Object Layout, the values of the "start" and "end" points are only initial values that can be changed in the logic.

It is important to note that the "start" of the line is the first point on which you click, when you draw the line in the Object Layout. The "end" of the line is where you release the mouse button after drawing. Each of these ends retains its name ("start" or "end") no matter how their physical positions change in relation to one another.

The *start* and *end* properties have various arithmetic functions and behave as coordinates. Values for the *start* and *end* properties must have the form *x@y*, where *x* is the x-coordinate of the start/end position and *y* is the y-coordinate.

You can use trigonometric functions to achieve the effect of a line rotating around a fixed point, such as in a clock or a dial:





If the end of the line is fixed at 100@100 (at the center of the circle), then:

x = 100 + l sin a
y = 100 - l cos a

where l is the length of the line and a is the angle from the vertical. You can build three mode activities that cause the line to rotate:

```
Line.start x := 100 + (Length * ((Angle degreesToRadians) sin))
Line.start y := 100 + (Length * ((Angle degreesToRadians) cos))
Angle changeBy: 1
```

See "Degrees and Radians" on p. 13–15 for details about *degreesToRadians*.



STRINGS

The string object is one of Rapid's basic data objects. It contains a collection of alphanumeric characters. The string object has just one property, *self*, which has many functions that allow you to manipulate the string. Some functions change the string itself; most of them return a new string value based on the original value, which remains unchanged.

The following pages describe some of the more complex string functions. For a description of all string functions, refer to the *Object Reference* manual.

❖ *NOTE: Apart from the string object, other objects make use of string properties. The value of the text display's contents property, for example, is a string.*

"Padding" Strings with Spaces

Two functions allow you to return a new string based on the original one, with the addition of spaces, either at the beginning or end of the string:

- *padSpacesLeftTo:*
- *padSpacesRightTo*:

These functions return a new string of the specified length by adding spaces to the left (or to the right) of the original string. If you specify a shorter string than the original string, the new string will be truncated.

The syntax for this function is:

<string> padSpacesLeftTo: <integer>

Example:

The string shown in the text display below contains 10 characters:



The following actions are carried out:

```
String := Display.contents padSpacesLeftTo: 15
Display.contents := String
```

The above syntax takes the display's contents and returns a string of 15 characters, 5 spaces being added at the left. The new string is then displayed:



As you can see, when the string appears in a display, it is always pushed to the right. Similarly, if you use the *padSpacesRightTo:* function, the string is always pushed to the left.

Changing Individual String Characters

You can address an individual character within a string using square brackets. The following example addresses the fifth character of the string:

String [5]

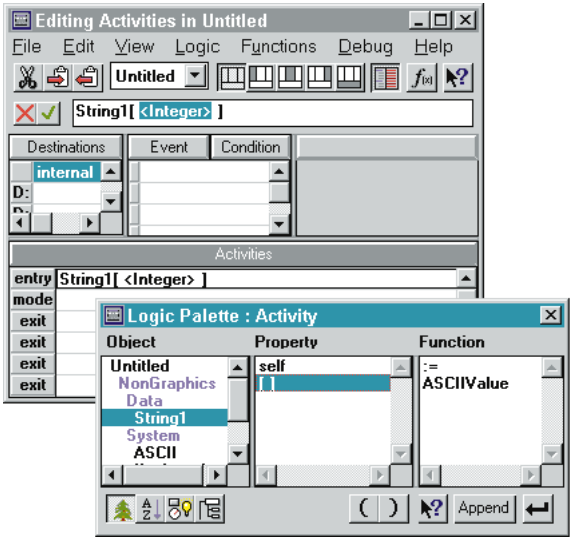
Use this syntax as the basis for returning or changing individual string characters. This method is similar to that used for addressing an array index.

There are also several functions, such as *changeASCIIBy:at:* and *insert:at:* that directly address individual characters, or a range of characters, within a string.

To address a string character:

1. Select the string object from the Logic Palette's object list.
2. Select the *[ ]* property from the property list.
3. If you want to use one of the functions (:= or *ASCIIValue*), select it (details of the individual functions are given below) and click Append. If not, click Append without selecting a function.

   The object-property-function sequence is copied to the edit line:



4. In the edit line, overtype <integer> with the integer that gives the character's position within the string (e.g., *5* for the fifth character).

:=

The := function assigns a particular value to the specified string character.

This value may be one of:

- A character, such as the letter "t."

  The following example assigns "t" to the sixth character of the string:

  String1 [*6*] := 't'

  You can only assign one character; *String1 [6] := 'abc'* will only assign "a."

- A character from another string. For example:

  String1 [*6*] := String2 [10]

- A character from the contents of a text display or text window object. For example:

  String1 [*6*] := Disp.contents [3]

- A character with a specified ASCII value (using the nongraphic ASCII object). For example:

  String1 [*6*] := ASCII withValue: *169*

ASCIIValue

The *ASCIIValue* function returns the ASCII value of the character. For example:

Display.contents := String [*6*] ASCIIValue

The above action carried out on the string "EMULTEK" will display "69" (the ASCII value of "E" is 69).

Determining the Format of a String

The *is formattedAs:* condition allows you to test whether the string object is made up of a particular format ("regular expressions"). Use *is formattedAs:* in the form "if the string is formatted as 'format', then perform action A," or, "if the string is not formatted as 'format', then perform action B."

The following syntax gives an outline of the use of *is formattedAs:*

& String is formattedAs: '<string>'

where <string> defines the pattern you want to compare to the string.

The format string must be enclosed in single quotes and the following characters have special meanings:

. [ ] - * ? + ^

SYMBOL	DESCRIPTION
\	A backslash (\) followed by a special character, matches the special character itself. For example *'Pardon\?'* matches "Pardon?".
.	Matches any single character except a new line character. For example *'.ive'* would match "jive" and "five."
[ ]	The *[* and *]* special characters are used to denote one-character regular expressions that will match any of the characters within the brackets. For example, *'[aeiou]'* would match either "a," "e," "i," "o," or "u." A string enclosed in square brackets denotes a set and tells Rapid to search for any character in that string, but no others. The characters ?, +, *, and . do not have any special meaning when used in a set.
-	The - special character is used within the [ ] special characters to denote a range (a "set") of characters to match. For example, *'[a-z]'* would match any lowercase alphabetic character between a and z. Dashes can be combined to form expressions like *'[a-bd-z]'* to match any letter except "c."
^	The ^ special character is used within a set to specify a search for any character except those specified. For example, *'[^g-v]'* would match any character **not** between g and v. This character only takes on its special meaning if it immediately follows the opening square bracket.

SYMBOL	DESCRIPTION
+	The + special character following a regular expression matches one or more occurrences of that expression. For example, *'[a-z]+'* matches one or more occurrences of lowercase alphabetic characters.
?	The *?* special character specifies that the previous character is optional. For example *'xy?z'* matches "xz" or "xyz."
*	The * special character following a regular expression matches zero or more occurrences of that expression. For example, *'[a-z]*'* matches zero or more occurrences of lowercase alphabetic characters.



NUMBERS AND INTEGERS

The number and integer objects are two of Rapid's basic data objects (along with string and point). The number object can only contain float-type numbers, that is, numbers containing a decimal point. The integer object can only contain integers. These objects have just one property, *self*, which has many functions that allow you to manipulate the objects' values. Some functions change the value itself, although most of them return a new value based on the original value (which remains unchanged).

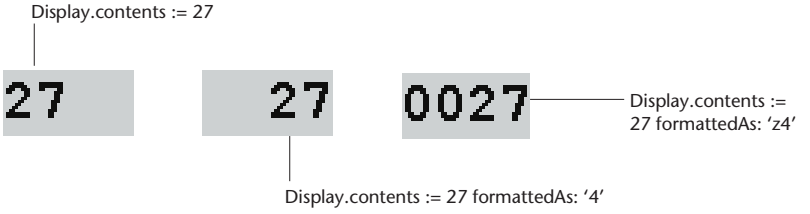
Rapid automatically converts numbers to integers and vice versa, whenever necessary. For example, *Number := 5* assigns 5.0 to the number object, while *Integer := 4.7* assigns 4 to the integer object.

As well as basic arithmetic operations, the number and integer objects' functions include more advanced operations such as *ln* (returns the natural log) and trigonometric functions. For a description of all arithmetic functions, refer to the *Object Reference* manual.

Formatting a Number as a String

The number, integer and random number objects have the *formattedAs:* function that converts each of them to a string value, formatted according to a format template. This function is useful when you want to align numbers to the right (e.g., in an LCD display), when you want to display a number with leading zeros (e.g., seconds in a clock display), and so on.

The difference between displaying regular numbers and displaying formatted numbers is illustrated below.



❖ *NOTE: If you display a number in a text display in Right entry mode, the numbers are reversed.*

The template is a string and has the syntax ***zm.net***, where any of its elements is optional:

- ***m*** specifies the number of digits to the left of the decimal point. For example:

  Display1 := Integer formattedAs: '4'.

  This activity ensures that the integer is always displayed as a 4-character string. If the integer is less than 1,000, the first characters are occupied by spaces. If the value of the integer is 27, the result of the above action is "<space><space>27".

  ❖ *NOTE: If the value is negative (e.g., -27), the resulting string is "-<space><space>27".*

- ***n*** specifies the number of decimal places. For example:

  String1 := Number formattedAs: '4.2'.

  This activity ensures that the string always has two decimal places. If the value of the number is 427.3, the result of the above action is "<space>427.30".

  If the number cannot fit within the *m* and *n* constraints, asterisks are generated. For example:

  String1 := Number formattedAs: '2.2'.

  If the value of the number is 427.3, the "427" part of the number cannot fit within the two digits specified and the result of the above action is "**.**".

- If ***e*** is specified, the resulting string is presented in scientific notation. *t* specifies the number of decimal places; if *t* is not specified, 8 decimal places are included. For example:

  String1 := Number formattedAs: '2.2e2'.

If the value of the number is 427.3, the result of the above action is "4.27e+02" (i.e., $4.27x10^2$).

- ***z*** causes generation of leading zeros. For example:

  String1 := Number formattedAs: 'z4.2'.

  This activity ensures that the string has leading zeros, so that the number of digits before the decimal point is always 4. If the value of the number is 27.3, the result of the above action is "0027.30".

  If the number is too big to be accommodated by the format, asterisks are printed. If is too small to be accommodated by the format, zeros are printed.

Degrees and Radians

When Rapid calculates the sin, cosine and tangent values of numeric values, it assumes that the values are in radians and not in degrees. For example the activity, *Number := 90 sin*, returns 0.894 and not 1.0. The *degreestoRadians* function is available to convert degrees to radians for Rapid's trigonometric functions. The previous activity would read:

Number := (90 degreesToRadians) sin

This activity returns 1.0.

Similarly, when Rapid returns the arcsin, arccos or arctan of a numeric value, it returns the result in radians. For example, the activity *Number := 1 arcSin* returns 1.57 (radians) and not 90 (degrees). The *radiansToDegree*s function is available to convert radians to degrees for Rapid's trigonometric functions. The activity would read:

Number := (1 arcSin) radiansToDegrees

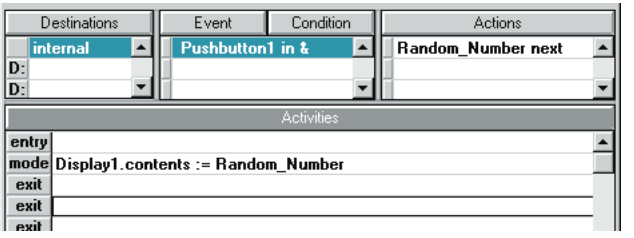
This activity returns 90.0.



RANDOM NUMBERS

The random number object is a read-only number (not integer) that is assigned a random value by Rapid whenever you start the Prototyper. The random number has up to six decimal places. It has all of the functions of the number object, except those that directly change the object's value, such as := or *changeBy:*, since Rapid itself assigns the values.

You can instruct Rapid to regenerate the random number's value by using the *next* function. A typical sequence using the random number is shown in the Logic Editor below:



Every time the pushbutton is pressed in, Rapid generates a new random value (*Random_Number next*). The mode activity constantly displays the current random number in a text display.

Generating a Random Integer

The random value that Rapid generates is a number. The two parts of the number, though—the whole number part and the decimal part—are calculated separately. For example, Rapid first randomly generates 6, and then randomly generates 0. 368935; the resulting random number is 6.368935.

This means that if you assign a random number value to an integer object, the integer will be just as random as the number:

Integer := RandomNumber

If you wanted to display a random integer, you could display the integer object that was used above. Alternatively, you could make use of the random number object's *truncated* function, as in the following mode activity:

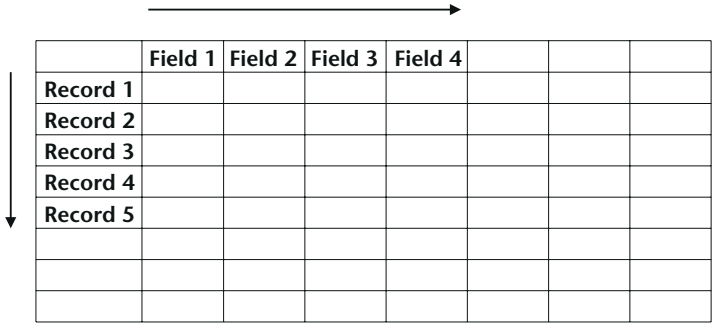
Display.contents := RandomNumber truncated

Random integers allow you to generate all kinds of random parameters in your applications. For example, you can generate random strings by assigning random integers to ASCII values; you can generate random colors by using random integers with the *setRed:green:blue:* function.



DATA STORES

The data store object stores two-dimensional structures of strings, integers and/or numbers. The structure of a data store can be compared to that of a table with varying numbers of columns and rows, as illustrated below:

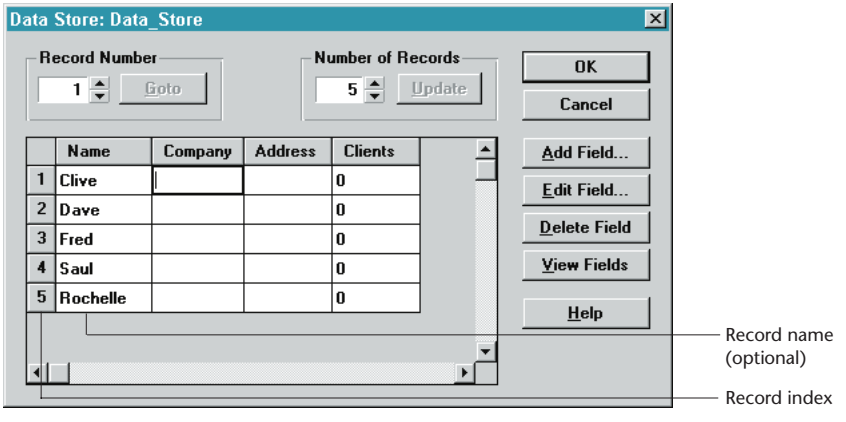


Schematic representation of data store

Records, Fields and Elements

A data store is comprised of records. Each record is identified in one or two ways:

- Record index—an integer that defines the position of that record within the data store—1, 2, 3, etc. Every record has an index.
- Record name—the (optional) name that you assigned to the record in the Object Layout's Data Store dialog box:



Each record is comprised of a number of fields, each of which can contain either a string, integer or number value. A specific field of a record is known as an element (equivalent to a table cell).

Syntax for Addressing a Data Store

In Rapid syntax, you can address a data store element using either the record index or the record name.

Referring to the record index

The syntax for addressing a data store element using the record index is:

DataStore [<integer>].<field name>

where <integer> is the record index. For example, in the data store dialog box shown above, the syntax for addressing Rochelle's address (record 5) is:

DataStore [5].Address

❖ *NOTE: The period (.) between the record and the field is an integral part of the syntax.*

You do not have to directly insert the actual record index. You can also use an integer object (or an expression that yields an integer) to address the data store. For example, the following address uses the number object "indexObject" to reference the "Address" field:

DataStore [IndexObject].Address

If the current value of IndexObject is 17, the expression references DataStore's 17th record; if the current value of IndexObject is 13, it references the 13th record, and so on.

You can also incorporate mathematical expressions, as in the following example:

DataStore [IndexObject + 11].Clients

If the current value of IndexObject is 4, this expression references the 15th record (4+11).

Referring to the record name

The syntax for addressing a data store element using the record name (if it has one) is:

DataStore ['<string>'].<field name>

where <string> is the record index. For example, in the data store dialog box shown above, the syntax for addressing Rochelle's address (record 5) is:

DataStore ['Rochelle'].Address

Note that the record name is a string and has to be enclosed in single quotes.

Assigning Values to Data Store Records, Fields and Elements

Overwriting an entire data store with another

You can overwrite the entire contents of one data store with the contents of another data store, as long as the data stores are compatible—that is, they have the same number and type of fields. The syntax is as follows:

<data store> := <data store>

For example:

DataStore1 := DataStore2

Overwriting an entire record with another

You can overwrite the entire contents of one data store record with the contents of another record, as long as the records are from compatible data stores. The syntax is as follows:

<data store> [<integer or string>] := <data store> [<integer or string>]

Where <integer> is an expression that yields a record index and <string> is a record name. For example:

DataStore1 [4] := DataStore2 ['Fred']

Assigning values to a data store element

The syntax for assigning values to a data store element is:

<data store> [<integer or string>].<field> := <number or integer or string>

The following examples all assign values to a data store element:

Customers [17].Owing := 1000

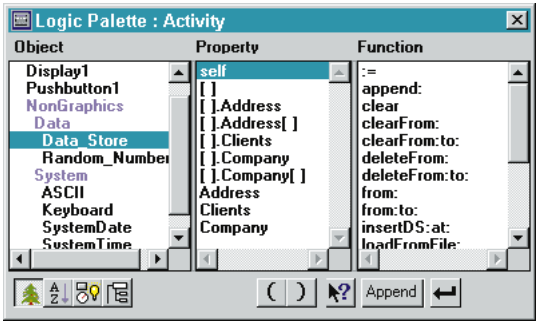
Customers ['John'].Address := 'London'

Customers [IntegerObject * 2].Sales := (Customers [25].Sales * 2)

Customers [10].Name := TextDisplay.contents

To assign a value to a data store element:

- Select the Data Store from the Logic Palette's object list. The Logic Palette looks something like:



The data store shown above has three fields:

- Address and Company (string fields)
- Clients (an integer field)

The options offered in the property list are:

PROPERTY	DESCRIPTION
self	For manipulating the entire data store (adding records, deleting records, etc.)
[ ]	For assigning values of one record to another.
[ ].<string field>	For manipulating individual elements within the data store.
[ ].<string field>[ ]	For manipulating individual characters within specific data store elements (string fields only).
[ ].<number or integer field>	For manipulating individual elements within the data store.
<field>	For manipulating entire fields (assigning values, performing searches and queries, etc.)

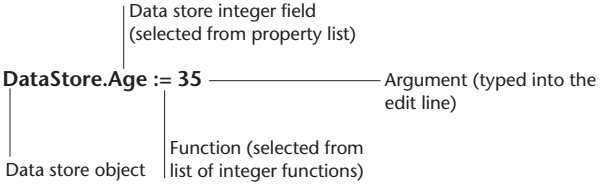
The method of using the *[ ]* properties to address individual records is similar to that used to address individual characters in a string (p. 13–10) and elements in an array (p. 13–26).

Assigning a value to a data store field (for all records)

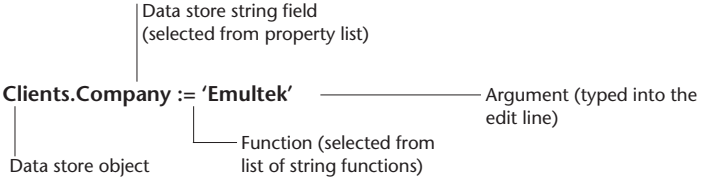
The data store fields are properties of the data store. Any functions that you specify for the field property are carried out on all the records in the data store. The syntax is as follows:

<data store>.<field> := <string or integer or number>

Rapid automatically converts the argument to the type necessary for the field. For example, if you assign a number to an integer field, Rapid truncates the number. The following example assigns the value 35 to the "Age" field of all records of a data store:



In the same way, if you had a string field called "Company" for a data store called "Clients," you could construct the following expression:



Saving and Loading Data Stores

The functions *saveToFile:* and *loadFromFile:* enable you to build activities or actions that save or retrieve data stores as ASCII text files. They allow you to load different initial values at the start of your application—values which you saved on a previous occasion. You are also able to load the data store into a different application in order to perform different operations on the data.

❖ *NOTE: Saving the data store saves the data* ***and*** *the structure of the store. You can only load a data store from one application to another if the second application already has a compatible data store, that is, it must have the same number and type of fields and records.*

Saving a data store

The syntax for saving a data store to disk is:

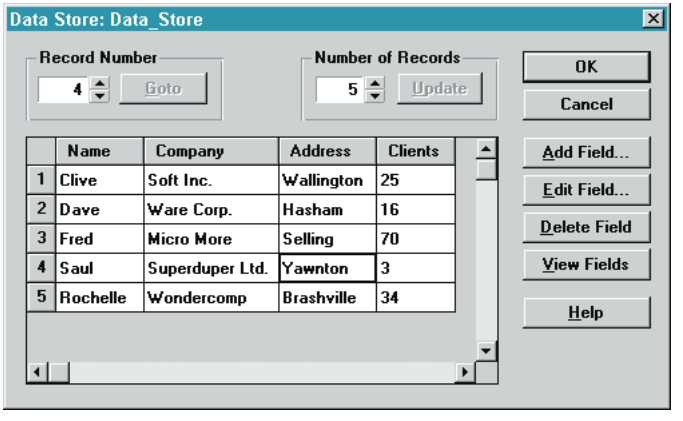
<data store> saveToFile: 'Pathname'

where 'Pathname' is the full pathname of the file to which you want to save the data store. Rapid assigns the default extension *.rds* (Rapid Data Store). If you do not specify a path, Rapid saves the data store to the application folder. Like any other string, the path must be enclosed by single quotation marks. For example:

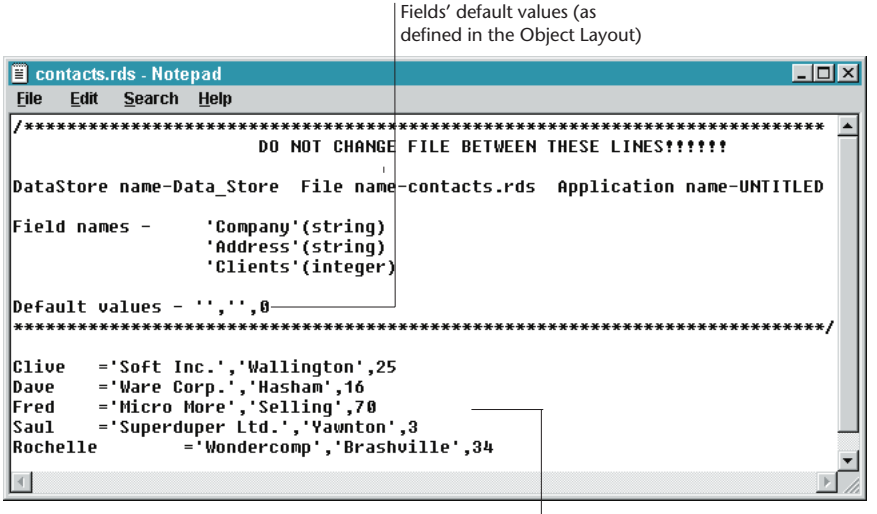
DataStore1 saveToFile: 'contacts'

Editing a data store file

Once the data store is saved, you can view and edit it in any text editor, such as Windows Notepad. The illustrations below show the dialog box of a data store, followed by the text file that it generates:



Dialog box of a data store...



...and its resulting text file.

You can edit the data store, being careful not to alter anything between the two lines at the top of the file. When editing the data, note that the fields are separated by commas; do not delete them!

❖ *NOTE: Saving a data store to a file which already exists, automatically overwrites the existing file (without a warning).*

Loading a data store

The syntax for loading a data store file from disk is:

<data store> loadFromFile: 'Pathname'

where 'Pathname' is the full pathname of the file which you want to load into the application. The path must be enclosed by single quotation marks. If you only specify the filename without the full path, Rapid searches for the file in the following folders:

- The application folder (where the current application is located).
- The Rapid folder (where *rapid35.exe* is located).
- The system search path.

❖ *NOTE: The values of a loaded data store file replace the current values of the data store into which it is loaded.*

Removing Data Store Records

Although the number of fields of a data store remains fixed, you can add and delete records.

Deleting records

The *deleteFrom:to:* function deletes a specified range of records, thereby shortening the data store. If you want to delete just one record, both arguments should be equal. For example, the following activity deletes record 3:

```
DataStore1 deleteFrom:3 to:3
```

The following activity deletes records 4 to 7 (inclusive):

```
DataStore1 deleteFrom:4 to:7
```

The *deleteFrom:* function deletes all records from the specified record to the end of the data store.

pullFrom:to: function

This function removes all the records between the specified points. The records below the "pulled" section move up to close the gap. A series of new records is created at the foot of the store. The new records contain the initial values that were set when you defined the data store in the Object Layout. This function does **not** shorten the data store.

❖ *NOTE: The second specified record (following the "to:") is not removed.*

The diagram below illustrates a data store made up of ten records (1 to 10). The following function is carried out on the data store:

```
DataStore1 pullFrom:4 to:7
```

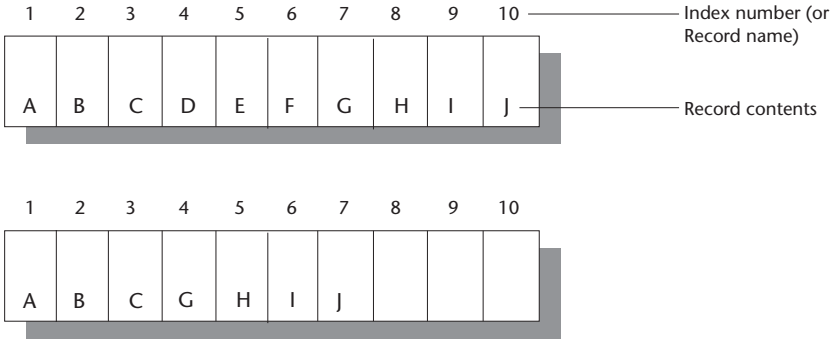


Illustration of the *pullFrom:to:* function

The contents of records 7 through 10 have been pulled up to the specified record. Records 8, 9 and 10 now contain the initial value, and the contents D through F have been deleted.

❖ *NOTE: This function also applies to one-dimensional arrays.*

Using Data Store Values in Conditions

There are several levels at which you can apply relational functions to data store values in order to construct condition triggers.

- You can compare an entire data store to another (compatible) data store and use the result to trigger a transition. The syntax is as follows:

  & DataStore1 = DataStore2

- You can compare one data store element to another and use the result to trigger a transition. For example, you could construct the condition:

  & DataStore1 [1].Age > DataStore1 [2].Age

  which would return True if the Age field value in record 1 of DataStore1 is greater than the Age field value in record 2 of the same data store.



ARRAYS

The array object allows you to store multi-dimensional arrays of either integers, numbers, strings or objects. The logic of arrays of objects is discussed in "Holders and Arrays of Objects," on p. 13-29.

❖ *NOTE: Many of the array's functions are only applicable to one-dimensional arrays. Refer to the Object Reference manual for full details.*

Syntax for an array's index

When you create an array in the Object Layout, you define how many dimensions it has and how many elements there are in each dimension. In the Logic Editor, an individual array element is indexed as follows:

Array [d1e , d2e , ...dne]

where "d1e" is the element in dimension 1, "d2e" is the element in dimension 2 and so on up to dimension n (maximum = 7). The element coordinates are separated by commas (,) and every dimension in the array has to be specified in your logic phrase for the index to be valid.

Overwriting an entire array with another

You can overwrite the entire contents of one array with the contents of another, as long as the two arrays have the same structure. The syntax is:

Array1 := Array2

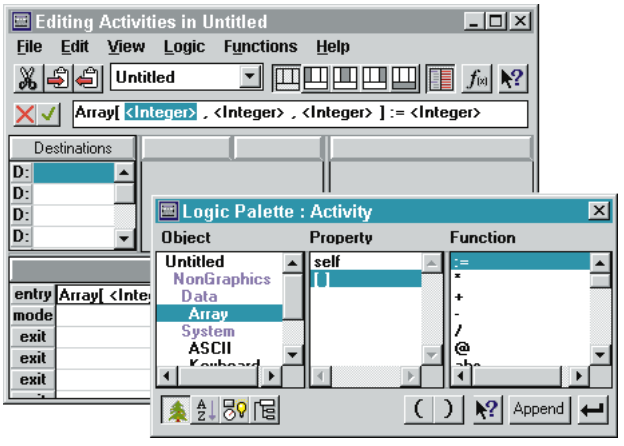
Addressing an array element

The following is a typical example for an integer array with five dimensions:

Array1 [1 , 3 , 2 , 2 , 4] := 10

To address an array's element:

1. In the Logic Palette, select the array from the object list and the *[ ]* property. Select a function, if necessary, and click Append:



2. Overwrite the first <integer> prompt with the element's first index.
3. Press Tab to highlight the next prompt, and enter the second index.
4. Repeat this procedure until all the dimensions have been indexed. (Make sure that you don't delete the commas.)

QUERIES & SEARCHES IN DATA STORES AND ARRAYS

The data store and array objects have two sets of functions that allow you to carry out queries and searches within the objects' data:

- Instances functions—return the number of times a specified value occurs in a given range.
- Search functions—search for the first occurrence of a specified value over a given range, and return the element's index.

❖ *NOTE: Most of these functions do not apply to multi-dimensional arrays.*

Instances Functions

- In data stores—return the number of times that a specified value occurs within a particular field, over a range of records.
- In one-dimensional arrays—return the number of times that a value occurs, over a range of elements.

- In multi-dimensional arrays—you cannot perform queries over a range, only over the entire array.

  Examples:

  Array instancesOf: 26.5

  Returns the number of times the value 26.5 occurs within the entire array.

  DataStore.Name instancesOf: 'John' from: 4 to: 17

  Returns the number of times the value "John" occurs within the data store's Name field, over the range Record 4–Record 17 (inclusive).

Search Functions

- In data stores—return the first record number where the specified value occurred, over a range of records.
- In one-dimensional arrays—return the first array element where the specified value occurred, over a range of elements.
- In multi-dimensional arrays—you **cannot** perform searches in a multi-dimensional array.

  Examples:

  Array searchFor: 26.5

  Returns the first array element where the value 26.5 occurs.

  DataStore.Name searchFor: 'John' from: 4 to: 17

  Returns the first record number where the value 'John' occurs, in the range Record 4–Record 17 (inclusive).

  DataStore.Name searchNextTo: 22

  Repeats the last search, up to Record 22.



HOLDERS AND ARRAYS OF OBJECTS

Holder objects are used for "holding" active objects, active primitive objects and groups. They are useful in applications containing several objects of the same type that use the same logic. Instead of writing the logic for each object, you can write the logic once for the holder and then instruct the holder to hold each object in turn. A holder possesses the properties and functions of the object being held as well as its own functions.

Each element in an array of objects is itself a holder. An array can only hold one type of object, thus all elements in an array can only hold one specific type of object. Arrays of objects allow you to address objects by referring to array indexes (made up of integers) rather than to specific objects.

Example:

You could create an array of lamp objects called LampArray, each element of which would hold a different lamp used by the application. Element 1 might hold a round red lamp, Element 2 a square green lamp, Element 3 a round yellow lamp, and so on. You can hide all the lamps when the application starts with one entry activity:

LampArray hide

During runtime, you can activate any one of the lamps in the array by addressing its element number. For example, you could display a blinking red lamp:

LampArray [1] show
LampArray [1] blink

Holders and arrays of objects can dynamically generate objects in runtime by making a copy of the object they hold. Such generated objects can be deleted during runtime by a suitable holder or array.

As well as possessing the properties and functions of the object being held, the holder and the array of objects have several particular functions of their own.

Holding Specific Objects

The function *hold:* tells the holder or array element to hold a specific object. For example, if Holder1 is initially defined to hold RockerSwitch1, the following would be a valid action/activity:

Holder1 hold: RockerSwitch2

If Array1 can hold round lamps, this would be a valid action/activity:

Array1 [4] hold: Lamp2

Generating Objects

The holder and array of objects are able to generate objects in runtime by holding copies of objects. The function that performs this is *holdCopyOf:*. The *holdCopyOf:* function is useful if you need several identical objects, but each one only for a short period of time. For example, dynamic primitive objects that appears only briefly (such as moving targets or radar blips).

The generated object is positioned at the initial position of the copied object (as defined in the Object Layout) and in front of any other objects. When it is generated, the new object is **hidden**; use the *show* function to see the object.

For example, if CrcArray is defined to hold circle objects, the following activity generates a copy of the object Circle1:

```
CrcArray [5] holdCopyOf: Circle1
```

Holders and elements of arrays of objects can only generate one copy at a time; if they hold a different object or generate a new copy, the first copy is deleted.

Arrays of objects also have the *addMore:copiesOf:* function that increases the size of the array by adding a specified number of copies of an object, without releasing any objects it is currently holding. For example, if CrcArray currently holds four circle objects, the activity:

```
CrcArray addMore: 3 copiesOf: Circle1
```

increases the size of the array to seven elements. Elements 5, 6 and 7 each hold a copy of Circle1.

The Rapid CD-ROM contains several example applications that demonstrate how to use an array of objects to generate and manipulate objects dynamically during runtime. You will find them in the \Examples\Arr_obj directory.

"Releasing" Objects and Deleting Generated Objects

The *clear* function tells the holder or array element to clear its current contents. If the object that was held was a real object, it is simply "released." If the object was a generated object, it is deleted.

❖ *NOTE: If a generated object is held by more than one holder or array element, it is not deleted until it is released by all of them.*

Events of Holders and Arrays of Objects

A holder or array element has all the events of the object that is holds. For example, to detect when the pushbutton held in the 7th element of an array is pushed in:

```
PushbuttonArray [7] in &
```

You can also detect when **any** of the pushbuttons are pushed in:

```
PushbuttonArray in &
```

Whenever an object in an array generates an event, its index is assigned to *lastEventIndex*, a read-only integer that is a function of all arrays of objects. This function allows you to address an object that generated an event without knowing which specific object it was.

For example, when the pushbutton held in the 7th element of an array is pushed in, *lastEventIndex* is assigned the value 7. You could then clear that element (without knowing beforehand which pushbutton is going to be pushed) by addressing it as follows:

```
Array1 [ Array1 lastEventIndex ] clear
```



TIME

The value of the time object is treated as an integer value. Therefore, all the arithmetic functions as described for the integer object's *self* property are valid for each of the time object's properties: *hours*, *minutes* and *seconds*.

Assigning values to the time object

Unlike an integer, though, the value of one of the time object's properties can directly affect the value of another. For example, when the *seconds* property reaches the value of 60, it returns to zero and the *minutes* property increases by one. Similarly, when the *minutes* property reaches the value of 60, it returns to zero and the *hours* property increases by one.

The *hours* property is similarly bounded, depending on how you defined the time object in the Object Layout: 12-hour, 24-hour or unlimited. If the object has a 12-hour format, the *hours* property will revert to one after reaching 12. If it has a 24-hour format, the *hours* property will revert to zero after reaching 23. If it has unlimited format, the *hours* property continues to increase (as in a stopwatch).



DATE

The value of the date object is treated as an integer value. Therefore, all the arithmetic functions as described for the integer object's *self* property are valid for each of the date object's properties: *day*, *month* and *year*.

Assigning values to the date object

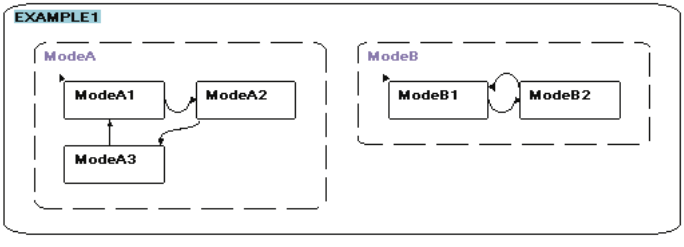
Unlike an integer, though, the value of one of the date object's properties can directly affect the value of another. For example, when the *month* property reaches the value of 12, it returns to one and the *year* property increases by one. Similarly, when the *day* property reaches the maximum value (depending on the month), it returns to one and the *month* property increases by one.



EVENT OBJECT

The event object is used to simplify Rapid logic when designing complex applications. For example, you can generate the event in one mode, and then use the event to trigger a transition in another active mode.

The illustration below shows an example application where the event object can be useful. The application has two concurrent modes, ModeA and ModeB. They both have a number of exclusive child modes with various transitions between them.



The logic behind the event object takes the form: "When trigger A is fulfilled, trigger the event; when the event is triggered, perform action B."

Trigger A may be a combination of events and conditions; associating the trigger with an event object avoids the need to repeat it in several locations.

Action B may be a whole list of actions; associating the list with an event object avoids the need to repeat it in several locations.

In addition, the event object can be used as an entry or exit activity in one mode to trigger an action in another active mode.

As an Activity to Trigger a Remote Action

Problem

In the application illustrated above, you want an action to occur in ModeB2 when (and only when) ModeA3 is entered.

Solution without an event object:

There is no complete solution! The nearest solution without an event object is to construct the following internal trigger in ModeB2:

& mode/ModeA3 is active

This will trigger the action in ModeB2 when ModeA3 becomes active. However, it will also trigger the action if the application exits ModeB2, enters ModeB1, and returns to ModeB2.

Solution with an event object

1. In the Object Layout, add an event object (“Event1”).
2. In the Mode Tree, select ModeB2, then open the Logic Editor and the Logic Palette.
3. In the object list, select the event object and build the following **internal** trigger for the mode:

   event1 triggered &
4. Build an action for this trigger in the Actions column.
5. Select ModeA3 and construct the following entry activity:

   event1 trigger

What’s the logic?

When the application enters ModeA3, it triggers the event object. When the Event object is triggered, it in turn triggers the action in ModeB2.

To Avoid Repeating Identical Lists of Actions

Problem

In ModeA2, you want a series of actions to take place when a certain internal trigger occurs. These actions include a lamp lighting, a text display updating, a calculation on a number being performed, etc. You also want the same series of actions to take place in ModeA3 and in ModeB1—each with its own, different trigger.

Solution without an event object:

The only method is to build the list of actions for one of the modes and then to copy and paste the logic into the correct position in each of the other modes. If this method is needed for long lists of actions, or for many different modes, it adds extra "bulk" to your application and may slow it down.

Solution with an event object:

1. In the Object Editor, add an event object ("Event1").
2. In the Mode Tree, select the root mode (here "EXAMPLE1"), then open the Logic Editor and the Logic Palette.
3. In the object list, select the event object and build the following **internal** trigger for the root mode:

   event1 triggered &
4. Select the new trigger that you have just made and, in the Actions column, construct the list of actions that you want to occur.
5. In ModeA2, construct the relevant internal condition and add the following action:

   event1 trigger
6. Copy and paste this action to the relevant places in modes ModeA3 and ModeB1.

What's the logic?

When the internal trigger in ModeA2 occurs, it triggers the event object in the root mode. When the event object is triggered, it in turn triggers the list of actions. The same actions happen whenever the event is triggered. Instead of the long list of actions appearing in several places, it appears just once, whereas the event trigger and action appear in each mode.

To Avoid Repeating Complex Conditions

Problem

In ModeA2, you want an action to take place when a certain complex trigger occurs. This trigger includes a combination of numerical conditions, positions of moving objects, etc. You also use the same trigger in ModeA3 and in ModeB1 to activate different actions.

Solution without an event object

The only method is to build the trigger for one of the modes and then to copy and paste it into the correct position in each of the other modes. If it is a complex trigger, it adds extra "bulk" to your application and may slow it down.

Solution with an event object:

1. In the Object Editor, add an event object.
2. In the Mode Tree, select the root mode (here "EXAMPLE1"), then build the complex trigger.
3. Select the trigger that you just made and construct the following action:

   event1 trigger
4. In ModeA2, build the following **internal** trigger:

   event1:triggered &
5. In the Actions column, build the action that is to occur when the complex condition is fulfilled (i.e., when the event object is triggered).
6. Repeat steps 4 and 5 for modes ModeA3 and ModeB1.

KEYBOARD

The nongraphic keyboard object is one of Rapid's system objects that is automatically included in every application.

The keyboard object allows you to enter keyboard data into the application as it runs in the Prototyper. The data is treated as string data and can be displayed in a text display, etc. Apart from entering simple text, you can use the keyboard object to:

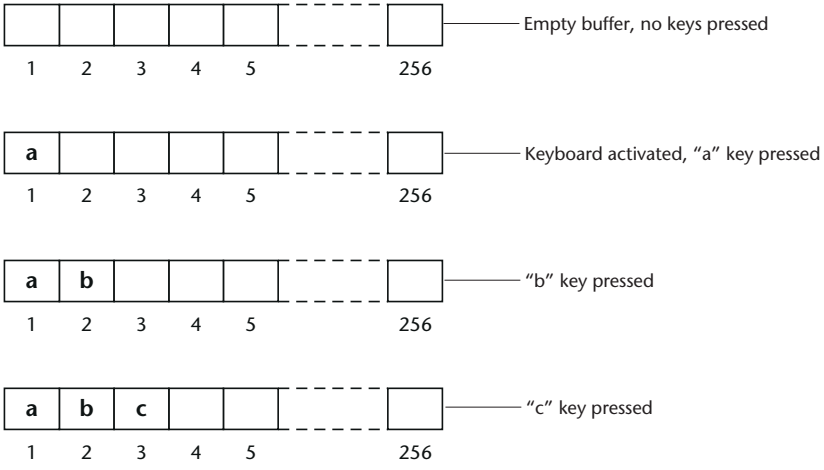
- Detect when the Ctrl or Shift keys have been pressed.
- Detect, in conjunction with the ASCII object, when various ASCII codes have been typed, such as a space or a tab.

Principles of Operation

Before you begin using the keyboard object, you must activate it with the activate function. It is usual to include the activate function in an entry activity of the mode in which you use the keyboard.

❖ *NOTE: If you forget to activate the keyboard object, Rapid will not respond to your key presses.*

Once the keyboard is activated, Rapid places every character that you type into an area of memory known as the keyboard buffer. The buffer can contain a maximum of 256 characters. Two functions, *character* and *getCharacter*, allow you to return the first character that entered the buffer, as string data. You can then use that data as with any other string.



Keyboard buffer

What's the difference between *character* and *getCharacter*?

getCharacter returns the first character that entered the buffer and deletes it from the buffer. *character* returns the first character but does not delete it. The significance of this difference is explained in the following section.

Displaying Text with the Keyboard Object

This section explains how to display data entered with the keyboard in a text display object. The keyboard object has an event, *keyPress*, that is generated each time a key is pressed (providing that the keyboard is activated, of course). Use this event to trigger an action that updates the display's contents. The appropriate logic is shown below.

Entry Activity:

Keyboard activate

Internal Transition:

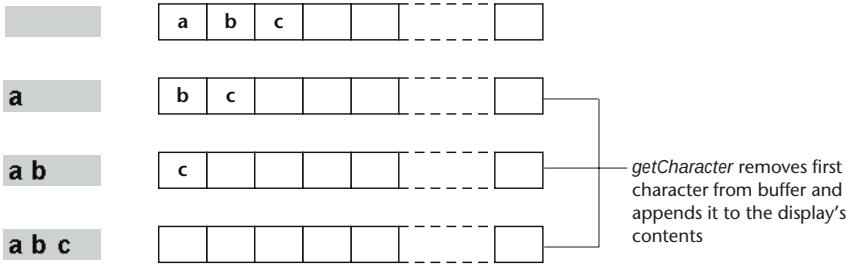
Trigger: **Keyboard keyPress &**
Action: **Display.contents append: Keyboard getCharacter**

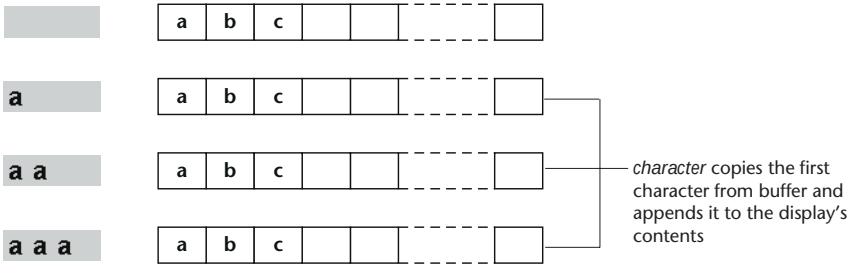
What's the logic?

When the mode is entered, the keyboard is activated (the entry activity). Whenever a key is pressed Rapid places the character in the buffer and the event is generated (*Keyboard keyPress &*). When the event is generated, Rapid removes the character from the buffer (*Keyboard getCharacter*) and appends it to the text display's contents.

Note that if you use the *character* function instead of *getCharacter*, Rapid does not remove the character from the buffer and, on every key press, appends the same character to the display (no matter what key was pressed).

The illustrations below demonstrate the difference between displaying text using *getCharacter* and *character*. For the sake of the example, the keyboard buffer initially contains three characters.





The difference between these two functions means that *character* is ideal for use as a condition when you want to test the value of the buffer character without removing it. For general keyboard entry, use *getCharacter*.

Testing for the Control and Shift Keys

The conditions *is controlKey* and *is shiftKey* test whether the Ctrl or Shift key has been pressed together with any other key. These conditions can be used to trigger a transition. For example, the logic below causes a lamp to light whenever the Ctrl key is pressed in conjunction with any other key:

Trigger: **is controlKey &**
Action: **Lamp on**

The keyboard object and Rapid shortcut keys

When you run an application in the Prototyper, Rapid shortcut keys, such as Ctrl+N (New Application), **override** the *is controlKey* condition. If you run this application in the Rapid Reviewer (that does not have such shortcut keys), the condition behaves as normal.

MOUSE AND CURSOR

The nongraphic mouse object is a system object which (1) tracks the mouse position and events and (2) assigns cursor shapes to graphic objects. Like all Rapid system objects, it is automatically included in every application and is listed under the System group in the Objects list of the Logic Palette.

When used to assign cursor shapes to graphic objects, the mouse object works in conjunction with a cursor object. There are two types of cursor objects:

- System: Rapid provides thirteen pre-defined cursor objects which appear automatically under the System group in the Logic Palette.
- User: You can define custom cursor objects in the Object Layout, as described on p. 5-26, “Cursor”. If you have defined user cursor objects, they appear under the Signal group in the Logic Palette.

In this section, you will read about:

- Activating and deactivating the mouse object.

- Using an event generated when a cursor enters or exits an active graphic object's bounding box.
- Enabling/disabling the *mouseMove* event.
- How Rapid determines cursor appearance in the case of overlapping graphic objects for which cursors shapes have been defined.
- Setting and changing the cursor position.
- Specifying how frequently a mouse event recurs (i.e., using the *autoRepeat* property of the mouse object).

Activating/Deactivating the Mouse Object

To use the mouse object functions in your Rapid application, you must first activate it in the Logic Palette. To disable the mouse object's functions, you must deactivate the mouse object.

❖ *NOTE: It is recommended that you activate the mouse only when you intend to use it. Keep it deactivated at all other times. Activating the mouse when it is not needed may unnecessarily slow down your application.*

To activate or deactivate the mouse object:

1. Select Mouse from the System group in the Object list of the Logic Palette.
2. Select *activate* or *deactivate* in the Functions list.

Using the *cursorEntered* and *cursorExited* Events

When the mouse object is activated, Rapid automatically generates a *cursorEntered* or *cursorExited* event when the cursor enters or exits the bounding box of a graphic object. You can use these events as triggers in your application logic.

For example, you may want to pop up an instructional message each time the user passes the cursor over a specific object. You would use that object's *cursorEntered* event to trigger the action of showing the message.

Enabling/disabling the *mouseMove* event

The mouse object generates the event *mouseMove* each time the mouse changes position during runtime. In order to optimize your application's performance, however, this event is disabled by default.

To be able to use the *mouseMove* event as a trigger in your logic, you must first use the function *enableMouseMoveEvent* as an entry activity or an action. Once enabled, you can use the function *disableMouseMoveEvent* to return to the default situation.

The condition *is mouseMoveEventEnabled* returns TRUE if the *enableMouseMoveEvent* function has been executed.

Defining cursor shapes

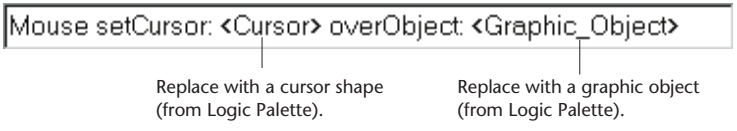
You specify the cursor shape for use during runtime by using a cursor object (either user or system) as an argument in mouse object functions. In this way you can:

- Set the cursor for a graphic object, a switch position, a group, a user-defined object (UDO) or the root object.
- Reset the cursor shape to the default shape.

❖ *NOTE: If you have not specifically defined a cursor shape for the root object, the default shape is your system default cursor.*

Setting a cursor shape for a graphic object

You use the mouse object's *setCursor: overObject:* function to define a cursor shape for a graphic object (including a group, a UDO, or the root object) in the Logic Editor. The syntax in the edit line of the Logic Editor is as follows:

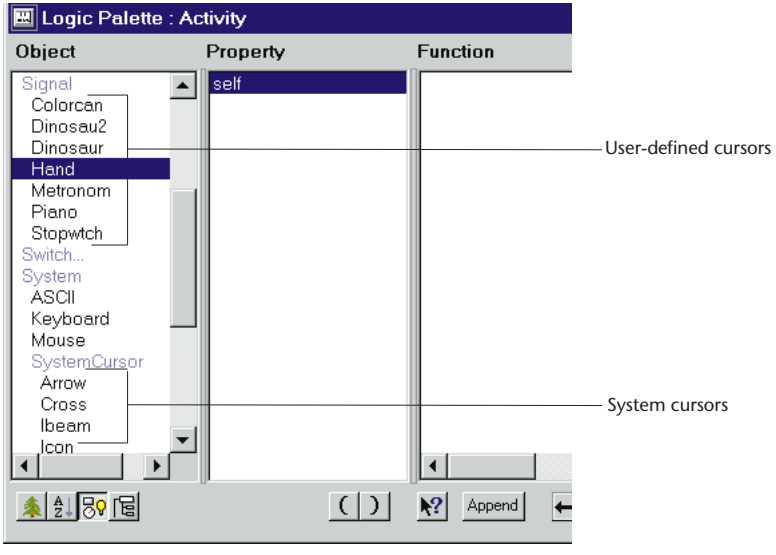


❖ *NOTE: In order for this logic statement to take effect during runtime, don't forget that you first have to activate the mouse object.*

Cursors in the Logic Palette

System cursors are found in the SystemCursor group (within the System group) of the Logic Palette object list. Do not define these cursors in the Object Layout; they are automatically available for use in the Logic Palette.

Custom (user-defined) *cursors* (from *.cur*, *.exe*, *.ani*, or *.dll* files) are found in the Signal group in the Logic Palette object list.



Example:

You have defined a hand-shaped user cursor named “Hand" which is to be displayed when the cursor is placed over a text display object named “NewSubject”. In the Logic Editor, the logic would look as follows:

Mouse setCursor: Hand overObject: NewSubject

Resetting an object’s cursor to the default shape

Use the mouse object’s *resetCursorOverObject:* function to reset a graphic object’s cursor shape to the default shape.

❖ *NOTE: The default is either the cursor set for the root object, or your system default cursor.*

Designating a cursor over a switch position

Rapid provides special functions for (re)setting a cursor over a switch object's active area. These functions are very similar to those for the other objects, discussed above.

```
Mouse setCursor: <Cursor> overSwitchPos: <Switch_Position>
```

Replace with a cursor (from Logic Palette).

Replace with a switch position.

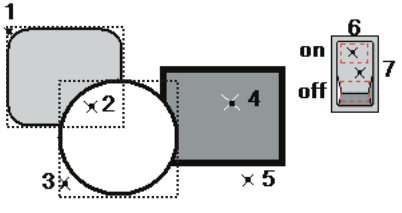
Cursor Priorities

There could be runtime situations where the mouse is positioned over several overlapping graphic objects, each of which has been assigned a different cursor. Rapid sets an order of priority that determines the cursor appearance should such a conflict occur. The table below gives the order of priority for various types of conflicts.

❖ *NOTE: Cursors can be designated for a UDO or a group of objects just as they may for a single graphic object. A UDO and a group of objects have the same priority.*

If the cursor is located at an overlap of:	Then the cursor is that of:
graphic objects	the topmost graphic object
UDOs and/or groups of objects	the bottom-most UDO or group of objects
a graphic object over a UDO or a group to which it does not belong	the graphic object

Example:



Dashed frames show the active areas of the switch positions.

Dotted frames show the bounding boxes of non-rectangular objects.

Suppose each of the four objects in the above figure has a defined cursor. The table below lists the cursor that will appear, when the mouse is at a given position.

For point number 5, “background” means one of the following areas:

- the root object
- the general area of a UDO, i.e., the area in the UDO which is either uncovered or on which none of its graphic objects has a defined cursor.
- the general area of a group of objects, i.e., the area in the group which is either uncovered or on which none of its own objects has a defined cursor.

Point Number(s) in Figure	Active Cursor Shape Designated for ...
1	rounded corners frame
2 and 3	circle
4	frame
5	the background (see bulleted explanation above)
6	“on” position of Rocker Switch
7	Rocker Switch

Setting and Changing Cursor Position

You use the *position* property of the mouse object to change or set the position of the cursor as it appears in the Prototyper window. The cursor position is a point property expressed in Rapid *x,y* coordinates and is relative to the root object of your application or your UDO. See “Point” on p. 5-20, for further information.

The cursor is automatically placed at (0,0) at the start of activation. Rapid keeps track of the cursor position as the mouse is moved inside the Prototyper window.

To set/change cursor position:

1. Select the mouse object’s *position* property in the Logic Palette.

2. Select the desired function in the Functions list. See “Point Functions” in the *Object Reference* for an explanation of all the functions in this list.

❖ *NOTE*: *If the value of the x or y coordinate is less than zero, it is automatically raised to zero.*

If the x or y coordinate value is ***greater*** *than the extent of the root object frame, it is automatically changed to a value equal to a point* ***on*** *the root object frame.*

Specifying Mouse Event AutoRepeat Frequency

You use the *autoRepeatPeriod* property of the mouse object to specify how fast a mouse event is repeated when pressing and holding down a mouse button at runtime.

❖ *NOTE: For a description of the mouse object events, see the Object Reference Manual.*

The *autoRepeatPeriod* property is an integer property whose value, in milliseconds (ms), represents how fast Rapid regenerates a mouse event. Its default value is 0, meaning no autorepeat.

Below is a table summarizing the values this property takes on, depending on your input.

If you input ...	The event is repeated every ...
0 or less	0 ms, i.e., no AutoRepeat
greater than 0 and less than 1000	Nearest multiple of 50, rounded up, in ms
1000 or greater	1000 ms

Examples

If you enter 79 for the AutoRepeatPeriod, the mouse object will trigger a leftButtonDown event every 100 milliseconds whenever you press and hold down the mouse’s left button at runtime. If you enter 283, the AutoRepeatPeriod will be rounded up to 300.

To set the autoRepeat period:

- Assign an integer to the mouse’s *autoRepeatperiod* property.

Example:

Suppose you want to display how many seconds you have held down the right mouse button. Let the number of seconds be displayed in a display object called “NumberBox”. Use the following logic and syntax:

TRIGGER (internal transition)	ACTION
Mouse rightButtonDown&	Counter changeBy: 1

ACTIVITY TYPE	ACTIVITY
entry	Mouse.autoRepeatPeriod:= 1000
mode	NumberBox.contents:= Counter



BARREL DIAL

The values you set in the dialog box determine how the barrel dial appears and how it rotates when the Prototyper is started. Through the logic, you can:

- Start the rotation (in either an upward or downward direction).
- Stop the rotation.
- Set the barrel dial's current value, rotation rate and colors.
- Connect two or more barrel dials so that the rotation of one affects the rotation of the other.

Starting and Stopping the Rotation

A barrel dial does not rotate unless you use either the *rotateUp* or *rotateDown* function. *rotateUp* causes the values to increase; *rotateDown* causes the values to decrease. The barrel dial rotates continuously until it is stopped (using the *rotateStop* function).

Direction of the rotation

The direction in which the barrel dial rotates depends on (a) its orientation setting (in the Object Layout dialog box) and (b) on the rotation function (in the Logic Editor).

OBJECT LAYOUT ORIENTATION	LOGIC EDITOR FUNCTION	ROTATION EFFECT
Up	*rotateUp*	The numbers increase in an upward direction (i.e., from the bottom to the top)
Up	*rotateDown*	The numbers decrease in a downward direction (i.e., from the top to the bottom)
Down	*rotateUp*	The numbers increase in a downward direction.

OBJECT LAYOUT ORIENTATION	LOGIC EDITOR FUNCTION	ROTATION EFFECT
Down	*rotateDown*	The numbers decrease in an upward direction.

Stopping and restarting the rotation

If you stop the rotation (using the *rotateStop* function), then restart it (using either *rotateUp* or *rotateDown*), the rotation begins at the initial value.

Setting the Current Value

You can use the *currentValue* property to assign a value to the barrel dial. The value you assign should be within the range you set in the dialog box, that is:

current value ≥ minimum value
current value < maximum value

The current value can never be equal to the maximum value; if you assign the maximum value, the barrel dial will display the minimum value.

Setting a value outside the range

If you set a value greater than or less than the range, Rapid wraps around in order to set an acceptable value.

Example

The barrel dial's minimum value is 0 and the maximum value is 15. If your application has the logic:

```
Dial1.currentValue := 17
```

the barrel dial will display "02."

Changing the Rotation Rate

The initial rotation rate is set in the dialog box, but you can change it through the logic. The range is 0.0-15.0 digits per second.

❖ *NOTE: If the rotation rate is set to 0.0, the barrel dial will not rotate.*

Connecting Barrel Dials

You can connect two or more barrel dials so that the rotation of one affects the rotation of the other. This connection is possible because barrel dials generate events which can be used to rotate other barrel dials. There are two events: *carry* and *borrow*.

Using the *carry* event

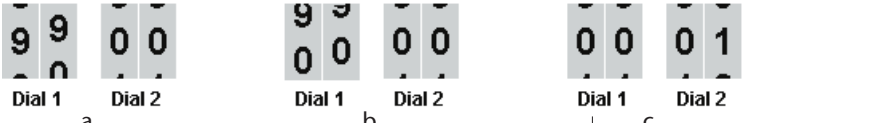
When a barrel dial rolls from its maximum value to its minimum value, the *carry* event occurs.

Example

In this example, Dial1 has a minimum value of 0 and a maximum value of 100. The *carry* event occurs when the dial rolls from 100 to 0. You can use the following logic to cause a second barrel dial to increase by 1 each time Dial1 rotates past its maximum value:

Trigger: Dial1 carry &
Action: Dial2.currentValue changeBy: 1

Below is an illustration of this trigger and action. (Note that Dial1 is defined with smooth scrolling; if it was set with jump scrolling, you would not see the smooth transition from 100 to 00.)



Using the *borrow* event

When a barrel dial rolls from its minimum value to its maximum value, the *borrow* event occurs.

Example

Using the same example shown above, the *borrow* event occurs when the dial rolls from 0 to 100. You can use the following logic to cause a second barrel dial to decrease by 1 each time BarrelDial1 rolls past its minimum value:

Trigger: Dial1 borrow &
Action: Dial2.currentValue changeBy: -1



SECTOR DIAL

The values you set in the Sector Dial dialog box (see p. 4-23) determine how the sector dial appears when the Prototyper is started. You can change most of these values during runtime through the logic (see the *Object Reference* manual). This section gives examples of how to achieve various effects using the sector dial.

Using the *reading* Property

You can create several types of sector dials using the *reading* property. For example:

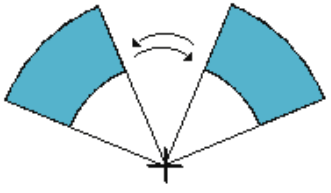
- A sector dial that simulates the continuous sweep of a radar.
- A sector dial that moves back and forth between two readings.
- A sectioned, circular dial.

To create a sector dial that simulates the continuous sweep of a radar:

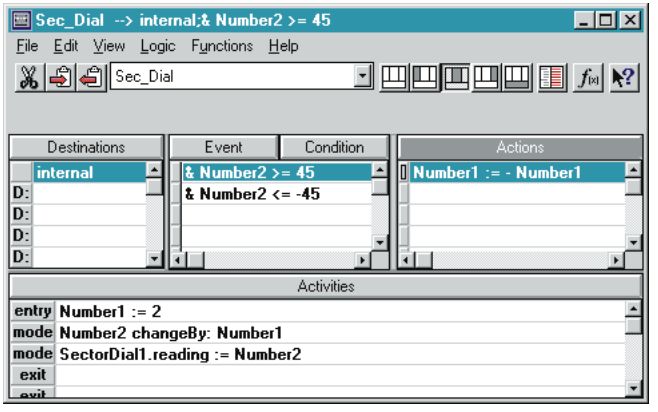
- Add a mode activity that uses the *changeBy:* function, such as:

```
Mode: SectorDial1.reading changeBy:2
```

To create a sector dial that moves back and forth between two readings:

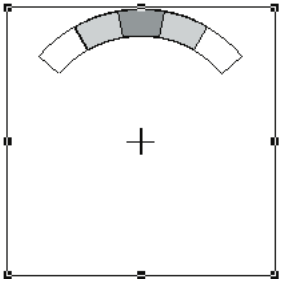


- Construct logic similar to the logic shown below:



(The second trigger has the same action as the first trigger.)

To create a sectioned, circular dial:



- Place several sector dials at the same position and set their initial readings to different values (you can set the values either in the dialog boxes or in the logic).




JOGDIAL

The Jogdial object is a specialized potentiometer that generates pulses as it rotates. It can be rotated in either or both directions and for unlimited numbers of rotations. It has two properties:

self

value

All of the jogdial's properties and functions are explained in the *Object Reference* manual. The following sections explain some of the more common features of the Jogdial object's logic.

Rotating the Jogdial

You can use the following logic functions to rotate the jogdial in the Prototyper:

FUNCTION	DESCRIPTION	EXAMPLES
rotateAt: <number>	Rotates the jogdial from its current position, at the direction and speed (in degrees per second) defined by <number>. The rotation continues until it is stopped by the logic.	Jogdial1 rotateAt: -4 *NOTE: The activity Jogdial1 rotateAt: 0 stops the rotation.*
rotateBy: <number1> at: <number2>	Rotates the jogdial from its current position by a distance (in degrees) determined by <number1>. The direction and speed of the rotation (in degrees per second) is defined by <number2>. *NOTE: If <number1> is greater than zero but <number2> is less than zero (or vice versa), then the function is not executed.*	Jogdial1 rotateBy: 30 at: Speed_Num *NOTE: The activity Jogdial1 rotateBy: 30 at: 0 stops the rotation, even if the jogdial has not yet rotated by 30°.*

If either of these activities moves the jogdial more than the pulse resolution, then a positivePulse or negativePulse event is generated. For example, assuming a pulse resolution of 30°:

ACTIVITY	EVENT(S) GENERATED
rotateBy: 40 at: -5	One negativePulse event
rotateAt: 3	One positivePulse event every ten seconds

Fast Rotation

In the Jogdial's More dialog box in the Object Layout, you set a rotation rate threshold in pulses per second. If during runtime the jogdial is rotated:

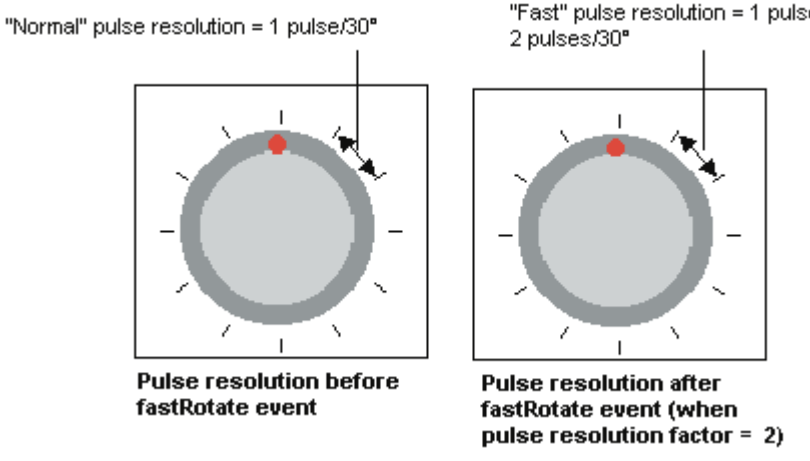
(a) faster than the threshold

(b) over a distance greater than the pulse resolution (also defined in the Object Layout)

then the *fastRotate* event is generated.

When the *fastRotate* event is generated, the pulse resolution changes automatically according to the pulse resolution factor that you defined in the Object Layout. The new pulse resolution = pulse resolution / pulse resolution factor.

For example:



❖ *NOTE: When Rapid detects that the jogdial rotation speed has returned to a value which is less than the rotation rate threshold, the pulse resolution automatically returns to normal.*

Clicking the Jogdial

You can use the Jogdial's *click* function to execute an action equivalent to the user clicking with the left mouse button on the hotspot at the center of the jogdial. The *clicked* event is generated.

Setting the Jogdial Value

Use the assign function of the Jogdial's *value* property to set the jogdial indicator's position relative to the vertical (0°) angle. Because the jogdial's rotations are unlimited, this number is unlimited as well; the indicator's relative angle, therefore, is calculated as (*value* modulo 360).

Examples (assuming a pulse resolution of 30°):

Current value:	Value assigned:	Result:	Events generated:
180	360	Half-turn clockwise	6 positivePulse
180	470	Three-quarters turn clockwise	9 positvePulse
180	90	A quarter-turn counterclockwise	3 negativePulse

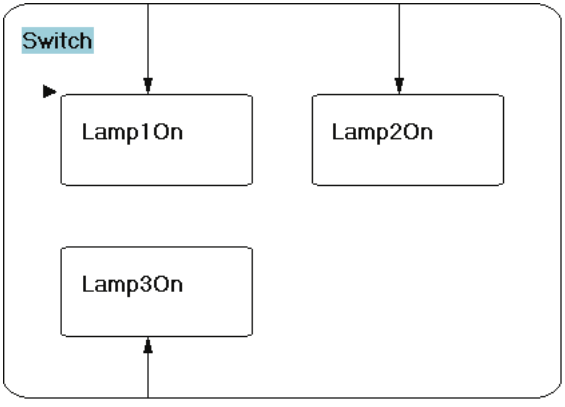


SWITCHES

When you define a trigger based on the position of a switch, you have the choice of using conditions or events. In many cases, the choice is not significant. However, when moving from one position to a non-adjacent position (e.g., from position1 to position3 in a 3-position rocker switch), there is an important difference between the behavior of triggers using events generated by switch positions and triggers using conditions that test switch positions. Intermediate triggers which are event-based will be triggered, but intermediate triggers which are condition-based will not be triggered.

Example:

Suppose you have an application comprised of a 3-position rotary switch and three lamps. You have three transitions to different modes that are triggered by three different **events**.

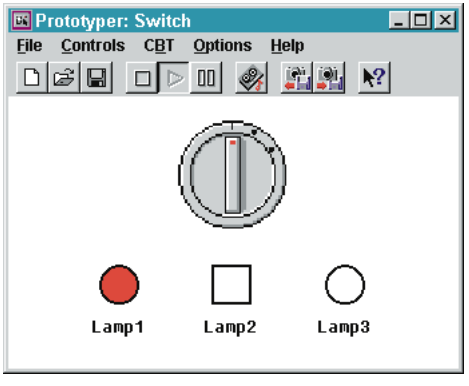


The three events are:

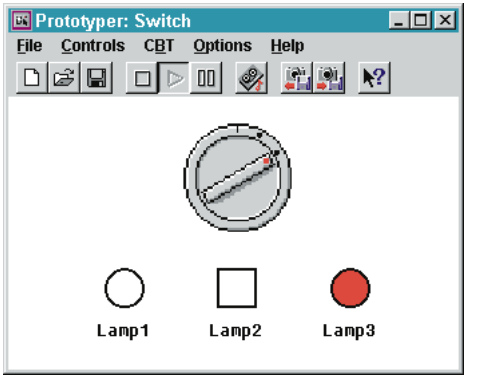
Switch1.position1 make &
Switch1.position2 make &
Switch1.position3 make &

Each mode has entry and exit activities that light and turn off one of the lamps: position1 lights Lamp1, position2 lights Lamp2 and position3 lights Lamp3. When each mode is exited, its lamp goes out.

When you run the Prototyper, Rapid turns on Lamp1 (the default mode):



If the switch were moved to position3, you might expect Lamp3 to light, but not Lamp2. However, "on the way" to position3, the event *position2 make* occurs, so Lamp2 momentarily lights as well.



When the switch is moved from position1 to position3, the following modes are entered and exited:

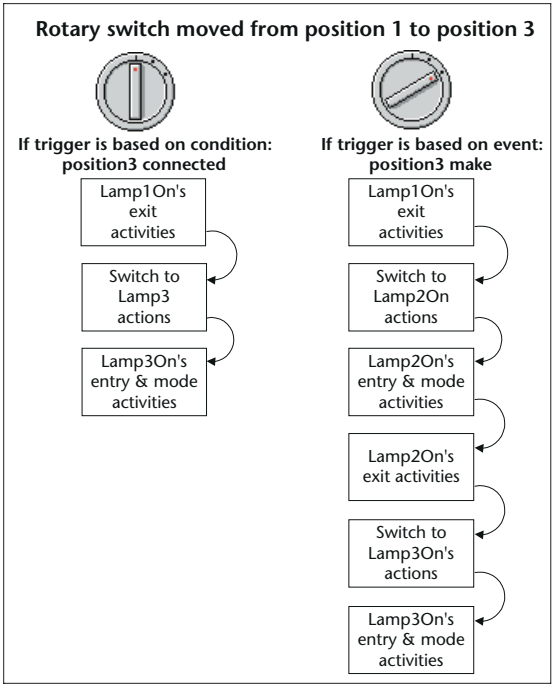
- Lamp1On is exited.
- Lamp2On is entered.
- Lamp2On is exited.
- Lamp3On is entered.

(This effect actually resembles a real-life switch; if you quickly move a switch from position1 to position3, position2 will momentarily connect.)

To avoid this situation, each of the triggers should be based on a condition, not on an event, as follows:

& Switch1 position1 is connected
& Switch1 position2 is connected
& Switch1 position3 is connected

The resulting chain of actions and activities—in both cases—is illustrated below:



This difference in behavior must be taken into account when choosing whether to use the *make* or *break* events generated by switch positions, or to use conditions that test switch positions. There are times when each particular pattern of behavior is desirable, depending on the application.



JOYSTICK

You can use a joystick to move (or activate) objects and bitmaps. The triggers and actions listed in the example below illustrate how a joystick moves a bitmap.

Example

This application has four objects:

- Joystick: all the positions were defined as auto repeat, except for the center position
- Imported bitmap: defined as dynamic

- Integer: called “step”
- Lamp

TRIGGER	ACTION
Joystick1.down make &	**Bitmap1.Position changeYBy: step**
Joystick1.downLeft make &	**Bitmap1.Position changeXBy: -step yBy: step**
Joystick1.left make &	**Bitmap1.Position changeXBy: -step**
Joystick1.upLeft make &	**Bitmap1Position changeXBy: -step yBy: -step**
Joystick1.up make &	**Bitmap1.centerPosition changeYBy: -step**
Joystick1.upRight make &	**Bitmap1.Position changeXBy: step yBy: -step**
Joystick1.right make &	**Bitmap1.Position changeXBy: step**
Joystick1.downRightmake &	**Bitmap1.Position changeXBy: step yBy: step**
Joystick1.center make & not Lamp1 is on	**Lamp1 on**
Joystick1.center make & Lamp1 is on	**Lamp 1 off**



ANIMATED OBJECT

Using the logic, the animated object can run once or repeatedly. It can run in place, move to a designated position or move across the screen. The frames can be flipped through in either ascending or descending order. The object can also generate an event when clicked.

Running the Animated Object

There are three functions that “animate” the animated object:

FUNCTION	DESCRIPTION
run	Flips through the frames once (at the flip rate) and stops at the last frame. The default flipping order is from the first frame to the last.
runRepeat	Flips through the frames repeatedly (at the flip rate). The default flipping order is from the first frame to the last.

FUNCTION	DESCRIPTION
currentFrame changeBy:	When you use the mode activity **Animated1.currentFramechangeBy: 1**, the frames are flipped through repeatedly. Be aware that you cannot control the flip rate with this function.

Moving the Animated Object

There are four functions that cause the object to move across the screen:

FUNCTION	DESCRIPTION
move	Moves the object across the screen.
move:	Moves the object a specified distance.
moveToPoint: and *moveToX:y:*	Moves the object to the specified coordinate.

Rate of movement

The rate at which the object moves is determined by the *speed* property. The default speed is 10 scale units per second. If your application uses unit other than pixels, you may need to change the speed.

You can assign a negative value to speed, in which case, the object will move in a backwards direction.

Direction of movement

The direction (in degrees) in which the object moves is determined by the *direction* property. The range is 0–360°; the default value is 90° (to the right)

Putting multiple sequences in one animated object

You could put more than one animated sequence in a single animated object and then choose which sequence to run by using the *firstFrame* and *lastFrame* properties. If, for example, Sequence A is in frames 1 through 6 and Sequence B in frames 7 through 12, you would use the following logic statements to run Sequence A:

```
Anim1.firstFrame := 1
Anim1.lastFrame := 6
Anim1 run
```

❖ *NOTE: If the lastFrame value is less than the firstFrame value, Rapid gives a runtime error.*



PIPES

The Pipe object has three properties:

- *self*
- *pattern*
- *speed*

The following sections explain some of the more common features of the Pipe object's logic.

Activating the Pipe's Flow

The *self* property's *flowOff* and *flowOn* functions control the activation of the pipe's flow. To activate a pipe's flow, for example when a valve is connected, use the following syntax:

TRIGGER	ACTION
& Valve.position2 is connected	Pipe1 flowOn

If the valve is connected to a different position, you may want to stop the flow in the first pipe and start it in another:

TRIGGER	ACTIONS
& Valve.position1 is connected	Pipe1 flowOff Pipe2 flowOn

Sometimes you might want to hide (and later show) the flow. When the flow is hidden, Rapid displays the pipe with its background color as defined in the Object Layout (by default, light gray); do not confuse this with the pattern's background color. If the flow is hidden, it is not enough to just turn the flow on; you also have to show it. To show (or hide) a pipe's flow, use the self

property's *showFlow* (or *hideFlow*) function. A full version of this example would read:

TRIGGER	ACTIONS
& valve position1 connected	Pipe1 hideFlow Pipe1 flowOff Pipe2 flowOn Pipe2 showFlow

If you want to show the flow as a stationary pattern, turn the flow off but do not hide it. (Alternatively, assign zero to the pipe's *speed* property, see below.) If you hide the flow, we recommend that you also use the *flowOff* function to avoid wasting your computer's system resources.

Changing the Flow's Pattern

The *hold:* function of the pipe's *pattern* property allows you to assign a different flow pattern to the pipe. The example below demonstrates logic that produces a new flow pattern in a given pipe.

TRIGGER	ACTION
& Valve.position2 is connected	Pipe1.pattern hold: Pattern3

Changing the Flow's Speed

The pipe's *speed* property is treated as an integer value and has all the functions of the Integer object with the following limitation:

- It cannot be less than -10 or greater than 10.

Apart from the above limitation, you may assign it a specific value or perform any mathematical function on it. The example below doubles the speed of the flow:

```
Pipe1.speed := Pipe1.speed * 2
```



VALVES






Use the valve objects to cause the flow in the pipes to turn on and off. This gives the effect of the material being rerouted to a different pipe.

Activating a Pipe's Flow with a Valve

When the valve is turned to a position, it generates an event which may trigger a transition. The following example turns on the flow in one pipe, and turns off the flow in another, when a selector valve is set to position 2:

TRIGGER	ACTIONS
Valve.position2 make &	pipe1 flowOff pipe2 flowOn



TANKS

The major use of the tank objects is to simulate the storage and/or collection of a moving fluid. Rapid treats the *contents* property as a number which you may alter in the application's logic. You may assign it a particular value or change it by a constant amount with the *changeBy:* function.

Assigning a value to a tank's contents

The following action doubles the tank's contents value:

```
Tank1.contents := Tank1.contents * 2
```

Continuously changing a tank's contents

The following action continuously decreases the tank's contents by 5:

```
Tank1.contents changeBy: - 5
```

If you use this as an activity, it must be a mode activity otherwise it will not continuously update.

Triggering actions/activities with a tank

A common trigger for a tank would be to turn off the flow of a pipe when the tank is empty. The condition for such a trigger would look similar to this:

& Tank1.contents = 0

Similarly, you might wish for a valve to disconnect when a particular tank becomes full. In this case you would replace the zero from the above example with the tank's maximum value (as defined in the Object Layout).

CHAPTER 14

Find and Replace

The Find and Replace feature performs the following functions:

- Searching for and replacing text in the application logic, i.e., in triggers, actions, activities, or user-defined functions (UDFs).
- Finding logic references to an object (or to object properties).

You can set search criteria so that Rapid limits the search to specified logic categories.

In this chapter you will read about:

- Defining a search string and various search options.
- Replacing a string in all or some logic statements.
- Focusing a search on specified types of logic statements and/or on statements that reference a specific object.

ACCESSING FIND AND REPLACE



To open the Find & Replace dialog box:

- Choose Edit|Find/Replace in the Mode Tree, or



- Choose Edit|Find/Replace in the Logic Editor, or

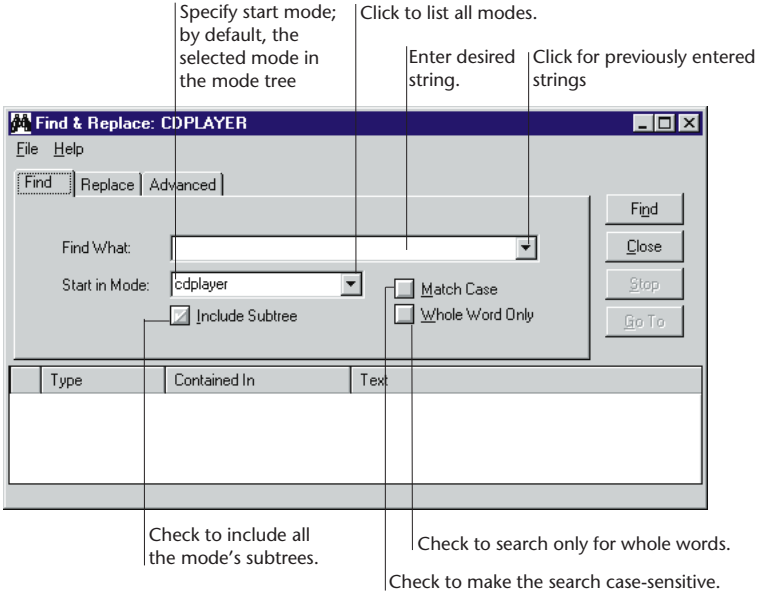


- Click on the Find/Replace button in the Logic Editor.

The three tabs in the Find & Replace dialog box (Find, Replace, and Advanced) each give access to a different set of operations.

USING FIND

The Find option looks for a specified substring in your logic.



Setting the Search Options

Before you initiate a search, you need to:

- Define the search string (using wild cards as desired)
- Define the search scope
- Set other search options, if desired.

Defining the search string

In the Find tab of the Find & Replace dialog box, enter the desired string in the “Find What” edit box. Clicking the arrow next to this box gives a list of up to the last five search strings.

See the section on “Setting wild cards” below.

Defining the search scope

In the “Start in Mode” box, enter the mode from which you would like Rapid to start its search. Clicking the arrow next to this edit box gives a list of all the modes in your application.

If the “Include Subtree” checkbox is checked, then Rapid searches through the start mode and all of its descendants.

Unless otherwise specified, Rapid starts from the mode currently highlighted in the mode tree and searches through all of its descendants.

Additional Search Options

- *Match Case* – If checked, Rapid’s search is case-sensitive. Default: unchecked.
- W*hole Word Only* – If checked, Rapid finds only those strings that exactly match the search string and that are enclosed by nonalphanumeric characters (such as space, comma, colon, ampersand). Default: unchecked.

  Example:

  If your search string is “lamp” and the “Whole Word Only” option is checked, Rapid

  - will **not** find matches such as “lamp1” or ”Redlamp”.
  - **will** find matches such as “lamp.blinkPeriod:=300" and ”&lamp is on".

Setting wild cards

You may specify wild cards in the search string by typing an asterisk (*) before, in the middle of, or after the substring.

❖ *NOTE: Wild card specification causes the "Whole Word Only" option to become disabled.*

Example:

Suppose your application has three objects, named "Switch1A", "Switch2A", and ""Switch3A". To search for all the logic items containing the name of any of these objects, type:

Switch*A

in the "Find What" field.

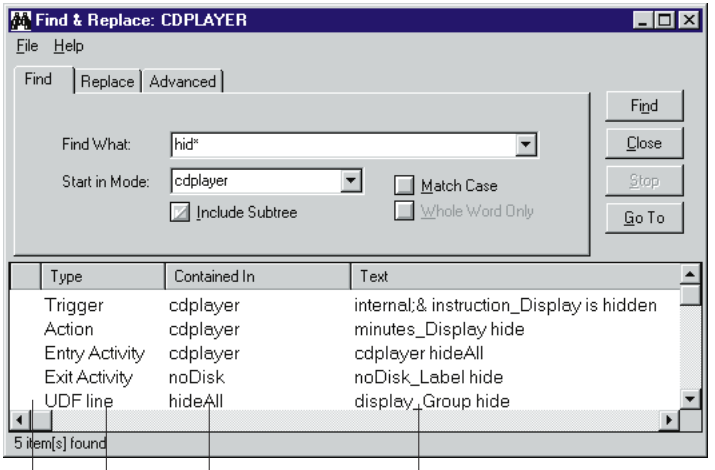
Initiating a Search

When you are satisfied with your search options, click Find to initiate the search. For information on setting advanced search filters, see p. 14-8.

Display of Search Results

At the end of the search, a list appears showing all the logic lines in which the search string appears.

An example of the display is shown in the figure below.



Lists the complete logic statements in which the search string is contained.

Lists the mode or UDF to which the found item belongs.

Lists the logic type (action, trigger, activity, UDF) in which the string appears.

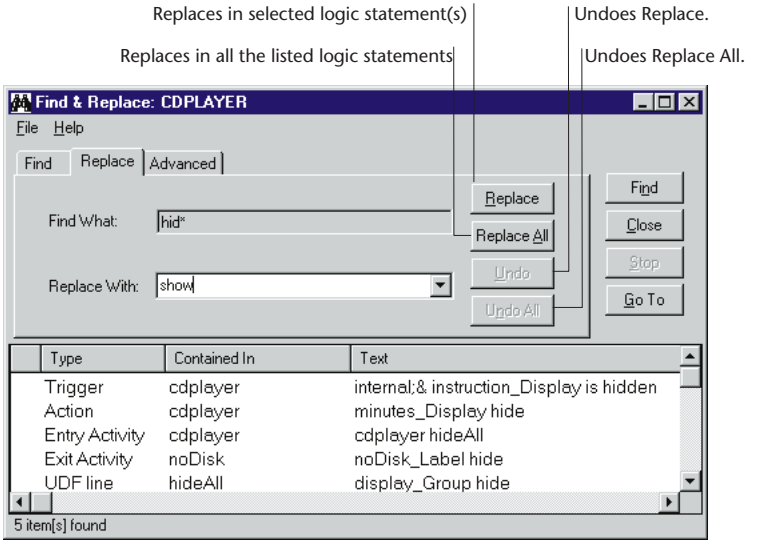
Status column: shows success or failure of replacement request. See "Logic Validity Check", p. 14-7.

The syntax of the search results is summarized below:

TYPE	CONTAINED IN	TEXT
Mode Activity Entry Activity Exit Activity Action	mode name	the activity or action
Trigger	source mode name	destination type and destination mode (if not internal); the trigger
UDF line	UDF name	the logic line

USING REPLACE

The Replace option replaces an existing string with a new string. Below is an example of the Replace window with a replacement request entered, before execution.



Replacing a String

To replace a string:

1. Do a search for the string you wish to replace. (See the section "Using Find", p. 14-2.)
2. Click on the Replace tab.
3. In the "Replace With" box, type in the replacement string.
4. If you wish to replace **all** the instances of the search string, click Replace All; otherwise,
   a. In the search results list, click on those lines in which you wish to replace the string. Rapid highlights the selected lines.
      ❖ *NOTE: You can use the standard Windows functions of Shift+click and Ctrl+click to select multiple lines.*
   b. Click Replace.

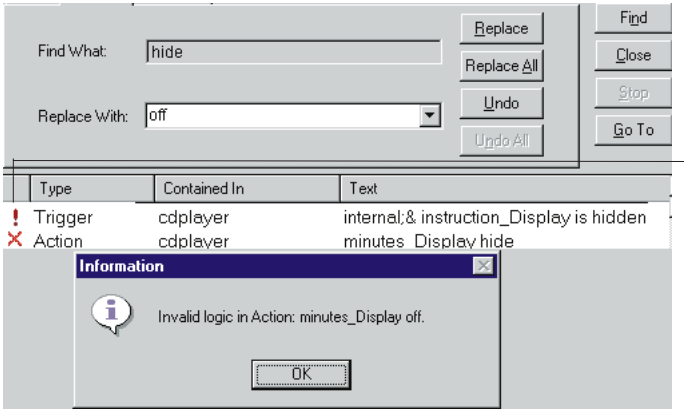
Rapid checks that the lines of logic created by the replacements are logically valid, i.e., that they are (1) syntactically correct, and (2) executable within the context of your application's logic.

The results of this logic validity check are given in the status column. Here any or all of the following marks are shown after execution:

- a green checkmark, if the entry is valid after the replacement.
- a red "X", if the entry is not valid after the replacement.
- an exclamation mark, if you changed a line of logic in the Logic Editor while your Find and Replace window is open and that line of logic is listed in the window.

Error Messages

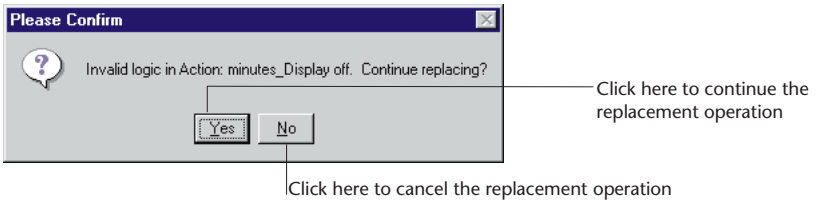
If a line becomes invalid after the replacement then, in addition to getting a red "X" at that line, a message appears:



Indicates that this trigger was changed in the Logic Editor while this line was displayed in the Find & Replace window

In this example, we tried to replace the string "hide" with the string " off". The resulting action ("minutes_Display off") is not logically valid. When an error such as this occurs, the replacement is not carried out.

If there had been more than one logic statement to replace, the error message would be as follows:



Undo

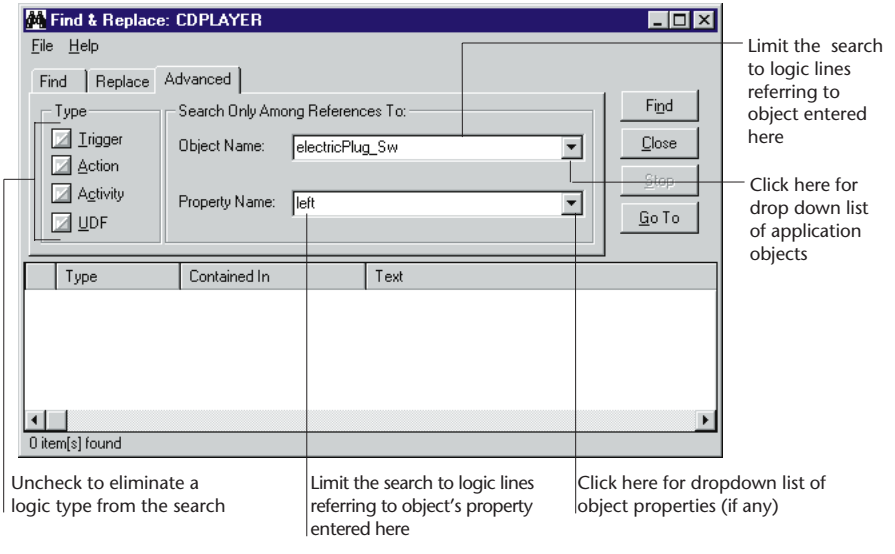
The Undo button is enabled after a Replace operation. Click it to undo the replacement(s). The Undo All button is enabled after a Replace All operation. Click it to undo all the replacements.

USING FILTERS ("ADVANCED" TAB)

Use the Advanced tab features to focus the search on logic types and/or objects of interest. The logic types that can be designated for filtering are those that pertain to triggers, actions, activities, and/or UDFs. In addition, you may limit the search to a particular object and property.

To access the advanced filter options:

- Click on the Advanced tab to access the options, as follows:



To limit the search to specific logic items:

- (Un)check the (un)desired types of logic items.

To limit the search to a particular object/property:

1. Either select the object from the drop-down list of application objects or type in a name in the edit box.

❖ *You can use wild cards for the object name. The entry* **Lamp*** *would include logic items that reference all objects whose name starts with Lamp.*

2. To see logic items referring only to a particular object's property, enter the name of the property in the Property Name box. Click the arrow to the left of the box for a drop-down list of object properties (if any).

❖ *NOTE: If you leave the Property Name box blank, the search will reference all the logic pertaining to the specified object.*

OTHER FIND & REPLACE FEATURES

Closing the Find & Replace Window

Click the Close button to close the Find & Replace window. The current search options are saved and appear as the default settings the next time you open the window.

Stopping a Search

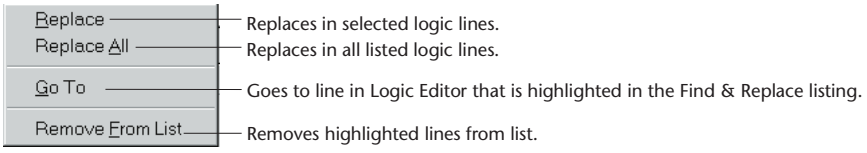
Click the Stop button to interrupt the search in process.

Jumping to a Logic Line

Click the Go To button to jump to the corresponding trigger, action, activity, or UDF in the Logic Editor that is highlighted in the Find & Replace listing.

Deleting Lines from the Displayed List

Right-click anywhere within the Find & Replace window to access the following popup menu:



Click Remove From List to remove the highlighted lines from the search result list.

SOME FIND & REPLACE EXAMPLES

The following examples show how you can combine various find and replace features to achieve certain results. These examples are not intended to be comprehensive and you will discover many other ways to use this module in your application-building procedures.

Example 1: Change Logic

Scenario: You implemented transitions between certain modes in subtree *PowerOn* via the event *Pushbutton1 in* and now you want to implement some of those transitions with the event *Switch1.up make*.

1. Set the following search filters:

   In the Find tab:
   - Find What: Pushbutton1 in
   - Start in Mode: PowerOn
   - Include Subtree: checked

   In the Advanced tab:
   - Type: only Trigger should be checked
   - Object Name: you could, optionally, enter Pushbutton1 here, but your search string in the Find tab eliminates this as a necessity.
2. Click the Find button.
3. When the search is complete, click the Replace tab.
4. In the Replace With edit box, type in (carefully) *Switch1.up make*.
5. Select the triggers that you want to change.
6. Click Replace.

Example 2: Replace Text

Scenario: You want to replace the string *power* with *Power* in all places where it appears as part of text display contents. This example assumes that all text display names in your application include the substring *disp*.

1. Set the following search filters:

   In the Find tab:
   - Find What: power
   - Start in Mode: the root mode
   - Include Subtree: checked
   - Match Case: checked
   - Whole Word Only: checked

   In the Advanced tab:
   - Type: all checked except Trigger
   - Object Name: *disp*
   - Object Property: because you have used a wild card, you cannot specify a property. If you were to select a particular display object in Object Name, then you could select the contents property here.
2. Click the Find button.
3. When the search is complete, click the Replace tab.
4. In the Replace With edit box, type in *Power*.
5. If the search (because of the Object Name wild card) found logic lines which are not of interest:
   a. Control+click to select those lines.
   b. Right-click anywhere and choose Remove From List from the popup menu.
6. Click Replace All.

Example 3: Going to an Activity

Scenario: You have a large application. During runtime, many of its modes are active simultaneously. A text display named Display1 clears during runtime and this is undesirable.

1. Set the following search filters:

   In the Find tab:

- Find What: blank
- Start in Mode: the root mode
- Include Subtree: checked

In the Advanced tab:

- Type: all checked except Trigger and UDF
- Object Name: select *Display1* from the dropdown list
- Object Property: select *contents* from the dropdown list.

2. Click Find.
3. You scroll through the logic lines and you notice that the cause of the problem is a Mode Activity which should be an Entry Activity.
4. Select the line and click Go To.
5. In the Logic Editor, change the activity from mode to entry.

CHAPTER 15

Debugger

The purpose of the Debugger is to facilitate the isolation of an error in your application. With the Debugger, you can see the "inner workings" of your application step by step while checking the contents of each or any object. You will be better able to find out which point(s) in your logic and/or which object value(s) are causing possibly undesired results.

In this chapter you will read about:

- Controlling the application flow during runtime while using the Debugger
- Flow display while using the Debugger
- Managing breakpoints
- Inspecting the contents of objects

OVERVIEW

The Debugger, in effect, takes control of the Prototyper and allows you to inspect closely those parts of an application's logic which are of particular interest.

You would achieve this by, for example, setting breakpoints at mode entries or exits, or at specific logic lines. Once set, you run the Prototyper via the Debugger and the Debugger automatically pauses the Prototyper when it encounters a breakpoint. You can also pause the Prototyper at any time during execution.

Once the Prototyper is paused, you can use the Debugger controls to execute the application's logic step by step. With the Logic Editor in focus and synchronized with the execution, you can observe which logic line is about to be executed. You can also open inspectors to examine the contents of objects in their current state.

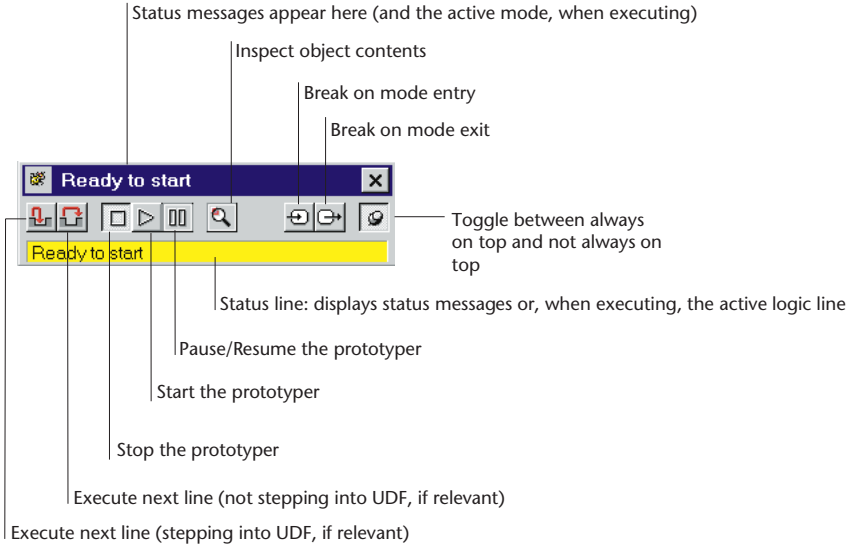
THE DEBUGGER WINDOW

To open the debugger:





In the Application Manager, choose Tools|Debugger or click the Debugger button.



CONTROLLING APPLICATION FLOW

There are two modes of application flow control when executing an application through the Debugger:

- stepping: the application's logic is executed one line at a time, as you click either the Step Over or the Step Into button.
- run: the application runs as usual in the Prototyper, until it encounters a breakpoint (or you click Pause).

❖ *NOTES: If have started the Prototyper before opening the Debugger, you will not be able to debug until you restart the Prototyper.*

You can initiate stepping or run mode when the Debugger is either stopped or paused. Thus, for example, you could start in run mode, pause at any time and continue with stepping mode—or you could start the application in stepping mode from the very beginning.

To step to the first (or next) logic line in your application, click either:



- Step Into: If the highlighted logic line is a call to a user-defined function (UDF), Step Into stops at the first logic line in the UDF.



- Step Over: executes the highlighted logic line and stops at the next one.

❖ *NOTE: If the highlighted logic line is a UDF call, Step Over executes the entire UDF and then stops at the next logic line.*

To run the application until the first (or next) breakpoint, click either:





- Start: if the Debugger is stopped, or
- Resume: if the Debugger is paused.

When you first start the Debugger (in stepping or run mode), the Prototyper window is displayed in the foreground and your application begins to run. Status messages appear in the Debugger window. For more information on these messages, see p. 15-5, "Debugger Status Messages".

To pause the application execution, either:

- Choose Controls|Pause in the Prototyper window, or
- Click the Pause/Resume button in the Prototyper or Debugger window.

❖ *NOTE: The Debugger pauses the application automatically when it hits a breakpoint (in run mode) or after it has executed one logic line (in stepping mode). The status messages, however, are different for each type of pause (see p. 15-4).*

To resume the application execution in run mode, either:

- Choose Controls|Resume in the Prototyper window.
- Click the Pause/Resume button in the Prototyper or Debugger window.

To resume the application execution in stepping mode:

- Click either Step Into or Step Over.

❖ *NOTE: If your application is at a point where it requires user input in order to continue, your application will remain idle (with the status message "Prototyper Running") after you've resumed the Debugger. Provide the required input for your application to continue (e.g., click a pushbutton or change a switch position).*

To stop the Debugger:



- Click the Stop button in the Prototyper or Debugger window. The status message becomes "Ready to start".

FLOW DISPLAY WHILE DEBUGGING

As you execute the application's logic, you can get visual cues from:

- status messages that are displayed in the Debugger window,
- changes in the Breakpoint at Entry and Breakpoint at Exit buttons in the Debugger window,
- highlighting of the current mode in the mode tree, and
- highlighting in the Logic Editor of the logic line which is about to be executed.

❖ *NOTE: In general: the Debugger indicates breakpoint settings in red and highlights active modes and logic lines in yellow.*

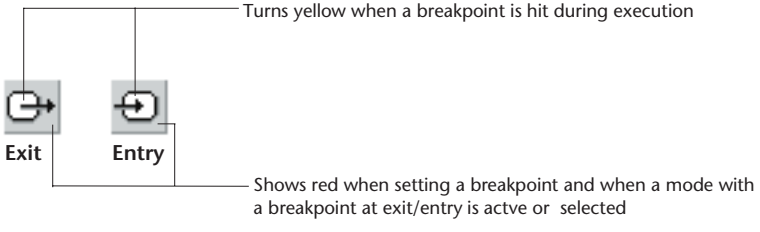
Debugger Status Messages

The following status messages are displayed in the Debugger window's title bar:

DEBUGGER STATUS	MESSAGE
Stopped	Ready to start
Started in run mode	Prototyper Running
Started in stepping mode	Stepped At: <logic line description>, such as Mode Entry, Mode Exit, Entry Activity, Exit Activity, etc.
Paused by the user	User Break At: Idle or User Break At: <logic line description>
Paused by a breakpoint	Hit Breakpoint At: <breakpoint location>, such as Mode Entry, Mode Exit, Entry Activity, etc.
In run or stepping mode, when a user event is required to continue	Prototyper Running

Breakpoint Buttons in Debugger Window

When the application hits a breakpoint at mode entry or mode exit, the mode box in the appropriate breakpoint button becomes yellow, as follows:



Active Mode in Mode Tree

If the Mode Tree window is in focus, you will see that the mode of the logic line currently being executed is highlighted in yellow. In addition, a yellow arrow pointing towards or away from a mode indicates a mode about to be entered or exited. See p. 15-7 for more information on breakpoint designations in the mode tree.

Active Line in Logic Editor

If the Logic Editor display is synchronized with the execution (see below), then you will see that the logic line to be executed next is highlighted in yellow in the Logic Editor.

Synchronizing the Logic Editor and the Debugger

There are two items in the Logic Editor's Debug menu which control synchronization between the Logic Editor display and application execution in the Debugger:

- Follow execution: when this option is checked (the default), then the Logic Editor display is automatically synchronized with the application execution each time that the Debugger pauses the application. In other words, the Logic Editor goes to (and highlights) the logic line to be executed next.
- Synchronize: when the Debugger is paused and you choose this item, the Logic Editor immediately goes to (and highlights) the logic line to be executed next.

Follow Execution vs Synchronize

You may be wondering why you would need to synchronize the Logic Editor and Debugger if you have Follow Execution enabled.

The Synchronize command is useful when you have been browsing to other modes in the Logic Editor while the Debugger is paused. You can then choose Debug|Synchronize to immediately resynchronize the Logic Editor and the Debugger.

MANAGING BREAKPOINTS

A breakpoint is a logic line (or mode entry/exit) in an application where the user designates that the Debugger pause. Its purpose is to enable the user to observe the state of an application at various stages.

You may select any logic line (i.e., destination, trigger, action, activity, UDF line), or the entry or exit point of a mode to be a breakpoint.

❖ *NOTE: You can(un)designate your breakpoints at any time, including while the Debugger is running.*

To put/remove a breakpoint in a logic line:

1. Select a logic line in the Logic Editor.
2. Do one of the following:
   - Click the gray box to the left of the logic line.
   - Press Ctrl+B.
   - Choose Breakpoint in the popup menu.
   - Choose Debug|Breakpoint.

To put/remove a breakpoint at a mode entry/exit:

1. Choose the desired mode, either in the Mode Tree or in the Logic Editor.
2. Choose Break on entry or Break on exit in the Mode Tree popup menu, or click the appropriate button (Mode Entry or Mode Exit) in the Debugger window.

To remove all breakpoints simultaneously:

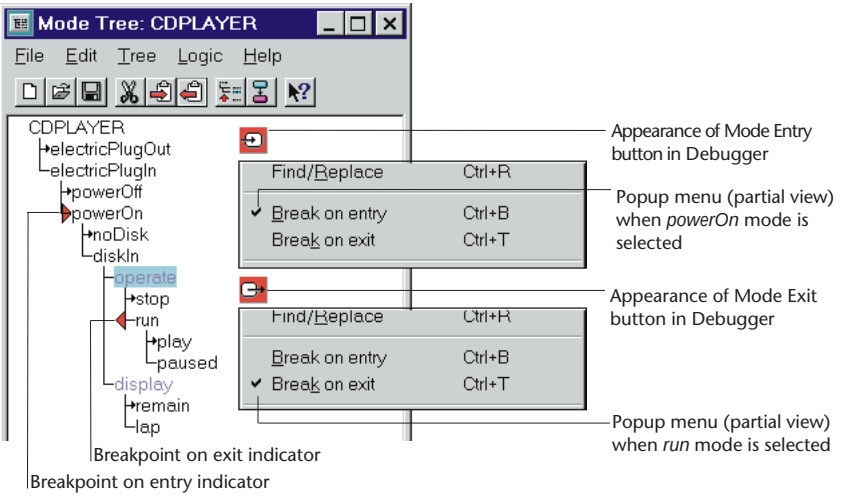
- Choose Debug|Remove all breakpoints in the Logic Editor.

Display of Breakpoints

When you choose a mode entry or exit as a breakpoint, the following display effects occur:

- In the mode tree, a red arrow points towards the mode (for break on entry) or away from the mode (for break on exit). See the figure below.

- In the Mode Tree popup menu, a check appears next to Break on entry or Break on exit when the mode is selected.
- In the Debugger window, the background of the Mode Entry or Mode Exit button becomes red.



INSPECTING THE CONTENTS OF AN OBJECT

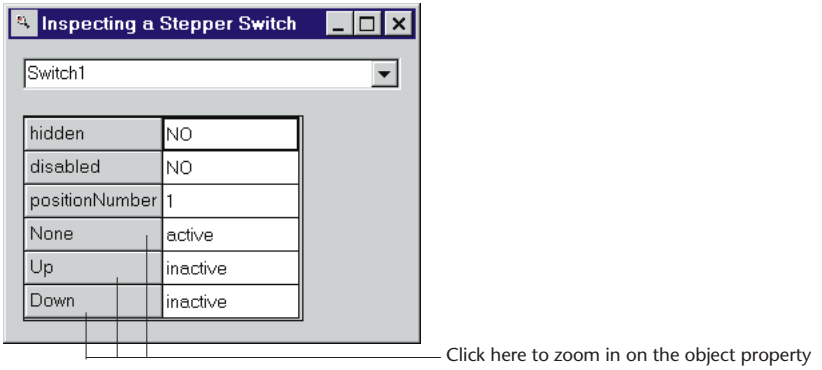


At any time when the Prototyper is started (running or paused), whether or not the Debugger is open, you can use the Inspect facility to display the state of an object.

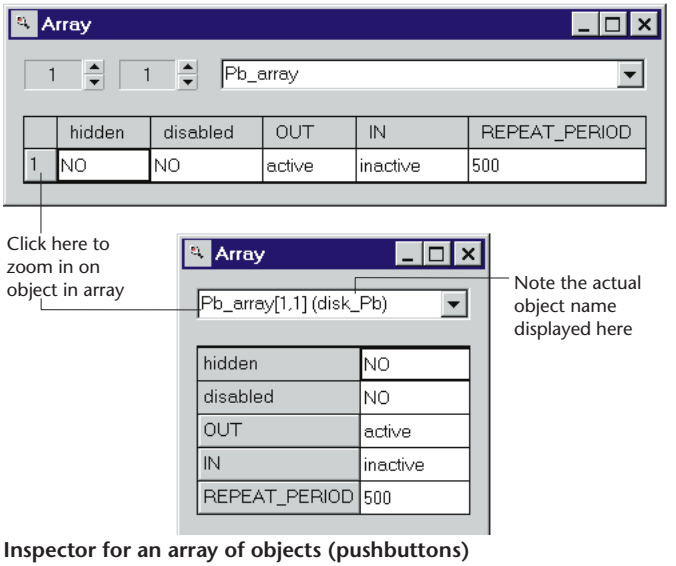
❖ *NOTE: When the Prototyper is paused, the data in the inspect window is up-to-date. When the Prototyper is running, however, the data are not necessarily the most current and so the contents are grayed.*

To inspect an object via the Logic Palette:

1. Start the Prototyper.
2. In the Logic Palette, choose from the Object list the object you wish to inspect.
3. Choose Inspect in the Logic Palette popup menu. An inspection window appears, as follows:



Inspector for a graphic object



Inspector for an array of objects (pushbuttons)

To inspect an object via the Logic Editor:

1. Start the Prototyper.
2. In the Logic Editor, enter the desired object in the edit line. This may be done by either:

- clicking on an activity that contains the name of the object, then highlighting the name in the edit line, or
- typing the name of the object in the edit line.



3. Choose Debug|Inspect.

❖ *NOTES: If you do not highlight the name of an object, Rapid looks through the entire edit line for names and opens inspectors for all of them.*

If no name was found in the edit line, or the edit line is empty, a dialog opens in which you are prompted to enter the name of an object.

To inspect an object via an inspector:

- Type in the name of an object in an open inspector.

The inspector now shows the contents of the object just entered, instead of the object which was previously displayed.

❖ *NOTE: Click on the arrow to the left of the combo box to drop down a list of the last five objects inspected.*

CHAPTER 16

Rapid Logic: Hints and Tips

This chapter covers several issues that are useful for the more advanced Rapid user. The topics covered are:

- Summary of Rapid logic syntax.
- Things to remember when working with arithmetic functions.
- Building a selector routine for situations like switching between TV channels.
- Sequence of operations after a transition is triggered.
- Optimizing Rapid applications.

SUMMARY OF RAPID SYNTAX

This summary of the syntax rules of Rapid logic is useful for those who prefer to type logic and for those who prefer the conventional method.

General

The basic syntax for all Rapid logic phrases is:

<object>.<property> <function> [argument]

where the argument itself can sometimes be made up of an object-property-function-argument clause. For example:

Stopwatch.time := Timer.count + 5

All functions that require arguments include one colon (:) for each argument required. The argument(s) must follow the colon[1], and every colon must be followed by an argument.

Conditions

- Conditions are always preceded by an ampersand (&).

INCORRECT	CORRECT
Stopwatch is running &	**& Stopwatch is running**

- Conditions always begin with the keyword *is* or *has*.

INCORRECT	CORRECT
& Stopwatch isRunning	**& Stopwatch is running**

- The "not," "and" and "or" operands must always precede the condition.

INCORRECT	CORRECT
& Timer is not running	**& not Timer is running**

- When a trigger consists of an event and a condition, the event must precede the condition.

INCORRECT	CORRECT
Timer is running & Switch.up make	**Switch.up make & Timer is running**

1 Except for :=, where the argument follows the equals sign.

Events

- Events are always followed by an ampersand (&).

INCORRECT	CORRECT
& Switch.up make	**Switch.up make &**

- When a trigger consists of several events, they are separated by exclamation marks (!).

INCORRECT	CORRECT
Switch.up make or Pushbutton in &	**Switch.up make ! Pushbutton in &**

- When a trigger consists of an event and a condition, the event must precede the condition.

INCORRECT	CORRECT
Timer is running & Switch.up make	**Switch.up make & Timer is running**

User-Defined Functions

General syntax

The general syntax for a user-defined function (with arguments) is:

functionName: <Object Type:Object Name> [with: <Object Type:Object Name>]

where:

ITEM	DESCRIPTION
functionName	The name of the function as it appears in the Logic Palette (together with the keywords). See p. 18–17 for guidelines on naming functions.
Object Type	A unique name that identifies the object type to Rapid; Round_Lamp, Flat_Pushbutton, etc. This item determines the type of argument that the function will accept.

ITEM	DESCRIPTION
Object Name	An object name. The object does not have to exist in the application (in which case it is a "local" object). Every object name defined in the function **must** refer to an object in the function's logic (that is enclosed in angle brackets).
with:	A keyword that precedes an argument. The keyword can be any word (see p. 18–17 for guidelines on naming functions). Each of a function's arguments must be preceded by a keyword and a colon.

Some examples of user-defined functions are given below:

displayText: <String:String1>

Displays a string.

displayText: <String:String1> in: <Text_Display:Display1>

Displays a string in a text display.

holdWith: <Round_Lamp_Holder:Holder1>

Holds a specific round lamp in a suitable holder.

Conditions

The logic of long conditions can be divided into several lines (multi-line conditions), but each line must begin with one of the three logic operators: *or*, *and* or *not*. See p. 18–3 for details on multi-line conditions.

Working with Arithmetic Functions

- Rapid assigns the same priority to all mathematical operators, such as +. -, *, /, sin, cos, sqrt, etc., and evaluates expressions from left to right, unless there are brackets to change the order. For example, Rapid evaluates the activity

  Integer1 := 6 + 3 * 9 sqrt

  by adding 6 to 3 (=9), multiplying the answer by 9 (=81), then taking the square root, to assign the value 9 (=square root of 81) to Integer1. Contrast this method with conventional systems which would first take the square

root of 9 (=3), multiply it by 3 (=9), then add 6 (=15), because they assign different priorities to mathematical operators.

Use parentheses to modify the order in which the parts of an expression are evaluated, or change the order of the numbers and the operations, so that evaluation from left to right gives the result you want.

To give the result 15, the above activity could be written, without parentheses:

Integer1 := 9 sqrt * 3 + 6

Or, retaining the original order and with parentheses:

Integer1 := 6 + (3 * (9 sqrt))

- Unlike standard mathematical notation, Rapid functions operate on what precedes them, not on what follows them. For example, to calculate the sin of a number object, x, and to assign that value to another object, y, the action would be:

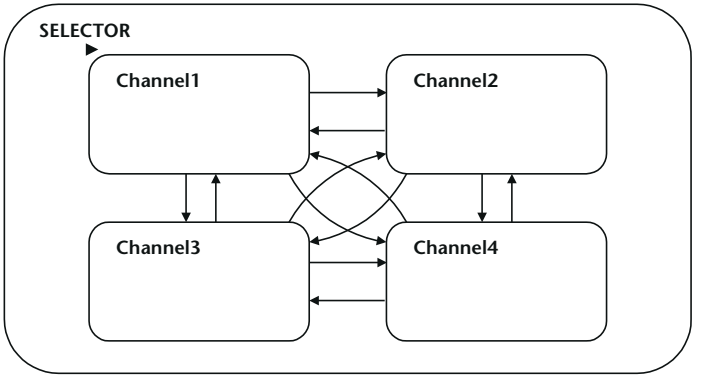
  y := x sin

- Add a space after arithmetic operators where there would be an ambiguity. For example, to distinguish between the value -5 and the operation - 5.

A SELECTOR: PARENT-TO-CHILD TRANSITIONS

A common element in many applications is a switch, or a series of pushbuttons, that transfers the application from one status to another. For example, a television has buttons that switch from one channel to another; a digital office-telephone allows you to switch from one line to another.

The most obvious (but not the best) solution, as illustrated in the State Chart below, is for every mode to have a transition to every other mode—triggered by a condition. If the application is in Channel3 mode and the switch is connected to position 4, the application moves to Channel4; when the switch is connected to position 1, the application moves to Channel1, etc.



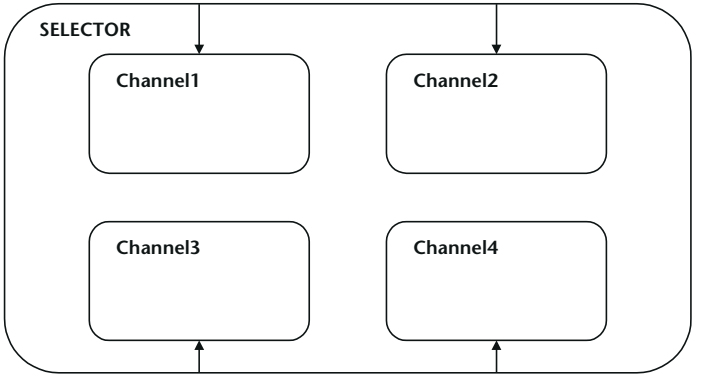
This solution has two major disadvantages:

- The number of transitions. Each possible transition between each of the modes must be explicitly represented. As the number of alternative modes increases arithmetically, the number of transitions between them increases geometrically.
- When the parent mode, Selector, becomes active, the default mode always becomes active. If the switch is in position 3, the application moves to Channel3, but only after Channel1 has first become active.

The more preferable solution consists of two parts:

1. Base all the transitions from one channel to another on transitions from the parent to the child.
2. Remove the default designation from Channel1.

Here is how this solution looks in the State Chart:



Each parent-to-child transition is triggered by a condition that tests the status of the switch, or tests which pushbutton is pressed in. For example

& Switch.position3 is connected

As soon as the parent mode becomes active, Rapid checks to see which of the conditions is true and activates that mode. After the transition, two modes are active: the parent and the child.

❖ *NOTE: Although the condition continues to be true, the transition to the child only occurs once.*

This method has two advantages:

- The total number of transitions is far fewer. Adding an additional channel option requires the addition of just one transition.
- When the parent mode is activated, the application does not always go to the same initial mode.

❖ *NOTE: When the parent mode becomes active, one of the conditions to the child modes **must** be true, otherwise Rapid will give a runtime error.*

SEQUENCE OF OPERATIONS DURING TRANSITIONS

When Rapid triggers a transition from one mode to another, it performs a whole sequence of operations, in a fixed order. This section describes what happens during and after transitions:

- How Rapid decides which modes to activate.
- The sequence of operations during the transition.
- What happens after the destination mode or subtree is activated.

How Does Rapid Decide which Modes to Activate?

When a transition's trigger becomes true, that is, an event occurs or a condition is fulfilled, Rapid calculates a list of which modes should become active and which should become inactive.

If the destination mode has children, Rapid calculates which of the children should become active, based on the type of transition that has just been triggered (default, history or deep history).

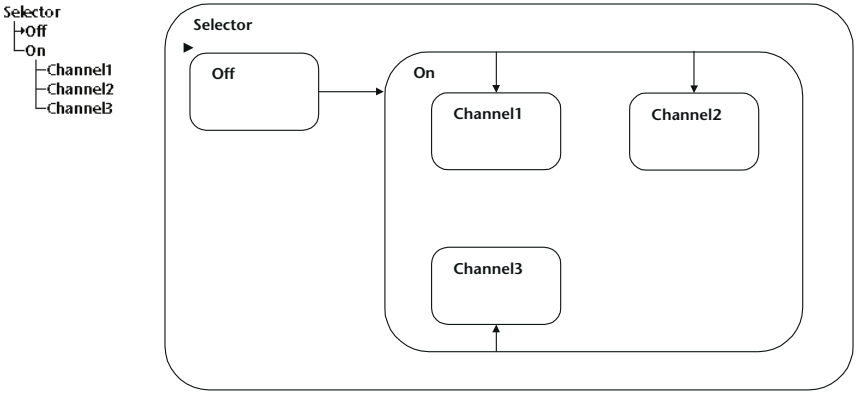
If the destination mode has ancestors, Rapid includes them in the "to be activated" list.

After Rapid has calculated "where it is going," it then executes the transition, exiting the source's set of modes and entering the destination's set of modes. Along the way, Rapid performs the necessary exit activities, actions and entry activities. These activities and actions do **not** affect the transition—Rapid activates modes based on its initial calculation, irrespective of any changes on the way.

As a general rule:

- **Rapid calculates "where to go" *before* it performs the transition. Any activities and actions that occur along the way do not affect the direction of the transition.**

This can sometimes lead to unexpected results, as in the following example:



The example application shown above has a mode, On, that has three children, none of which are a default mode. (See "A Selector: Parent-to-Child Transitions," 16-5 for a discussion of the significance of having no default mode.)

The transitions from On to each of its child modes are based on the value of an integer object, as follows:

- To Channel1: **& Integer = 1**
- To Channel2: **& Integer = 2**
- To Channel3: **& Integer = 3**

Question: If Off has an exit activity, **Integer := 2**, which channel mode will become active after the transition from Off to On?

Answer: We don't know. Rapid calculates which modes to activate before the transition, that is, before Off's exit activities occur. Rapid decides which channel mode to activate based on the value of Integer before the transition.

If Integer was equal to 3, Rapid would activate Channel3 and only then activate Channel2 (following Off's exit activity).

Question: What would happen if, prior to the transition, Integer was equal to 4?

Answer: Rapid would generate a runtime error. At the time when Rapid calculated which modes to activate, none of the transitions from On to its children could be triggered (as Integer was not equal to 1, 2, or 3). As a result, On had no active children, an illegal situation.

User events and logic events

Rapid makes the distinction between user events and logic events. A user event is an event generated as a direct result of a user's action. If you turn a switch, for example, you generate a user event.

A logic event is an event generated as a result of some action or activity in the application's logic. If your application has an action that turns a switch (e.g., *Switch.up connect*), the generated event is a logic event.

Some events can either be user events or logic events, depending on how they are generated. *Switch.up make &* is one example, *Pushbutton in &* is another.

Some events can only be user events. For example, *Keyboard keyPress &* or *Circle click &*.

Transition to an Exclusive Mode

After Rapid has decided which modes should be activated (see preceding section), it performs a fixed sequence of operations. This section discusses the sequence of operations during and after a transition from one exclusive mode, ModeA, to another, ModeB.

The sequence is:

1. ModeA's exit activities
2. Transition's actions
3. ModeB's entry activities
4. ModeB's mode activities (once)

After Rapid has executed all the mode activities once, it begins the following process:

1. **Checks for events.**

   Rapid checks whether any logic event resulted from the preceding activities and actions. If so, and if the mode has a trigger dependent on such an event, Rapid triggers that transition. If the transition is to another mode, the current mode is exited (before any conditions are checked).

2. **Checks the conditions** (if the mode was not exited).

   The conditions are checked in the order in which they appear in the Logic Editor. As it checks, Rapid "makes a list" of those conditions that are true.

   As soon as it finds a condition for an external transition that is true, Rapid stops evaluating the mode's conditions. Therefore, any conditions further down the list will be ignored, even if they are true.

   If there are no true conditions, Rapid goes to step 4.

3. **Executes the internal actions (if any).**

   Rapid then goes back and executes the actions of those internal transitions whose conditions were true, according to the "list" it made earlier.

❖ *NOTE: After* **every** *internal transition, the mode activities may also be executed.*

The last transition that Rapid triggers is the external transition, deactivating the mode.

❖ *NOTE: Rapid executes the actions* ***after*** *it has finished checking the conditions.*

4. In the absence of user or logic events, Rapid executes the mode activities and begins again from step 1, above.

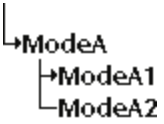
❖ *NOTE: If Rapid ever finds that more than one event has been generated, it will trigger the transition whose event occurred first, not the transition that appears first in the Logic Editor.*

Remember the following general rules:

- **After entering a mode, Rapid executes the entry and mode activities (once) before it begins to check the triggers.**
- **Rapid calculates which conditions are true *before* it executes the actions. Any actions that are executed along the way do not affect any conditions that were previously true.**

This can sometimes lead to unexpected results, as in the following examples.

Conflicting trigger and activity



In the application shown above, there is a transition from ModeA1 to ModeA2 based on the condition:

Integer = 0

ModeA1 has a mode activity:

Display.contents := 100/Integer

Question: What will happen when there is a transition to ModeA and Integer is equal to 0?

Answer: There will be a runtime error due to a division by zero.

Rapid tries to execute ModeA1's mode activity before evaluating its triggers. The activity involves a division by zero. The transition to ModeA2 (based on Integer being equal to zero) does not prevent the error.

Conflicting internal transitions

A mode has a set of internal transitions with associated actions:

TRIGGER	ACTION
Integer < 0	**Integer := 0**
Integer sqrt > 2	**Lamp on**

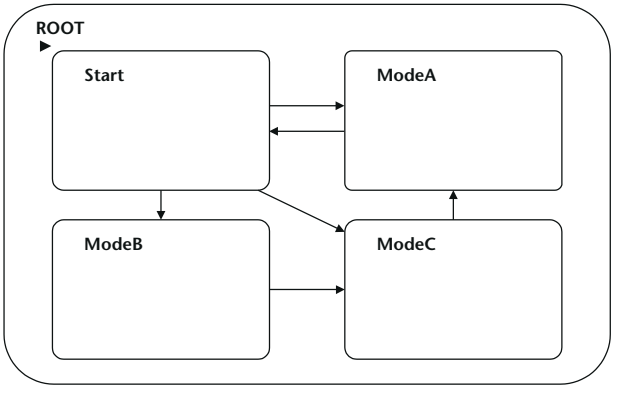
Question: What happens if Integer is less than zero when the mode is entered?

Answer: There will be a runtime error due to an invalid numeric operation (square-root of a negative number).

Rapid evaluates **both** conditions before executing the actions. The second condition contains an invalid operation, since Integer is less than zero. Although the first condition is true, its action does not "protect" the second condition.

Further examples

In the application shown below, there are four exclusive modes: Start, ModeA, ModeB and ModeC. The table shows the transitions from Start to the other modes.



Transitions from Start		
Destinations	**Triggers**	**Actions**
D:ModeA	& Switch1.up is connected	Sound start
internal	& Potentiometer.value >= 30	Lamp blink
D:ModeB	Pushbutton in &	
	& Potentiometer.value >= 50	Switch1.up make
D:ModeC	& Potentiometer.value >= 40	Switch1.up make
	Switch2.left make &	Potentiometer.value := 100

Example 1: A logic event is generated

In this example, on entry into Start, Rapid connects Switch2 to its left position and then pushes Pushbutton in (two logic events).

After completing one cycle of mode activities, Rapid checks whether there are any transitions triggered by the logic events that were generated. Rapid triggers the transition whose event occurred first, i.e., ModeC is activated.

Example 2: A condition of an external transition is true

On entry into Start, Rapid assigns 55 to Potentiometer's value.

After completing one cycle of mode activities, Rapid checks whether there were any logic events generated (in this example, there are none). Rapid now begins to check the conditions in the order that they appear in the Logic Editor. Rapid finds that the internal condition is true and that the condition of the transition to ModeB is also true. Note that the transition to ModeC's condition is not even checked since it appears lower down in the Logic Editor.

At this point, Rapid stops checking any further conditions (it has found a condition to an external transition that is true), and triggers all the true conditions that it has found till now. In our example, the Lamp starts to blink (the internal transition) and the transition to ModeB is triggered.

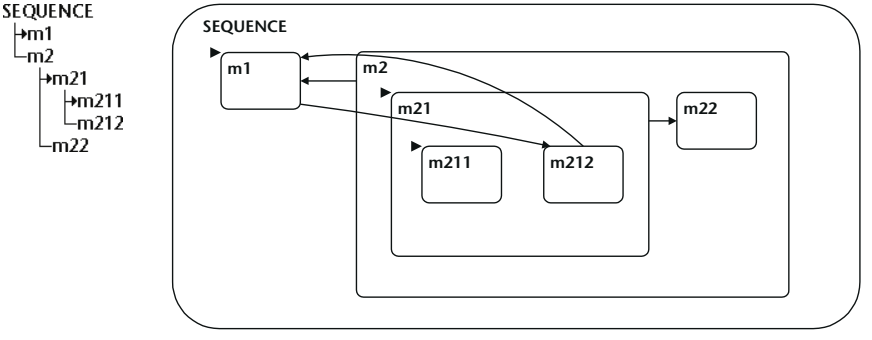
Entering a Subtree of Exclusive Modes

After a transition to a mode that is part of a subtree, the sequence of operations is as follows:

1. Entry activities of the mode that is highest up the tree.
2. Mode activities of the mode that is highest up the tree (once).
3. Entry activities of the active child mode.
4. Mode activities of the active child mode (once).
5. Entry activities of the active grandchild mode.
6. Mode activities of the active grandchild mode (once).
7. Entry and mode activities of further descendants down the tree, until the last mode activity of the last active mode is executed.
8. Rapid then begins to check the triggers for all the modes, starting from the top mode.

Example

The application shown below demonstrates the sequence of operations when entering a subtree:



When the m1-to-m212 transition is triggered, the sequence is as follows:

1. m1's exit activities.
2. m1-to-m212 actions.
3. m2's entry activities.
4. m2's mode activities (once).
5. m21's entry activities.
6. m21's mode activities (once).
7. m212's entry activities.
8. m212's mode activities (once).
9. Rapid then begins to check the triggers in all the modes, starting from m2.

Transitions within a Subtree

Conditions that are always true

There are two situations when a condition of an active mode can be continuously true:

- A parent-to-child transition.
- An internal transition.

When a condition of a parent-to-child transition continues to be true after the transition, Rapid does **not** repeat the transition. This allows you to build a "selector" routine (see p. 16–5).

When the condition of an internal transition continues to be true, Rapid repeats the transition. That is, an internal action is like a conditional mode activity.

Transitions between modes within a subtree

When there is a transition between modes within a subtree, the mode activities of the destination mode's ancestors occur before the destination mode's entry activities.

In the example shown on p. 16–14, when the m21-to-m22 transition is triggered, the sequence is as follows:

1. m21's exit activities.
2. m21-to-m22 actions.
3. m2's mode activities (once).
5. m22's entry activities.
6. m22's mode activities (once).
7. Rapid then begins to check the triggers in all the modes, starting from m2.

Exiting a Subtree of Exclusive Modes

When Rapid exits a mode that is part of a subtree, the sequence of operations is the opposite of that for entry into a subtree. That is, Rapid starts with the exit activities of the mode lowest in the subtree.

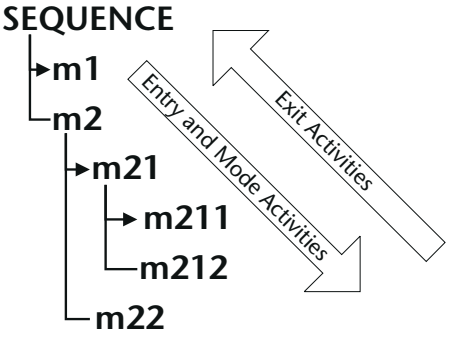
Example:

If the m212-to-m1 transition is triggered, the sequence is as follows:

1. m212's exit activities.
2. m21's exit activities.
3. m2's exit activities.
4. m212-to-m1 actions.
5. m1's entry activities.
6. m1's mode activities (once).
7. Rapid then begins to check m1's triggers for all the modes.

Entering and exiting a mode subtree

The diagram below illustrates the sequence of activities when entering and exiting a subtree:

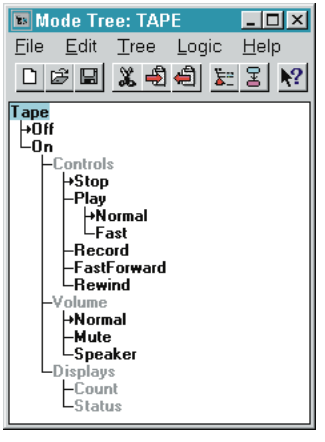


Transitions To and From Concurrent Mode Subtrees

There is **no** sequential relationship between concurrent modes. When Rapid triggers a transition to the parent of concurrent modes, all the concurrent modes are activated simultaneously. Within a concurrent subtree, however, the sequence of operations is as described above for exclusive modes.

Example:

The tape recorder application shown below has three concurrent modes (Controls, Volume and Displays) that are children of an exclusive mode, On:



When Rapid triggers the transition from Off to On: Controls, Volume and Displays are simultaneously activated. Each subtree, however, behaves as an independent subtree. For example, within the Controls mode, the sequence of operations among the exclusive child modes is as described previously.

OPTIMIZING RAPID APPLICATIONS

You can follow certain guidelines when building your application's logic to ensure that the application will behave the same way on different systems. Some of these guidelines were not critical in earlier versions of Rapid that ran under Windows 3.x, since there was no real multi-tasking. Following these guidelines should make your applications:

- Run faster.
- Utilize your computer's CPU resources more efficiently.
- Run at roughly the same speed on different CPUs.

Base transitions on a timer

When building your application's logic, **avoid** the following:

- Condition-only internal transitions that are always true.
- Condition-only transitions to an ancestor that are always true.
- Mode activities that are dependent on the speed of the mode activity being repeated (e.g., *integer1 changeBy: 1*).

  That is, any number (such as speed or position) that needs to change continuously should not be implemented as a mode activity. Do **not** use *changeBy:* as a mode activity.

  In all the above cases, it is better to use transitions that depend on a timer object generating the *tick* event. Otherwise, the behavior depends on the speed of your computer and the CPU load, and is therefore unpredictable.

Use the dragged event

Avoid long chains of dependencies through mode activities that affect graphic appearance during runtime. An example would be having the position of multiple graphic objects chained via mode activities:

Object1.position := Object2.position
Object2.position := Object3.position
Object3.position := Object4.position

The more concurrent modes and user-defined objects you add, the slower your application will appear.

A better solution is to use the *dragged* event to trigger the change in position:

Trigger: **Object4 dragged &**
Actions: **Object1.position := Object2.position**
Object2.position := Object3.position
Object3.position := Object4.position

If there is no logical difference, base transitions on events

For faster applications, it is better to base transitions on events. Event handling is **not** related to the number of active modes, it is related to the amount of logic dependent on the event. So, in a huge application, where dependencies on events are localized, the response to a particular event is not reduced due to the size of the application.

In contrast, the response to condition-only transitions depends on the number of active modes in the application.

❖ *NOTE: Because conditions are true over a period of time and events only occur momentarily, the use of a condition is often preferable.*

Use vector fonts in the scales of dials

The graphic performance of dial objects can be increased significantly by using vector fonts in their scales rather than Windows (True Type) fonts.

CHAPTER 17

The State Chart

The State Chart gives you a "bird's-eye view" of your application's logical construction and flow. The State Chart graphically displays the mode hierarchy, with child modes "nested" within their parent mode.

The State Chart's major advantage over the Mode Tree is that it displays every mode's transitions, in the form of arrows from one mode to another. In one glance, you can obtain information about the application, for which you would otherwise need to open the Mode Tree and the Logic Editor.

In addition, the State Chart's Logic Information pane displays a mode's destinations, triggers, activities and actions.

In this chapter you will read about:

- Recognizing exclusive and concurrent modes in the State Chart, and the transitions between them.
- Displaying the logic of a selected mode and/or transition in the Logic Information pane.
- Manipulating your view of the State Chart, to focus on a particular section of the application.
- Using the State Chart to define transitions and application logic (triggers, actions and activities).

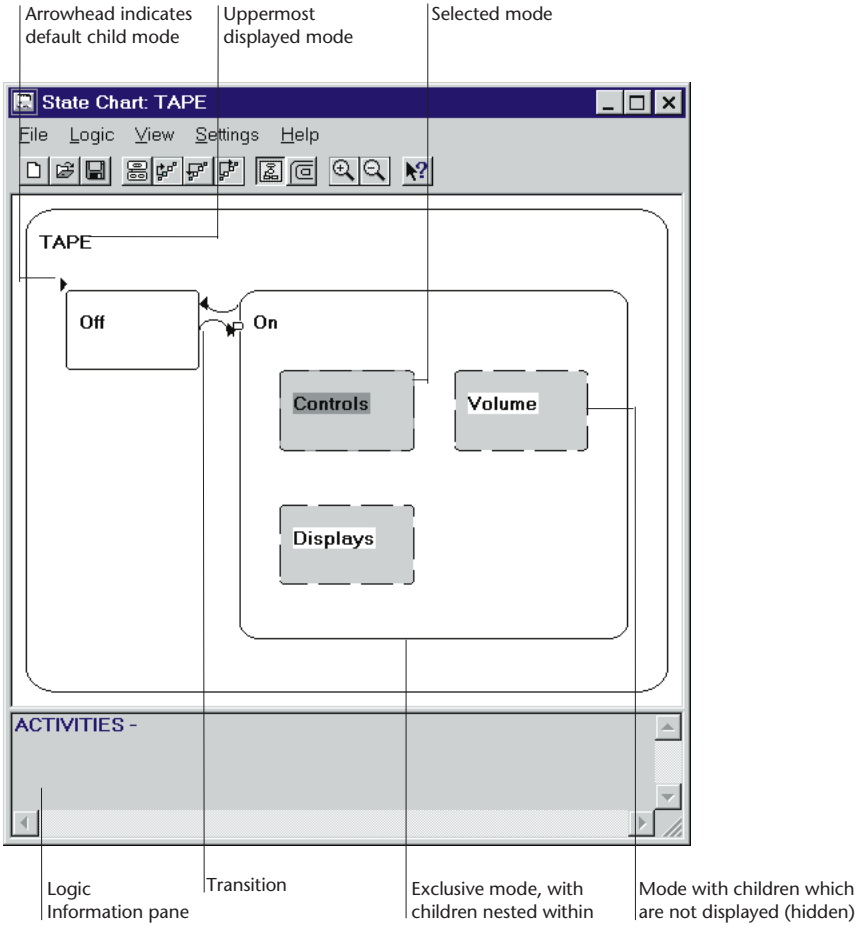
THE STATE CHART WINDOW

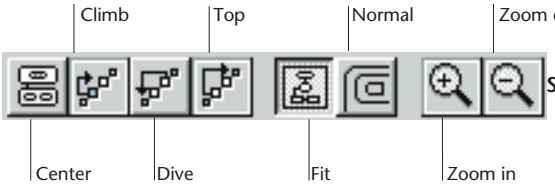
To open the State Chart:



- Choose Tools|State Chart from the Application Manager, or click the State Chart button.

The State Chart window is illustrated below. The sample chart presents the different mode and transition types:





State Chart toolbar buttons

Representation of Logic Elements

In order to understand the State Chart display, you must understand the way it represents logic elements (modes and transitions).

Modes

Modes are represented by rounded rectangles. The name of each mode appears inside the rectangle. Each of the mode types is represented as follows:

MODE TYPE	REPRESENTATION
Exclusive	Rounded rectangle with a black border and black name.
Concurrent	Rounded rectangle with a dashed black border and blue name.

The following rules apply to the presentation of modes:

- The name of the currently selected mode is displayed with a cyan background.
- Children are nested within their parents. That is, child modes are represented as smaller rectangles located within the larger rectangles of their parents.
- Default child modes are indicated by a black arrowhead outside the child mode rectangles.
- A mode with children that are not displayed is grayed.

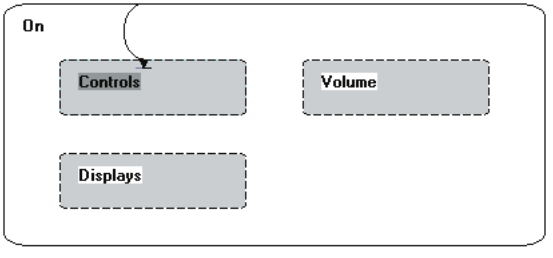
Transitions

Transitions are represented as arrows. These arrows begin at the source mode and point towards the destination mode. If the transition is **not** a default transition, a letter appears between the arrowhead and the destination mode, indicating the transition type. Each transition type is represented as follows:

TRANSITION TYPE	REPRESENTATION
Default	No letter
History	H
Deep History	P

❖ *NOTE: Transition type refers to the method of determining which of the destination mode's descendants are activated when the transition occurs. For a description of the different mode entry types, see Chapter 10: "Rapid Logic: The Basics."*

If either the source or destination mode is not displayed (see "Setting the Number of Displayed Levels" below), a cutoff (a short line perpendicular to the transition line) is used to terminate one end of the transition. In the illustration below, there is a transition from On to Play, an exclusive child mode of Control:



- If the source or destination is a descendant of a displayed mode, the cutoff is shown inside the closest displayed ancestor (as in the illustration above).
- If the source or destination is not a descendant of a displayed mode, the cutoff is shown outside the highest level mode displayed in the State Chart.

Logic Information Pane

The Logic Information pane displays all the triggers (internal and external), activities and actions of a selected mode or transition. This feature allows you to examine logic information without the need to open the Logic Editor. You can choose whether Rapid displays this pane or not. If you want the chart itself to be as large as possible, you should hide the pane.

To display the Logic Information pane:



- Choose Settings|Display Logic Info.

To view a mode's activities, internal trigger(s) and their actions (if any):

- Click the mode.

To view a transition's external trigger(s) and their actions (if any):

- Click the transition.

OPTIMIZING THE STATE CHART

A major function of the State Chart is to provide a highly readable display of the application's logic flow. With complex applications, however, even the State Chart can become crowded and unwieldy. Thus, a number of State Chart menu commands allow you to optimize the chart presentation.

❖ *NOTE: Rapid rebuilds and redraws the State Chart after any change in the settings.*

You can alter the following display characteristics:

- Font type, size and style of mode names.
- Length of mode names.
- Number of mode levels displayed.
- Display transitions (toggle).
- Number transitions
- Highlight transitions of selected mode (toggle).

Setting the Font

You may prefer different typefaces or font attributes depending on a number of factors, including complexity and size of the display. You can choose to display your chart with any font that is installed in your Windows system.



To change font attributes:

- Choose Settings|Options and click the Font button. Make the necessary changes in the Font dialog box.

❖ *NOTE: The Logic Information pane uses Rapid's system font.*

Setting the Length of the Mode Names

The lowest level mode names in a complex chart can be difficult to read because of crowding. When working with a complex chart, you may want to reduce the length of the displayed mode names. In other cases, you may want to display lowest level mode names in their full length, in order to distinguish between similarly named modes.

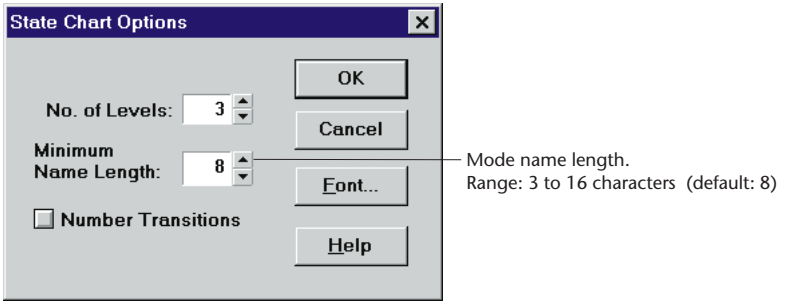
Mode name displays can be from 3 to 16 characters in length (the default length is 8). Mode names whose actual "drawing length" exceed the mode width are truncated, and terminated by a hash mark (#).

The name length determines the size of the mode rectangles when the State Chart is in Normal view. Mode rectangles are set to accommodate the widest possible displayed name length (that is, the number of characters multiplied by the width of "W", the widest letter). In Fit drawing mode, displayed mode names may be shorter than the maximum you specify, because the Fit drawing mode can reduce the size of the mode rectangles.

To change mode name lengths:



1. Choose Settings|Options to open the State Chart Options dialog box.



2. Overtype the number in the edit box or click the stepper box arrows.

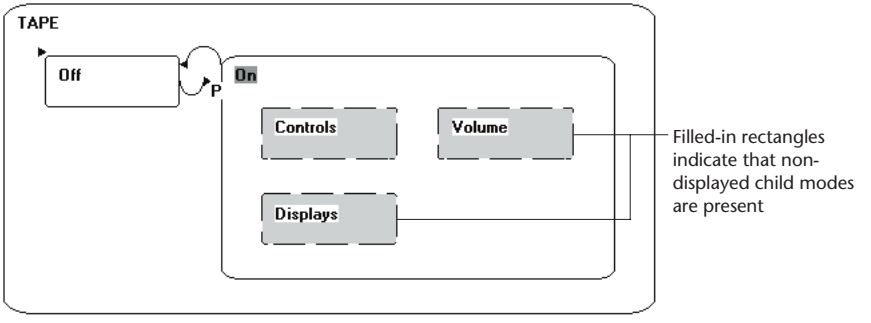
Setting the Number of Displayed Levels

A level consists of all modes that are the same number of generations removed from the root mode. The modes within a level need not be siblings, but siblings are always at the same level. Rapid can display from two to six different mode levels simultaneously, the default being three. The commands in the View menu, as discussed in detail below, determine which mode level is currently the highest visible level in the State Chart display.

You can determine how many levels (starting with one of the modes of the currently highest visible mode level) will be displayed in the chart. For example, if the levels setting is three and the highest visible mode level is that of the root mode, the State Chart will display:

1. The root mode
2. All of its children
3. All of its grandchildren

If the grandchildren also have children, then the grandchildren mode rectangles will be grayed, indicating that there are non-displayed modes below that level. This situation is illustrated below:



State Chart with hidden modes

To change the number of levels:

1. Choose Settings|Options to open the State Chart Options dialog box (shown on p. 17-7).
2. Overtype the number in the No. of Levels edit box or click the stepper box arrows.

Displaying Transitions

To display the modes' transitions:



- Choose Settings|Display Transitions. When enabled (i.e., preceded by a check mark), Rapid displays the transitions between the modes as arrows.

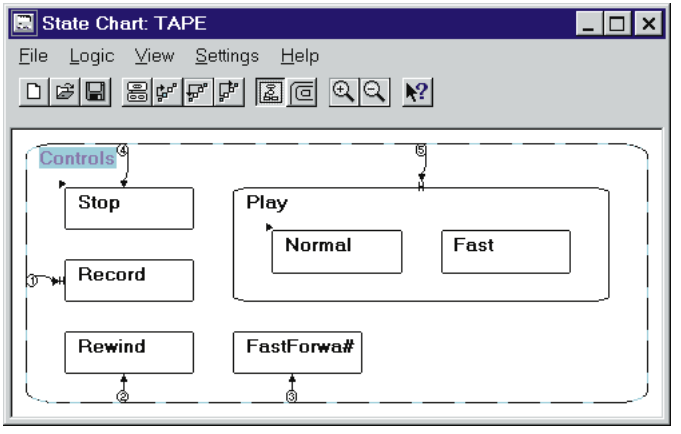
Numbering Transitions

For reporting and general reference purposes, you may find it useful to have the modes' transitions numbered in a chart.

To number the modes' transitions:

1. Choose Settings|Options to open the State Chart Options dialog box (shown on p. 17-7).
2. Click the Number Transitions checkbox.

When this option is checked, Rapid numbers the displayed transitions, as shown below:



State Chart with numbered transitions

3. To turn off the numbers, uncheck the option in the State Chart Options dialog box.

Highlighting Transitions

To highlight the transitions of the selected mode:



- Choose Settings|Highlight Transitions. When checked, Rapid highlights all transitions from the selected mode in blue. If you select a transition, Rapid highlights it in red.

NAVIGATING THROUGH THE STATE CHART

Once you have optimized the State Chart settings and display, you are ready to navigate through the chart to explore your application's logic. You can print the State Chart by choosing Reports|Modes|State Chart Plot in the Application Manager (see Chapter 24: "Generating Reports").

Choosing a View

You use the View menu commands to view the modes within a chart from different perspectives. The commands provided by the View menu are dupli-

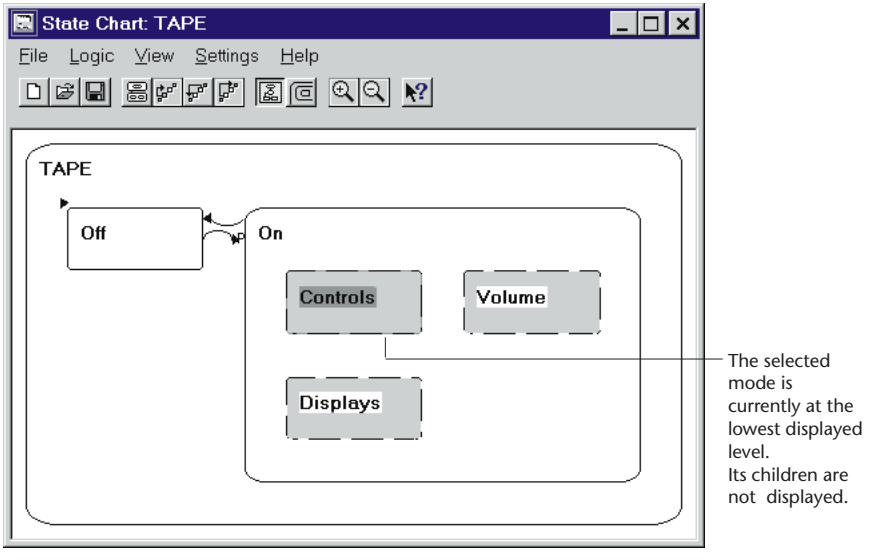
cated by the buttons in the State Chart toolbar. The highest mode level displayed in any chart is referred to as the top mode level.

Center

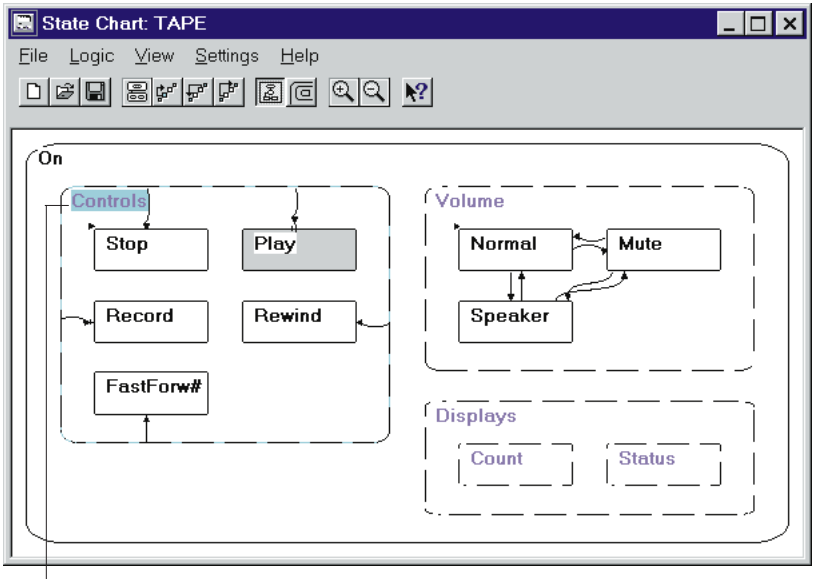


The Center command makes the level of the selected mode the second level in the State Chart display. Thus, you see the parent of the selected mode, the selected mode and its siblings, and as many levels of descendants as allowed by the Settings|Levels value.

The Center command helps you to focus your attention on the selected mode and its position in the logical hierarchy of your application. In the first illustration below, a selected mode is shown prior to choosing Center.



The effect of selecting Center is shown below. Note that the selected mode now has mode levels above and below it.



The selected mode is now at the second displayed level. Its children and its parent are visible.

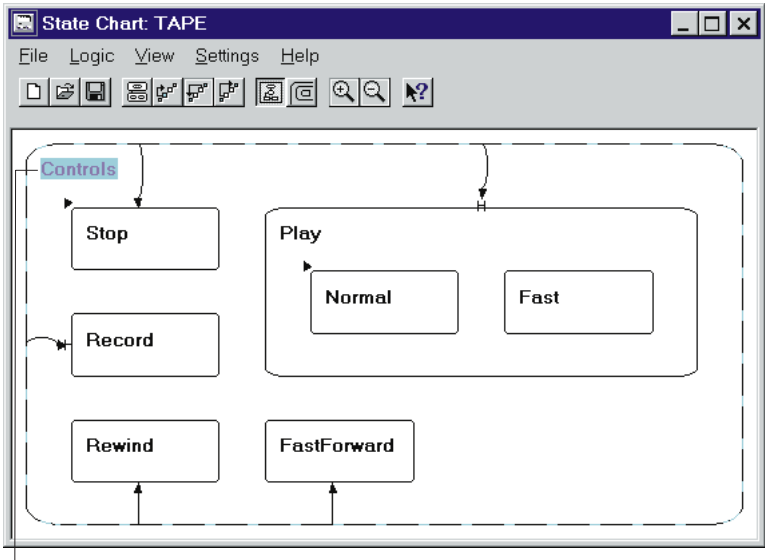
Dive



The Dive command descends to the level of the selected mode. It makes the selected mode the current top mode level, displaying it and its descendants (up to the number of mode levels selected for display).

The illustration on the previous page shows the chart prior to diving from the selected mode. The next illustration shows the chart after Dive is chosen. Note that the selected mode is now at the top level.

HINT:
You can also dive to and select a mode by double-clicking it.

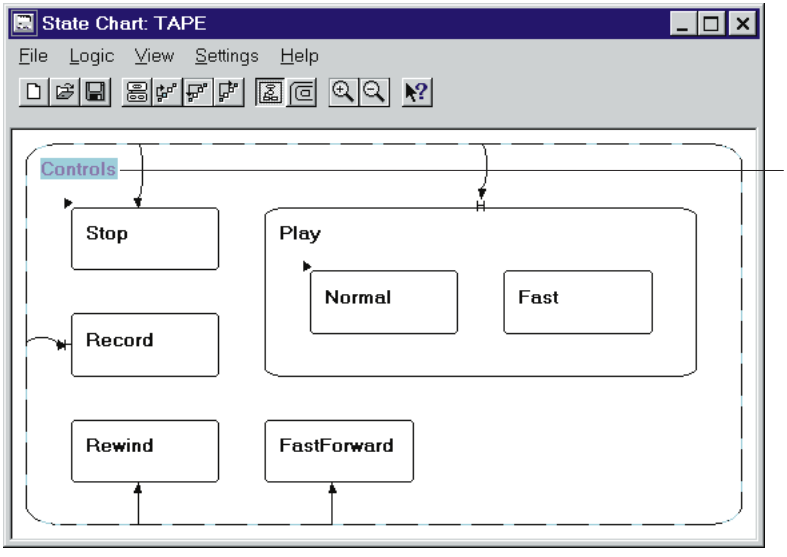


The selected mode is now the top displayed mode.

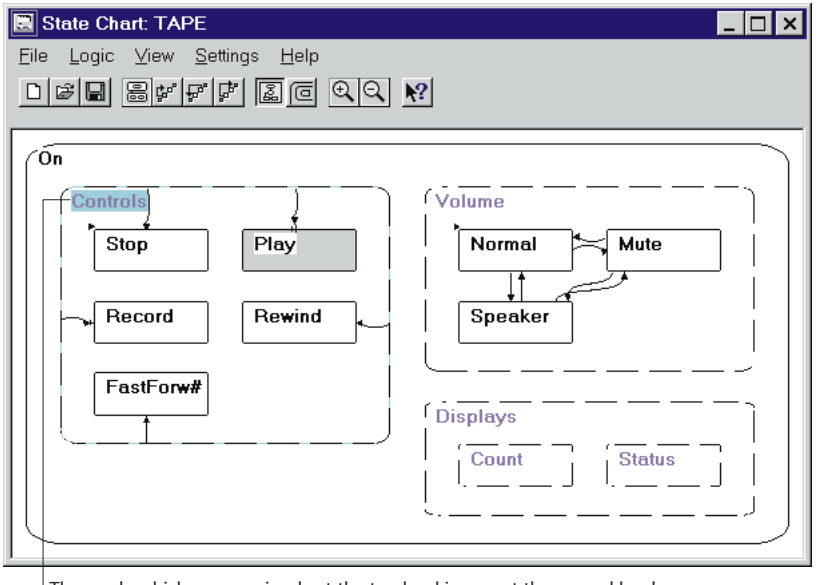
Climb



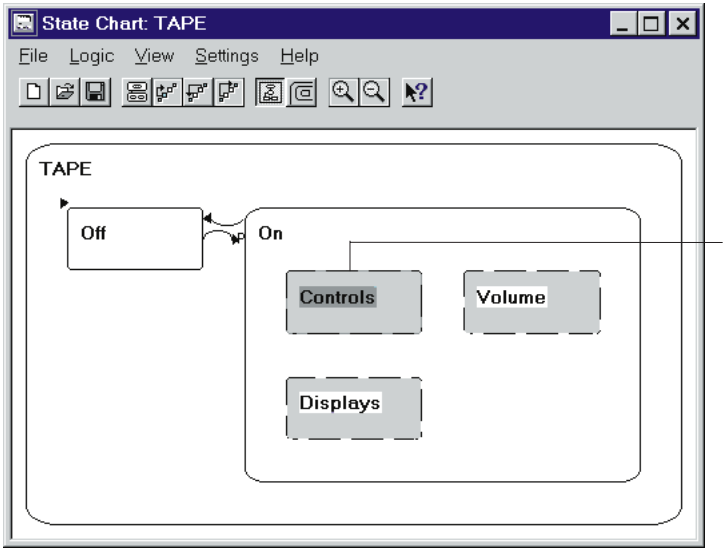
The Climb command ascends to the next highest level above the displayed top level of the application,that is, it changes the display so that the top mode is the parent of the current top mode. The illustration below shows the chart prior to climbing from the displayed mode level.



The next illustration shows the chart after Climb is chosen.



The next illustration shows the chart after Climb is chosen again.

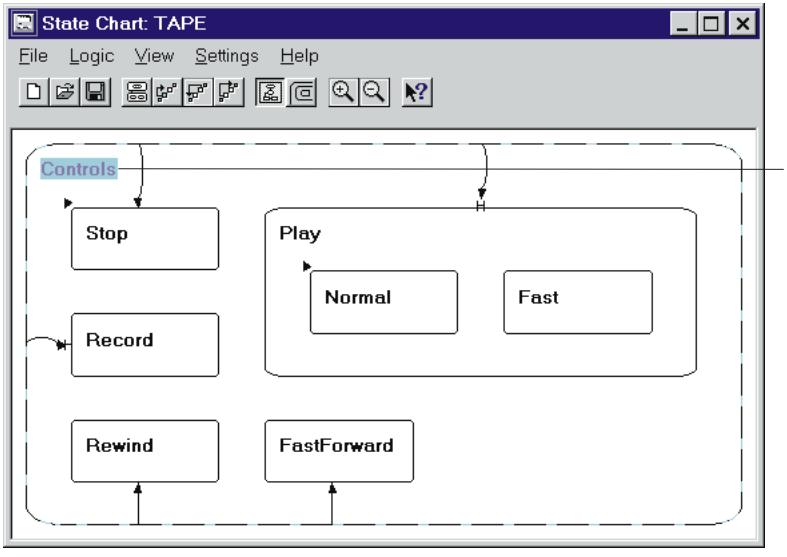


❖ *NOTE: After repeatedly choosing the Climb command, the selected mode may no longer be displayed. In that event, the selected mode's lowest-level displayed ancestor will be filled in cyan to indicate that a non-displayed descendant is currently selected.*

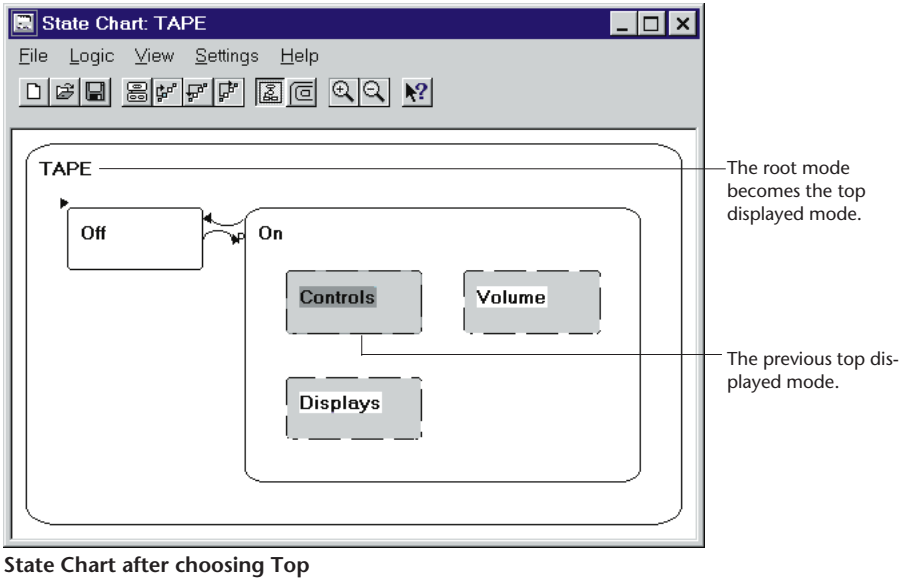
Top



The Top command ascends to the top level of the entire application, that is, it makes the root mode the top displayed mode. The entire application is displayed in the State Chart, down to the level determined by the levels setting. The illustrations below show the chart prior to and after choosing Top.



State Chart before choosing Top



State Chart after choosing Top

❖ *NOTE: After choosing Top, the selected mode may no longer be displayed. In that event, the selected mode's lowest-level displayed ancestor will be filled in cyan to indicate that a non-displayed descendant is currently selected.*

Zooming the State Chart

The View menu has several commands that allow you to alter the size of all or part of a chart within the State Chart window. They are useful for focusing on a section of the chart.

Zooming out

The Zoom Out command reduces the State Chart area by a factor of four (by halving both horizontal and vertical dimensions). Thus, you see four times as much of the chart area in one view.

To zoom out:

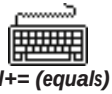



- Choose View|Zoom|Out or click the Zoom Out button.

Zooming in

The Zoom In command magnifies the State Chart area by a factor of four (by doubling both horizontal and vertical dimensions). Thus, elements of the chart appear at four times their original size.

To zoom in:

- Choose View|Zoom|In or click the Zoom In button.

Magnifying a Section

The Frame command zooms into a selected area of the chart.

To use Frame:



1. Choose View|Frame; crosshairs appear on the State Chart window.
2. Hold down the mouse button and drag a box around the area you want to magnify.
3. Release the mouse button. The State Chart zooms into the area you selected.
4. Click Normal or Fit to get out of this mode (see next page).

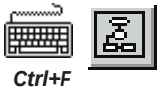
Redrawing the State Chart

There are two ways to alter how modes are drawn: Fit and Normal.

Fit view

Fit redraws the chart so that the uppermost mode level fits perfectly into the State Chart window.

To switch to Fit view:



- Choose View|Fit, or click the Fit button.

When the State Chart is in Fit view, all subsequent changes to the chart will be redrawn in this manner.

Normal view

Normal redraws the chart so that the mode rectangles for the lowest level of displayed modes are wide enough to contain the name length set in the Names dialog box (see p. 17–6). Normal is useful when you need to undo several changes in perspective.

To switch to Normal view:



- Choose View|Normal, or click the Normal button.

When the State Chart is in Normal view, all subsequent changes to the chart will be redrawn in this manner.

LOGIC OPERATIONS

When you view the State Chart in conjunction with the Mode Tree and/or the Logic Editor, it offers several alternative methods for developing some aspects of the application's logic.

Selecting a Mode

Some logic operations require you to select a mode. You can select the mode in the Mode Tree, as explained in Chapter 11, or use the State Chart.

To select a mode:

- Click its name in the State Chart window.

  After you select the mode in the State Chart, Rapid automatically selects it in the Mode Tree. Rapid also selects the mode in the Logic Editor's mode list.

Selecting a Transition

To select a transition:

- Click its arrowhead in the State Chart window.

  The transition and its source mode are in the State Chart. In the Mode Tree, the source mode is selected, and in the Logic Editor, the source mode and the transition destination are selected.

Creating Transition Destinations

You can carry out the procedure for creating a transition destination in the State Chart as well as in the Mode Tree.

To create a transition destination in the State Chart:

1. Select the source mode.
2. a) If the Logic Editor is not open: choose Logic|Transitions; the Logic Editor for the selected mode opens.

   b) If the Logic Editor is already open: click in the Destinations column.
3. Alt-click the destination mode in the State Chart.

   The mode is highlighted in the State Chart and the name appears in the Logic Editor's Destinations column.
4. Once you have finished creating the transition in the Logic Editor (i.e., choosing a transition type, accepting it), it appears in the State Chart.

❖ *NOTE: To get the State Chart into a configuration where the destination mode is visible, use the various commands (e.g. Top, Climb, Dive, and Center) or Shift-click a mode as described in the previous section "Navigating through the State Chart."*

See Chapter 11: "The Mode Tree," for more details on transition destinations.

Creating Activities, Actions and Triggers

If you are working in the State Chart and would like to build logic, the Logic menu provides the following commands: Activity, Action and Trigger, which open the Logic Editor in the appropriate editing mode.

To create logic elements from the State Chart:

1. Select a mode or transition in the State Chart.
2. From the Logic menu, choose either Activity, Trigger, or Action.

CHAPTER 18

User-Defined Functions

Rapid's user-defined functions are a great way to simplify an application's logic, with macro-like behavior. More significantly, they are also a basic building-block for the creation of user-defined objects (see Chapter 19: "User-Defined Objects").

In this chapter you will read about the following topics:

- What is a user-defined function, and what are the different types of user-defined functions?
- Building user-defined functions.
- Using the Function Editor—the section of the Logic Editor where you build and edit user-defined functions.
- Exporting user-defined functions for use in user-defined objects.
- Building user-defined functions that have arguments.

WHAT IS A USER-DEFINED FUNCTION?

A user-defined function is a block of Rapid logic that can be invoked as a single function within an activity, action or condition. You build a user-defined function in the Logic Editor's Function Editor, and it becomes a function of the application's *self* property. You can give a user-defined function any name (within certain limitations), and it can have one or more arguments.

You would build a user-defined function in the following situations:

- To repeat a section of logic in several places in the application.
- When you want a user-defined object to have a function that can be used in a parent application. (In this case the function must be "exported.")

There are two kinds of user-defined functions: activities and conditions. User-defined activities can also be used as actions.

User-Defined Activities

A user-defined activity is a collection of activities that can be run as a single activity (or action). For example, a function that is called *initializePanel* might contain the following activities:

```
PowerLamp on
Display.contents := 'Initializing...'
Sound start
CounterInteger changeBy: 1
```

If the application is called "MYAPPLIC," then the following activity would perform the entire series of activities shown above:

```
MYAPPLIC initializePanel
```

Such a user-defined function can be likened to a macro; that is, a single function that carries out a whole series of predetermined activities.

❖ *NOTE: A user-defined function cannot be used as a mode activity.*

User-Defined Conditions

A user-defined condition is a condition that is made up of a single or compound condition. For example, a condition that is called *is minutes* might contain the following compound condition:

```
& minutes >= 10 and minutes <= 20 and not minutes = 15
```

If the application was called “MYAPPLIC,” then the following condition would be equivalent to the compound condition shown above:

& MYAPPLIC is minutes

Like all Rapid conditions, user-defined conditions’ names must begin with the word “is” or “has.”

Multi-line conditions

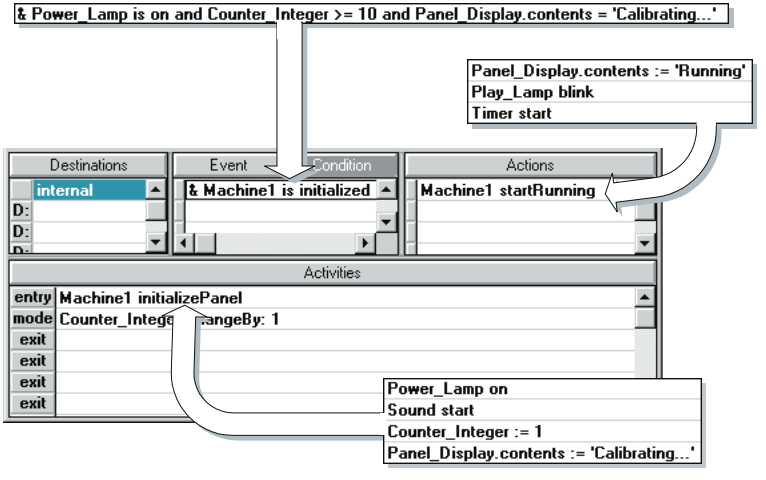
Unlike regular conditions, a user-defined condition can contain several lines of logic, each sub-condition being a separate line. The first line must start with “&” and each additional line must begin with either “and” or “or”. For example, the *is minutes* condition could contain one line of logic (as above):

& minutes >= 10 and minutes <= 20 and not minutes = 15

or it could contain three lines of logic:

& minutes >= 10
and minutes <= 20
and not minutes = 15

Multi-line conditions allow you to break up a multiple condition into its separate components. Instead of one long line of logic, you have several shorter lines, which can improve your logic’s readability.



Some user-defined functions and their logic

User-Defined Functions with Arguments

A user-defined function, like other Rapid functions, can have arguments. The arguments can be any Rapid object type. This gives the function added flexibility. Instead of the same series of activities taking place (or conditions being checked) with the same objects every time, the function can occur each time with different objects.

User-defined functions with arguments are described in more detail on p. 18–13.

BUILDING USER-DEFINED FUNCTIONS

You can build user-defined functions in two ways:

- With the Logic Editor's Make Function command.
- In the Function Editor.

If you want to build a simple user-defined function that just combines several activities (or actions) into one user-defined function, use the Make Function command. If you want to add arguments to the function, "export" it, or build a multiple-line condition, use the Function Editor.

Building User-Defined Functions in the Logic Editor

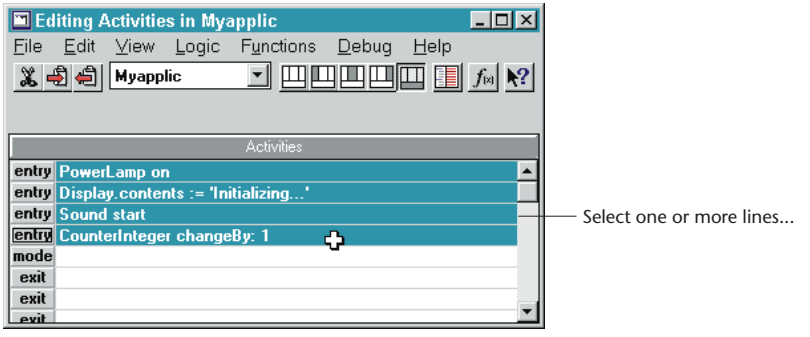
You can select one or more existing activities or actions, or a condition, and convert them into a user-defined function.

❖ *NOTE: You cannot convert mode activities into a user-defined function.*

To build a user-defined function from existing logic:

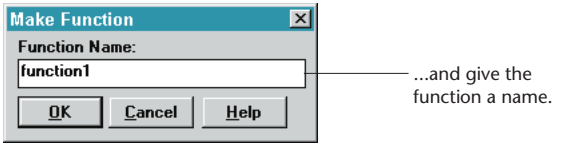
1. Open the Logic Editor.
2. Select one or more lines of logic.

   If you are making activities into a user-defined function, they must all be of the same type—you cannot combine an entry activity and an exit activity into one function.

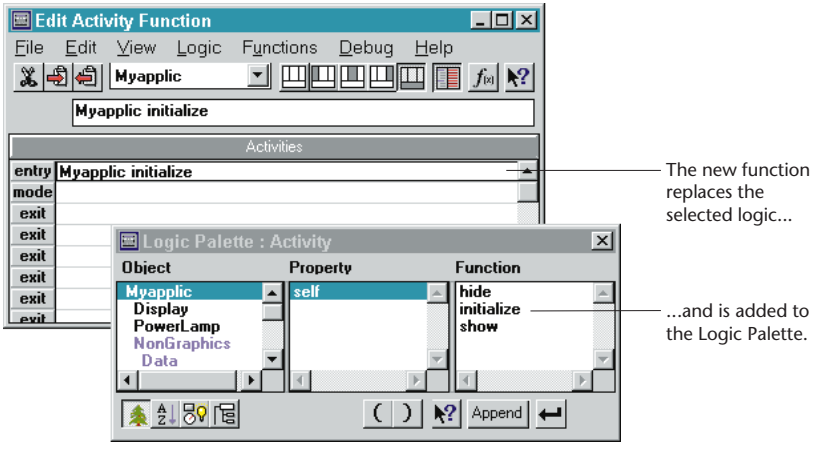


3. Either:
   Choose Edit|Make Function.

   Or:
   Right-click, then choose Make Function from the pop-up menu.



4. Name the new user-defined function. (See p. 18–17 for rules on naming functions.) Rapid replaces the selected logic with the user-defined function. The new function is added to the Logic Palette as a function of the root object:



Example:

In one mode, you have the following series of entry activities:

```
PowerLamp on
Display.contents := 'Initializing...'
Sound start
CounterInteger changeBy: 1
```

You can select these activities and choose Make Function. After naming the new function "initialize," the above activities are replaced by the activity:

```
MYAPPLIC initialize
```

The new user-defined function will carry out all the logic of the lines that you selected. You can now use this function just as you would use any other function, except that you cannot use it as a mode activity. In the Logic Palette, it appears as a function of the root object's *self* property.

Deleting objects involved in user-defined functions

If you delete an object that is involved in a user-defined function, any lines in the function containing this object will be "remarked" out with "\\". For example, a user-defined function might contain the following activities:

```
PowerLamp on
Display.contents := 'Initializing...'
```

If you delete the PowerLamp object, the relevant line is remarked out:

```
\\PowerLamp on
Display.contents := 'Initializing...'
```

If the object was involved in a user-defined condition, Rapid "comments" out the section that contained the object (with quotes) and appends *& 1 = 0* to the line. For example, if you deleted an object, Display1, the following multiple-line condition:

```
& Display1.contents = 'Hello'
and (Lamp1 is on
or Pushbutton1 is in)
```

would become:

```
& 1 = 0 "& Display1.contents = 'Hello'"
and (Lamp1 is on
or Pushbutton1 is in)
```

Building User-Defined Functions in the Function Editor

Building user-defined functions in the Function Editor gives you the added ability to:

- Add arguments to the function.
- Export the function, for use in a user-defined object.
- Assign a Help ID number to the function. A Help ID number allows the function to call a topic from a Help file.

Opening the Function Editor

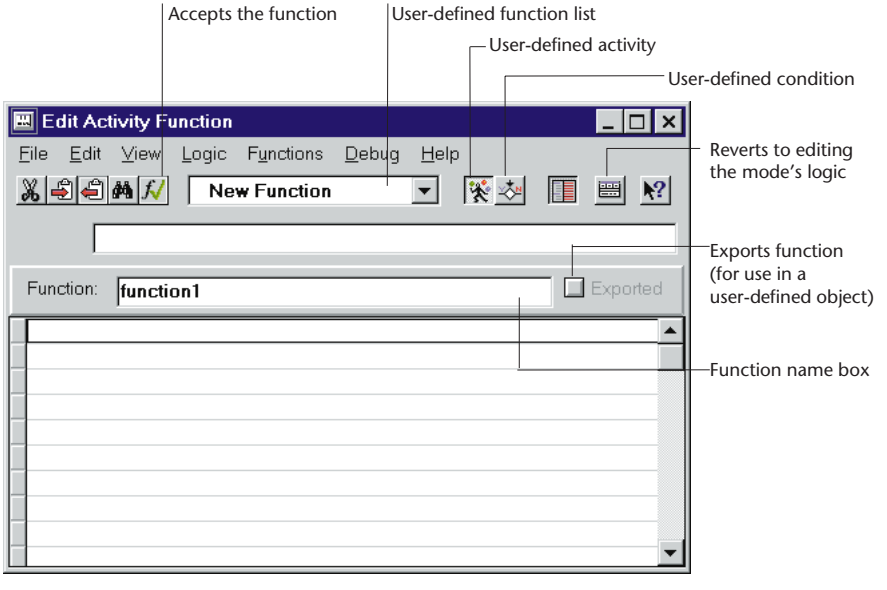
To open the Function Editor:

1. Open the Logic Editor.
2. You can open the Function Editor in one of two ways:



   - Click the Function Editor button.
   - Choose View|Function Editor.

   The Function Editor opens, ready for building a user-defined activity:



Building and accepting a function

To build a function:

1. If you are building a user-defined activity, either click the User-Defined Activity button or choose Functions|Activity Function.

   If you are building a user-defined condition, either click the User-Defined Condition button or choose Functions|Condition Function.

2. Type a name for the function in the Function name box. Use the usual rules for Rapid function names (see p. 18–17).

3. Click the first logic line, and build activities (or a condition), just as you build logic in the Logic Editor.

4. When you move to the next line—e.g., by clicking it or by pressing Enter—Rapid checks the syntax of the logic without accepting the function as a whole.

❖ *NOTE: Recursive calls to user-defined functions—a function that calls itself—will cause a fatal runtime error.*

While you build the logic of your user-defined function, Rapid checks the syntax of each line of logic as you go along. Rapid does **not** add the function to the application's logic (i.e., to the Logic Palette's function list) until you accept it.

To accept the function:

There are several ways to accept a new user-defined function:



- Either:
  Click the Accept Function button.

  Or:
  Choose Functions|Accept.

Rapid also accepts the function when you move the focus away from the current function. That is, when you select another function from the user-defined function list, switch back to editing the mode's logic, etc.

Building additional new functions

To build another user-defined function:

Either:
Select New Function from the user-defined function list.

Or:
Choose Functions|New.

SHORTCUT:
Right-click the user-defined function list and choose New.

❖ *NOTE: You cannot have two user-defined functions with the same name.*

Building a multi-line condition

Multi-line conditions allow you to break up a multiple condition into its separate components, that behave as one condition with its own name. Instead of one long line of logic, you have several shorter lines which can improve your logic's readability.

Multi-line conditions must have the following structure:

- The first line must start with "&".
- Each additional line must begin with either "and" or "or".
- You can place parentheses anywhere within the condition; for example, an opening parenthesis on one line and a closing parenthesis on the next.

With a multi-line condition you can build a condition, such as the one below, over several lines:

& Integer = 5 or (Button is in and Switch.position3 is connected)

The above condition could become:

& Integer = 5
or (Button is in
and Switch.position3 is connected)

Editing, Renaming and Deleting User-Defined Functions

Editing user-defined functions

To edit a user-defined function:

1. Open the Function Editor.
2. Select the function from the function list.
3. Edit the function as necessary.
4. Accept the changes.

Renaming user-defined functions

You can change the name of a user-defined function by editing its name in the Function name box.

To rename a user-defined function:

1. In the Function Editor, select the function from the function list.
2. Edit the name in the function name box.
3. Accept the changes.

   Rapid changes the function's name in the Logic Palette and wherever the function appears in the application's logic.

❖ *NOTE: For details on how to rename a user-defined function with arguments,* *see p. 18–19.*

Deleting user-defined functions

You can permanently remove a user-defined function with the Delete command.

To delete a user-defined function:

1. Select the function from the user-defined function list.
2. Choose Functions|Delete.

Exporting User-Defined Functions

Unless otherwise specified, all user-defined functions remain internal functions of the application. This means that if the application is converted to a user-defined object and added to a parent application, the user-defined object's functions will not be available to the application's logic. This situation is not a problem if the functions only involve the user-defined object's internal behavior.

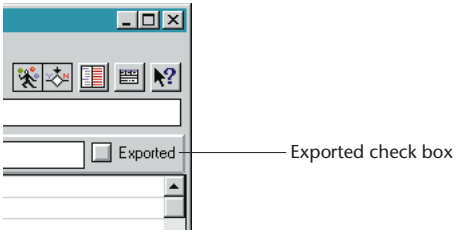
If the parent application needs to use the user-defined object's functions, the user-defined function must be **exported**. You can only mark the function as exported in the user-defined object itself. In other words, when you are building a user-defined function in an application, you should know whether the function will be needed in the future by a parent application.

❖ *NOTE: As a general rule, if you are building a user-defined object, always export its user-defined functions.*

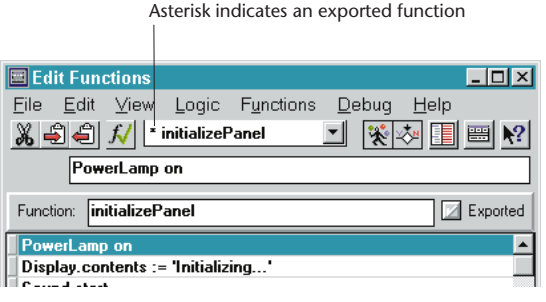
To export a user-defined function:

1. Select the function from the function list.
2. Either:
   Choose Functions|Exported.

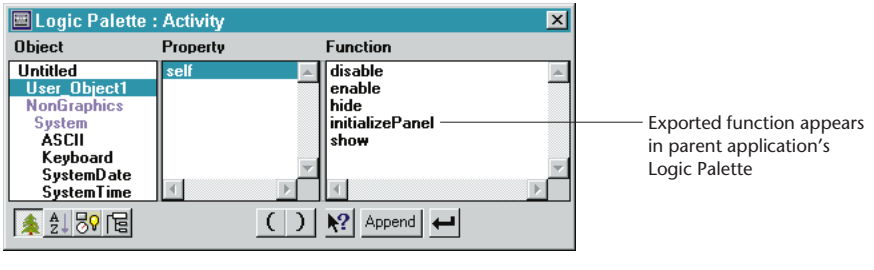
   Or:
   Check the Exported check box to the right of the function name box.



3. Click the Accept Function button. An exported function is indicated in the function list by an asterisk:



When the user object is added to a parent application, its exported functions are included in the Logic Palette:



Assigning Help ID Numbers

A user-defined object can have its own Help (*.hlp) file. A user object's Help file allows whoever adds the user-defined object to a parent application to get Help on its exported functions. You get Help on a user-defined object's functions in the same way as for any function: select it in the Logic Palette and press F1.

❖ *NOTE: You can get context-sensitive Help on functions and events, but not on properties. If you want Help on a property, choose Help|Search for Help On.*

For details on how to assign a Help file to a user-defined object, see Chapter 19: "User-Defined Objects."

For a user-defined function to be able to call a topic from a Help file, it must have its own unique ID number. This number must match the ID number of the topic in the Help file. (For details on how to build a Help file, and assign ID numbers to its topics, refer to your Help tool's documentation.)

To assign a Help ID number:

1. Select the function from the function list.
2. Mark the function as exported. (A function that is not exported cannot have a Help ID number.)
3. Choose Functions|Edit Help ID; a dialog box opens in which you assign a number.
4. Assign a unique number to the function.
5. Click the Accept Function button.

USER-DEFINED FUNCTIONS WITH ARGUMENTS

You can build a user-defined function so that it requires arguments, like other Rapid functions. The arguments can be any Rapid object type. An example of a standard function that has an argument is *changeBy:* — an arithmetic function that accepts either an integer or a number argument (depending on the context).

Defining all, or some, of a user-defined function's objects as arguments gives your function greater flexibility. You can use the same basic set of activities at several points of your application, but with different objects each time.

Examples of User-Defined Functions with Arguments

In the following two examples, *displayText* is a user-defined function (without arguments) that displays a particular string in a particular text display. It consists of the following activity:

Panel_Display.contents := Status

where Panel_Display is a text display and Status is a string. The examples illustrate different ways this user-defined function can be assigned arguments.

Example 1: All the objects as arguments

You can define both the text display and the string to be arguments of the function and change the function's name to *displayText:in:* In this case, the function could display a different string each time, in a different text display.

When you append this function from the Logic Palette, Rapid prompts you just as with any other function that has arguments:

MYAPPLIC displayText: <Text_Display> in: <String>

After you supply the arguments, the final activity could look like:

MYAPPLIC displayText: 'Welcome!' in: MainDisplay

or

MYAPPLIC displayText: ResultString in: ReadoutDisplay

Example 2: One of the objects as an argument

You could define only the string to be an argument. In this case, the function could display a different string each time—but in the same text display. The function's name would be *displayText:* (one argument only), and the prompt would be:

MYAPPLIC displayText: <String>

Converting all of the Objects to Arguments—"Generating Arguments"

You can automatically convert **all** the objects in your user-defined function to arguments. This is known as generating arguments.

To generate arguments:

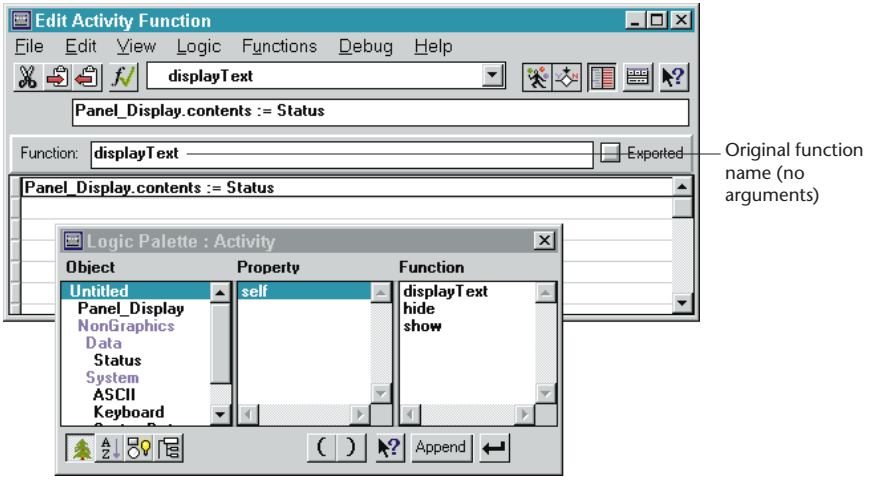
1. Select the function from the function list.
2. Choose Functions|Generate Arguments.
3. Accept the function.

The result of this command is that the function's name becomes the original name followed by a series of colons, keywords and arguments, one argument for each object involved in the function. The syntax for a user-defined function's name with arguments is:

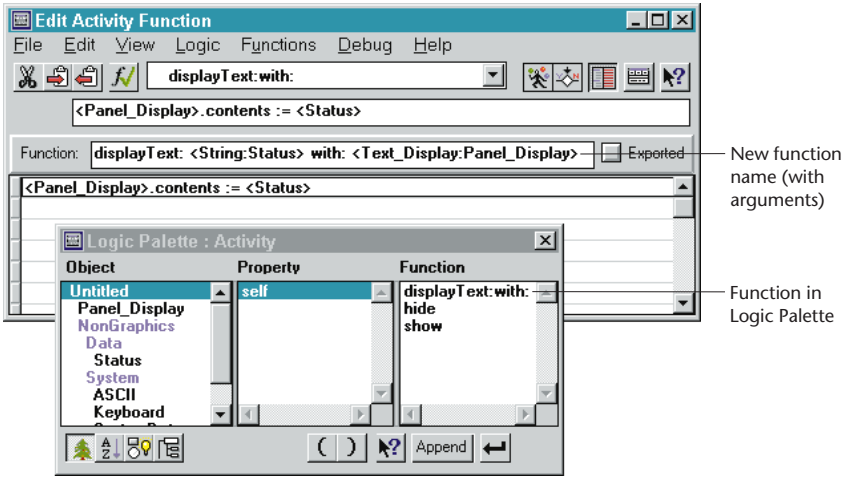
function: <Object Type:Object Name> with: <Object Type:Object Name>

Example:

The function *displayText* is shown below in the Function Editor:



After converting all the objects to arguments (using the Generate Arguments command), and accepting the function, the Function Editor looks as follows:



The result of generating arguments is that the function name has changed from:

displayText

to:

displayText: <String:Status> with: <Text_Display:Panel_Display>

The function's logic has changed from:

Panel_Display.contents := Status

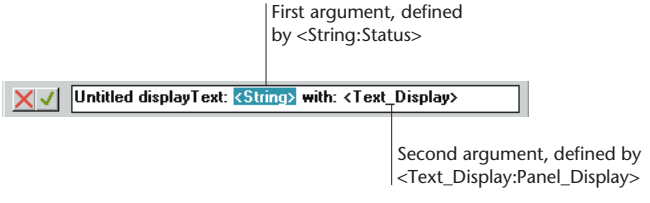
to:

<Panel_Display>.contents := <Status>

Note that the Panel_Display and Status objects are enclosed in angle brackets, denoting that Rapid treats them as variables.

In the function name, <*String:Status*> defines the first argument as being a string and <*Text_Display:Panel_Display*> defines the second argument as being a text display.

The actual name of the function in the Logic Palette is *displayText:with:*. When you use this function, Rapid prompts you as follows:



Improving the function name

After you have defined the function, there are two things you can do to make the function more readable:

- Change the order of the arguments by cutting and pasting them in the edit line.
- Change the default keyword ("with"), to help give a meaningful function name (see "Naming a user-defined function" on the next page.)

For example, the default function shown above is:

displayText: <String:Status> with: <Text_Display:Panel_Display>

You can change the keyword from *with:* to *in:*, thus:

displayText: <String:Status> in: <Text_Display:Panel_Display>

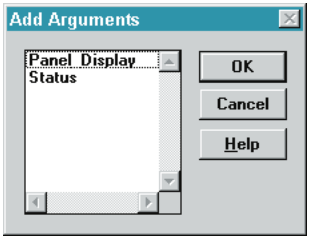
The final function name as it appears in the Logic Palette is *displayText:in:*.

Selecting Certain Objects to Convert to Arguments

You can choose to convert only **some** of the objects in your user-defined function to arguments, if you want some of them to remain as constants.

To add certain arguments:

1. Choose Functions|Add Arguments; a dialog box opens that lists all the objects in the user-defined function:



2. Select which of the objects you want to be arguments.
3. Accept the function.

   The name of the function becomes the original name followed by a series of colons, keywords ("with") and arguments; one argument for each object selected.

Example:

If you select Status to be an argument, the function name changes from:

displayText

to:

displayText: <String:Status>

This function would allow you to display different texts in Panel_Display.

Naming a user-defined function

You can give a user-defined function any name you like, within the following limitations:

- Rapid's usual rules for naming objects and modes, that is, no spaces are allowed (except in conditions after the "is" or "has"), the name must begin with a letter, etc.
- Conditions must begin with "is" or "has."
- Function names must begin with a lowercase letter.
- A function's argument must always be preceded by a keyword and a colon.
- Words that follow a colon must begin with a lowercase letter.
- The following words are "reserved," that is, you cannot use them as names of objects, modes, properties or functions: *action*, *and*, *begin*, *clear*, *end*, *entry*, *exit*, *has*, *internal*, *is*, *mode*, *not*, *or*, *resetValue*, *self*, *subroutine*, *x*, *y*.
- The following are reserved function names: *changeBy:*, *hide*, *hold:*, *holdCopyOf:*, *show*, *enable*, *disable*, *is hidden*, *is enabled*, *is overlapping:*, *is placedOn*.

Directly Typing a User-Defined Function's Arguments

If you prefer directly typing an application's logic to the "point-and-click" method, there is a quick way of adding arguments to a user object.

To type a user-defined function's arguments:

1. Type the function's name, including its arguments, directly into the Function name box.
2. Type the function's logic; define objects as arguments by enclosing them in angle brackets (< >).

Example:

You want a user-defined function called *displayText:in:* that displays a string in a text display. Both objects are to be arguments.

1. In the function name box, type:
   displayText: <String:Status> in: <Text_Display:Panel_Display>
2. Type the activity as follows:
   <Panel_Display>.contents := <Status>

Typing Arguments Without Adding Objects

You can use the typing method to define arguments **without adding objects to the Object Layout.**

You can type the function name and activity shown above, even if the application does not contain objects named "Status" or "Panel_Display." An object name within angle brackets, is recognized by Rapid as a "local" object —relevant only to the user-defined function in which it appears.

Typing the function's name and its logic allows you to define an argument **even if the application has no objects at all** of the argument's type. For example, you can type the above function name and logic, even if the application has no string or text display objects at all!

Question: How can you use a user-defined function if the application does not contain any objects of the type used as the function's arguments?

Answer: There are two possibilities, depending on the type of argument:

- If the argument is a basic data type, such as a string or an integer, when you use the function you can supply a string or an integer as the argument.
- If the argument is not a basic data type, such as a text display, you will not be able to supply the argument when you try to use the function (the application does not contain any objects of that type). That is, the function can be classed as "not yet implemented."

Planning for the future...

In the second case described above, you cannot use the function in the current application. However, creating a user-defined function that is not supported by your current application allows you to add potential functionality that can be realized by **future** development. Such a user-defined function will be usable:

- When the application is further developed (i.e., an object of the correct type is added).
- When the application is a user-defined object within a parent application that does contain an object of the correct type.

Renaming a User-Defined Function with Arguments

You can rename a user-defined function with arguments in the same way as a function without arguments (see p. 18–10). However, if you change the number or types of arguments, there are certain limitations.

Renaming without changing arguments

You can rename a function without changing its arguments by editing its main name or keywords. For example, you can change the function:

displayText: <String:Status> in: <Text_Display:Panel_Display>

to:

displayScript: <String:Status> at: <Text_Display:Panel_Display>

Changing the types of arguments

You cannot change the types of arguments unless you change the name as well. For example, if you wanted to change the display type in the previous function from a text display to a text window you can change the function's name from:

displayText: <String:Status> in: <Text_Display:Panel_Display>

to:

displayWord: <String:Status> in: <Text_Window:Panel_Display>

but **not** to:

displayText: <String:Status> in: <Text_Window:Panel_Display>

Changing to a name that already exists

If a function with the same name and arguments already exists, Rapid asks whether you want to overwrite the older function.

Changing the Number of Arguments

Increasing the number of arguments

Increasing the number of arguments creates a **new** function and does not replace the original one. You should edit the function's logic so that it contains objects referenced by the arguments that you added (otherwise the function will not do anything with the arguments it receives).

For example, the function:

displayText: <String:Status> in: <Text_Display:Panel_Display>

has the logic:

<Panel_Display>.contents := <Status>

If you wanted to rename the function *displayText:in:andIn:* so that it displays the same text in two text displays, you would rename the function to:

displayText: <String:Status> in: <Text_Display:Panel_Display> andIn: <Text_Display:Readout_Display>

You should change the logic so that it reads:

<Panel_Display>.contents := <Status>
<Readout_Display>.contents := <Status>

The result of this operation is that you have **two** functions; one called *displayText:in:* and one called *displayText:in:andIn:*.

Reducing the number of arguments

Reducing the number of a function's arguments creates a **new** function and does not replace the original one. You must edit the function's logic so that it no longer contains objects referenced by the arguments that you delete.

For example, if you want to rename the above *displayText:in:* function as:

displayText: <String:Status>

you must change the logic so that it reads:

Panel_Display.contents := <Status>

where Panel_Display is the name of a text display (or text window) in the application.

The result of this operation is that you have **two** functions; one called *displayText:in:* and one called *displayText:*.

CHAPTER 19

User-Defined Objects

Rapid has a large variety of built-in simulation objects that are the basic building blocks for most applications. You can, however, use the Rapid Object Development Kit (ODK) to develop supplementary customized objects externally from Rapid. These external objects are called RPX objects; once registered in Rapid, an RPX object cannot be distinguished from a built-in object. It resides as a DLL outside of Rapid, with the extension .RPX. For complete instructions on developing an RPX object, refer to the "Rapid External Object Development Kit: Programmer's Guide".

It is also possible to use Rapid itself to build user-defined objects. User-defined objects are encapsulated Rapid applications with the extension .UDO. Building a user-defined object allows you to use a set of objects and related logic over and over again, and in any application that you want.

In this chapter, you will read about the following topics:

- What is a user-defined object, and when to use one.
- How a user-defined object behaves in a parent application.
- Building a user-defined object.
- Adding properties, events and functions to a user-defined object.
- Defining a user-defined object's interface with a parent application—its buttons in the Object Palette and its Help file.
- Registering and adding a user-defined object to a parent application.

A short tutorial at the end of the chapter will help you practice the new skills you have learned.

WHAT IS A USER-DEFINED OBJECT?

A **user-defined object** is a Rapid application that is used as a discrete object within other Rapid applications. In other words, it is a collection of objects and logic that behaves as a single object. Like any object, it can have its own properties, functions and events.

Building a user-defined object is like building any other Rapid application; you add objects, modes and logic as usual, until your application looks and behaves just as you want. You can add properties, functions and events to the user-defined object as you build it.

After you save the application as a user-defined object, you can add it to another application just as you would add any standard Rapid object. An application that contains a user-defined object is called a **parent application**.

When to Use a User-Defined Object?

User-defined objects are useful in the following situations:

- When a unit of objects and logic is reused in several applications.
- When different people are working on separate subsections of a large application. Each developer can build a user-defined object and then each one can be added to the parent application.

How Does a User-Defined Object Behave in a Parent Application?

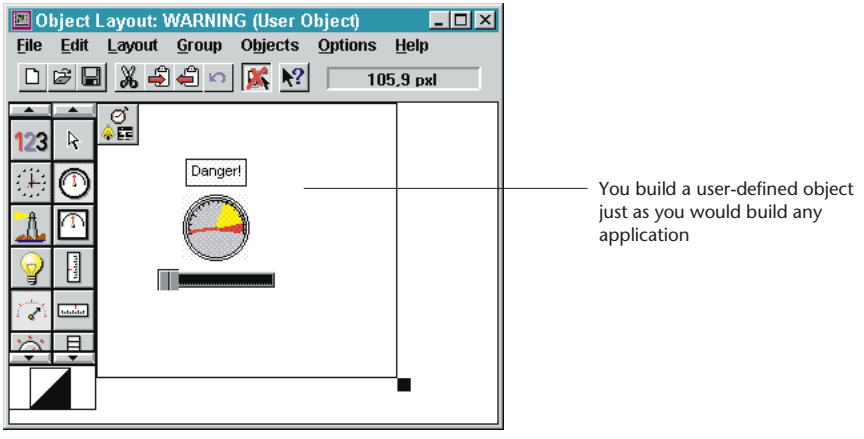
After you have placed a user-defined object in an application, it behaves as a closed, self-contained unit—you cannot edit its objects or its logic.

In the Object Layout

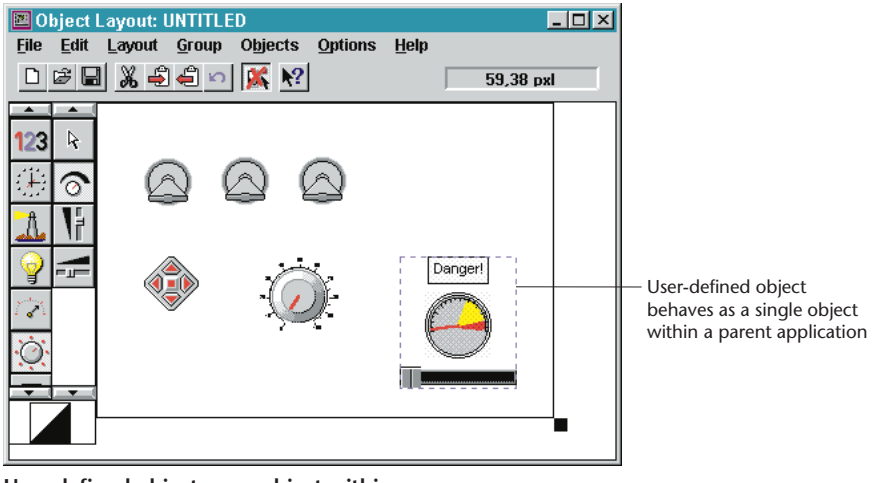
- All the user-defined object's component objects appear as they were defined when it was built.
- You move the user-defined object as a single object (like a group).
- You cannot resize the user-defined object.
- You cannot select individual objects within the user-defined object.
- If you added any properties and events, or any "exported" user functions to the user-defined object, you can decide which of them are available to the parent application.

In the Logic Editor

- The user-defined object appears in the object list as a single object under the object class you defined when building it.
- You can use the user-defined object's properties, functions and events that were made available in the Object Layout.



User-defined object as an application



User-defined object as an object within a parent application

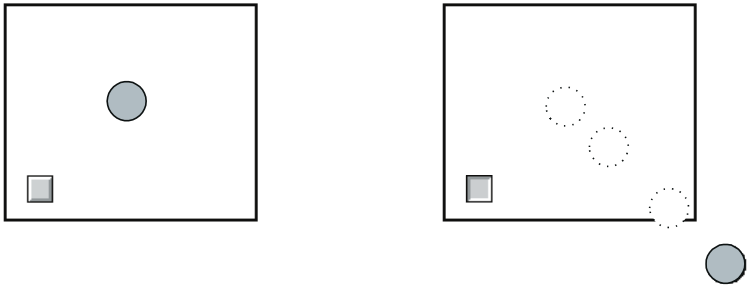
BUILDING A USER-DEFINED OBJECT

You build a user-defined object just like any Rapid application, that is, by adding objects to the Object Layout, modes in the Mode Tree, and so on.

Adding Objects to a User-Defined Object

When you build the layout of a user-defined object, take into account the following points:

- The size of a user-defined object is not determined by the size of the layout frame. Its size is the smallest enclosing rectangle that contains all the graphic objects.
- The position of an object within a user-defined object is relative to that user-defined object, not to the parent application (see p. 19–23).
- A dynamic object **can** move beyond the bounds of the user-defined object, if its *position* property is changed sufficiently.



An object within a user-defined object can move beyond its bounds

- A drag ‘n drop object **can** be dragged beyond the bounds of the user-defined object, unless the user-defined object itself is defined as a drag ‘n drop object.
- A user-defined object cannot contain a copy of itself.

Saving an Application as a User-Defined Object

You can save an application as a user-defined object at any time. Once the application is saved as a user-defined object, you can continue working with it as usual.

To save the application as a user-defined object:

1. Choose File|Save As User Object.
2. Save the application; a user-defined object has the extension *.udo*.

After you have saved an application as a user-defined object, the title bar appears as: "<application name> (User Object)". This title serves as a reminder that the application you are editing is a user-defined object; otherwise it looks identical to a regular (*.rpd*) application.

Opening a User-Defined Object

You can open a user-defined object for editing just like any regular (*.rpd*) application.

To open a user-defined object:

1. Choose File|Open.
2. Select the "User-Defined Object (*.UDO)" file type.
3. Select the file and click Open.

SHORTCUT:
To open a UDO embedded in a parent application, right-click on the UDO in the Object Layout and choose Open<udo name> from the popup menu. The parent application closes (with prompt for unsaved changes, if any) and the UDO opens in its place.

Runtime User-Defined Objects

As with a normal Rapid application, you can generate a runtime version of a *.udo* file. The runtime version is a "read-only" application that can only be run in the Rapid Reviewer. It has the *.rdo* extension.

To generate a runtime user-defined object:

- Choose File|Generate Runtime and enter a name and location for the *RDO* file.

You can add an RDO object to a parent application in the same way that you add a regular user-defined object; see "Adding and Registering User-Defined Objects" on p. 19–15.

❖ *NOTE: Later versions of Rapid might not be able to load an application that contains an RDO object. To load the application you will need to load the original user-defined object, save it in the new version and regenerate the runtime version.*

Nongraphic user-defined objects

A nongraphic user-defined object is one that contains only nongraphic objects. The only differences between nongraphic and graphic user-defined objects are those between standard nongraphic and graphic objects.

ADDING USER PROPERTIES, EVENTS AND FUNCTIONS

A user-defined object can either behave independently within a parent application, or be an integral part of the parent application's logic. Independent behavior is suitable if the user-defined object does not need to be involved in the parent application's logic. It would be a fully-functioning unit, within the larger application, just "doing its own thing."

If the user-defined object is going to be involved in a parent application's logic, you must take this into account **when you build it**. Rapid allows you to add several features to a user-defined object that allow it to interact with a parent application:

- User properties—allow a parent application to check and change the values of the user-defined object's properties.
- User events—allow a parent application to react to events in the user-defined object.
- Functions—user-defined functions that are "exported" by the user-defined object can be used in the parent application's logic.

Adding User Properties to a User-Defined Object

You can add user properties to a user-defined object in the same way you add properties to active primitive objects.

If you add a user property to a user-defined object, a parent application will be able to read or assign a value to the property. User properties allow the object to pass data to the parent application and vice versa.

Example:

You add an integer property to a user-defined object and call it "Counter." In the object's logic, the integer is increased by one whenever a pushbutton is clicked. When you add the user-defined object to a parent application, you can build a condition based on the value of Counter. If the object is called "Increase," the condition would look like:

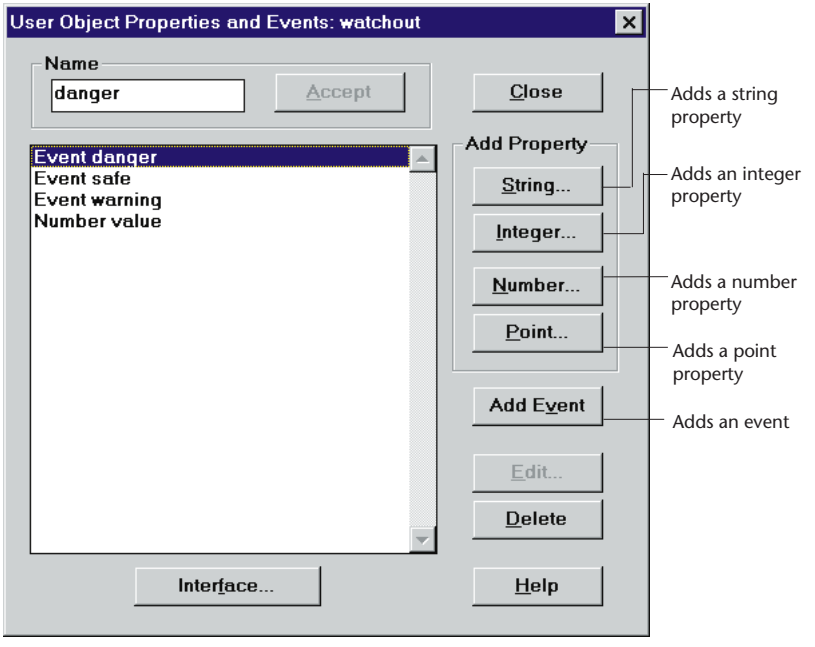
& Increase.Counter = 5

To add a property to a user-defined object:

1. Either:
   Choose Edit|User Object Properties.

   Or:
   Double-click anywhere in the Object Layout workspace (but not on an object). Click the More button in the root object's parameter pane.

   The root object's User Properties and Events dialog box opens.



A sample User Properties and Events dialog box

2. Click one of the property buttons, then define the property's value. You can add five types of properties; each contains a different type of data:

   - String
   - Integer
   - Number
   - Array
   - Point

   By default Rapid names the properties Integer1, String1, String2, etc.

To change a property's name:

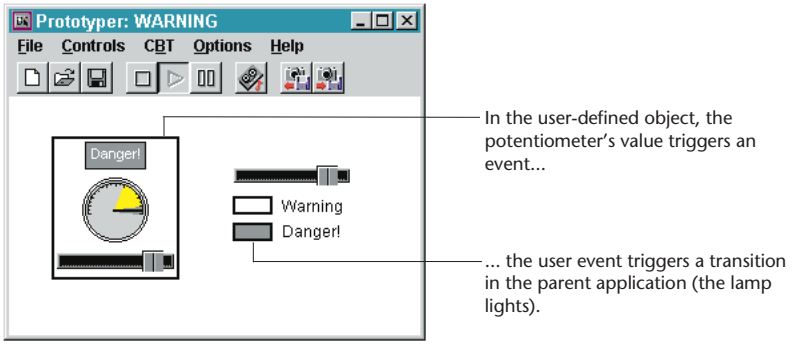
1. Select the property from the list.
2. Edit the name in the Name box and click Accept. (To be consistent with Rapid's normal properties, user properties' names should begin with lowercase letters.)

Adding Events to a User-Defined Object

Adding an event to a user-defined object is similar to adding an event object; it can be triggered by another event or a condition (with its *trigger* function). When the user event is triggered, it, in turn, can trigger a transition (based on the *triggered* event).

If you add an event to a user-defined object, it can trigger a transition in the parent application. These events appear in the Events column in the parent application's Logic Palette.

The series of events can be compared to a chain reaction. A normal trigger in the user-defined object triggers the user event; the user event then triggers a transition in the parent application.



Example:

You add a user event called "danger" to a user-defined object. In the logic, the event is triggered when a potentiometer's value exceeds a certain limit. The logic looks like:

Trigger: **& Potentiometer.value > 90**
Action: **danger trigger**

When you add the user-defined object to a parent application, you can trigger a transition using the object's *danger* event that lights a lamp. If the object is called "Increase" the logic would look like:

Trigger: **Increase danger &**
Action: **DangerLamp on**

❖ *NOTE: If you add a user event to a user-defined object, don't forget to trigger it somewhere in its logic. If the event is never triggered, the parent application will never receive the event and the transition will never occur!*

To add a user event to a user-defined object:

1. Either:
   Choose Edit|User Object Properties.

   Or:
   Double-click anywhere in the Object Layout workspace (but not on an object). Click the More button in the root object's parameter pane.

   The root object's User Properties and Events dialog box opens.
2. Click the Add button in the Event group; the event is added immediately to the list (an event does not have a value that you need to define).
3. Change the event's name (optional).

Adding Functions to a User-Defined Object

If you want the parent application to directly influence the behavior of the objects within the user-defined object, add functions. If a user-defined object has functions, **and those functions are exported**, they can be used in the parent application.

To add functions to a user-defined object:

1. Build a function in the Function Editor.
2. Export the function.

Refer to Chapter 18: "User-Defined Functions" for information on how to build and export user-defined functions.

DEFINING A USER-DEFINED OBJECT'S INTERFACE

As you will see in "Adding and Registering a User-Defined Object," on p. 19–14, there are two methods for adding user-defined objects to a parent application. This section discusses two options that will make it easier to add the object to an application and to work with it after you have added it:

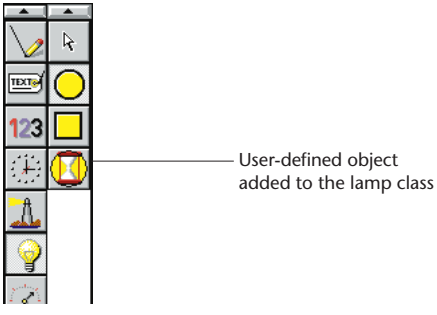
- You can add new buttons to the Object Palette, allowing you to add the user-defined object just like any of Rapid's "built-in" objects. (This option is only relevant if you "register" the user-defined object.)
- You can assign a Help (*.hlp*) file to a user-defined object. This file allows you to get help on the object's functions, if it has any.

For details on how to register a user-defined object and on how to add it to an application's layout, see "Adding and Registering a User-Defined Object" on p. 19–14.

Defining the Object's Buttons in the Object Palette

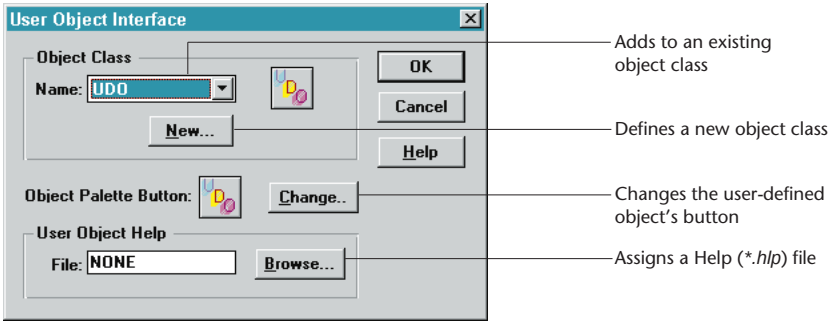
When you register a user-defined object, Rapid adds a new button to the Object Palette. Rapid allows you to define:

- The object class in which the button will appear. The class can be one of the existing classes (switches, indicators, etc.,) or you can create a new class with its own button. Unless you specify otherwise, Rapid adds a default new object class button.
- The bitmaps of the user-defined object's button and class button (if you defined a new class).



To set up the new button(s):

1. Open the User Object Properties dialog box (choose Edit|User Object Properties).
2. Click the Interface button; the User Object Interface dialog box opens.



To select the object class:

Either:
Select an existing class from the Name list; the bitmap of the class's button is shown.

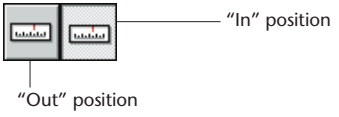
Or:
Click the New button to define a new class and button.

To define the user-defined object's button:

1. Click the Change button.
2. Select a suitable bitmap (*.*bmp* or *.*ico* file).

Designing a bitmap for a user-defined object's button(s)

A *.bmp* bitmap for an object's button must be 64 pixels wide by 32 pixels high. The first 32 pixels of the width appears when the button is in its "out" position. The second half of the bitmap appears when the button is in its "in" position. The bitmap must be a 16 color bitmap.



Bitmap of a linear indicator button

To design a new bitmap:

1. Open the file *objbttn.bmp* (in Rapid's */objects* folder) in any paint program. This bitmap is a blank object button template that you can use to design a new button bitmap.



2. Edit the image. Do not delete the bitmap's border because it gives the image its 3-dimensional appearance.
3. Save the image as a 16 color bitmap with a new name.

If you use an *.ico* file to define the bitmap, it need only be 32×32 pixels. The icon should represent the "out" position; Rapid will automatically generate the "in" position.

Assigning a Help File to the User-Defined Object

You can get help on any logic function by selecting a function in the Logic Editor and pressing F1. The Help topic that appears is taken from Rapid's main Help file, *rapid.hlp*. If your user-defined object has its own user functions (and events), you can create a custom Help file that will give Help on those functions.

To assign a Help file to a user-defined object:

1. Open the User Object Properties dialog box (choose Edit|User Object Properties).

2. Click the Interface button; the User Object Interface dialog box opens.
3. Either:
   Click the Browse button and select the Help file.

   Or:
   Type the Help file's pathname into the File text box.

❖ *NOTE: If someone else is going to work with your user-defined object, don't forget to provide the Help file together with the UDO file.*

Creating your Help file

There a number of commercial programs available for creating Windows Help files. For directions on how to create a Help file, refer to your program's documentation. To give your user-defined object Help the same appearance as the main Rapid Help, Emultek has provided two template files:

userobj.hlp — a sample Help that shows the appearance of a function Help topic.

userobj.rtf — a sample RTF file that you can use as a template for your Help topics.

You must place the user-defined object's Help file in the Rapid folder.

ADDING AND REGISTERING A USER-DEFINED OBJECT

There are two ways to add a user-defined object to an application's Object Layout:

- Adding it straight from a file (without registering it).
- Registering it and then adding it like any other object. (When you register a user-defined object, it is added to the Object Palette.

❖ *NOTE: A user-defined object cannot contain a copy of itself.*

You can add a runtime user-defined object (RDO) to a parent application in the same way that you add a regular user-defined object.

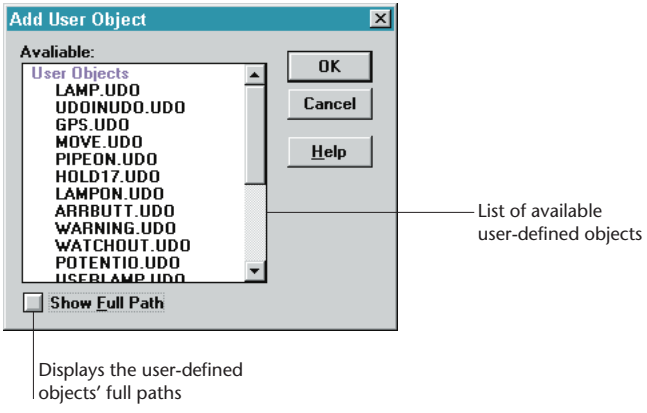
❖ *NOTE: Later versions of Rapid might not be able to load an application that contains an RDO object. To load the application you will need to load the original user-defined object, save it in the new version and regenerate the runtime version.*

Adding a User-Defined Object (without Registering it)

You can add a user-defined object directly to your layout without registering it. This method does not add the object to the Object Palette. It **does** add it to the object list in the New Object dialog box (which you open by choosing Objects|Add).

To add a user-defined object:

1. Choose Objects|Add User Object.



2. Select a user-defined object from the Available list and click OK.
3. Position the crosshairs on the layout where you would like the object to be drawn and click. The user-defined object appears with a blue dotted outline. Unlike a group, you cannot resize it.

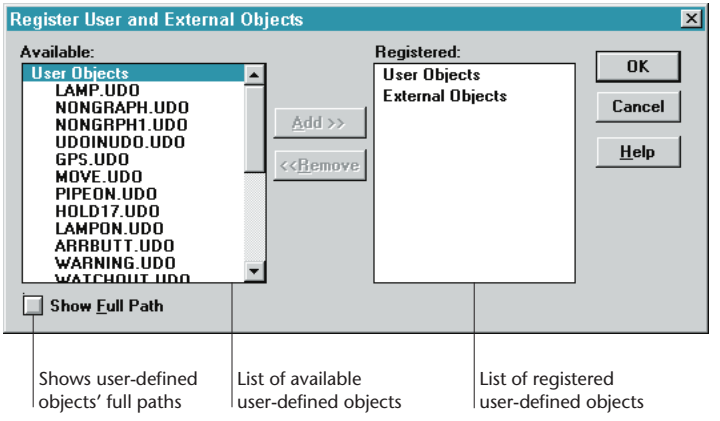
Registering a User-Defined Object

You can register a user-defined object and then add it to the layout. Registering a user-defined object adds it to the Object Palette and the New Objects dialog box.

❖ *NOTE: If you save your settings when you exit Rapid, the registration will be saved in the rapidxx.ini file and will be permanently part of your Object Palette.*

To register a user-defined object:

1. Choose Objects|Register; the Register User and External Objects dialog box opens. In this dialog box you register (and unregister) user-defined objects and external objects (**.rpx*)[1].



2. Select a user-defined object from the Available list (on the left).
3. Click Add; the object is added to the Registered list (on the right).
4. Click OK; Rapid registers the object and adds its button(s) to the Object Palette.
5. Make sure that the Application Manager's Options|Save Settings on Exit is checked. (If it is not, the user-defined object will not be registered next time you open Rapid.)

1 Rapid external objects (RPXs) are Rapid-compatible "plug-in" objects, created outside Rapid. They are written by a programmer in a language such as C++.

How does Rapid build the list of "available" user-defined objects?

The list of available objects that Rapid displays is the result of a search in the following folders:

- The current folder (the folder where the current application is located).
- Rapid's *objects* folder.
- Rapid's *applics* folder.
- Rapid folder (wherever *Rapidxx.exe* is located).
- The folders that appear in your path.

If a user-defined object does not appear in the Available list, it is because it is not located in any of the above folders. Move it to one of the search folders and try to register it again.

Adding a registered user-defined object

To add a user-defined object after registering it:

- Add the object in the same way as any other object. That is, click its button in the Object Palette or choose Objects|Add.

DEFINING A USER-DEFINED OBJECT

After you have added a user-defined object, you can define its parameters like any other object, except for its size. If it is a graphic object, double-click it to open its parameter pane. If it is a nongraphic object, double-click the non-graphics icon to open the Edit Nongraphics dialog box.

Dynamic and Drag 'n Drop User-Defined Objects

You can define a graphic user-defined object to be dynamic. A dynamic user-defined object has the *position* property that allows you to change the position of the object as a whole in the application's logic.

You can define a user-defined object to be drag 'n drop, in which case it obeys the same rules as a drag 'n drop group:

- If the user-defined object itself is drag ‘n drop, you cannot drag any of its graphic objects which have been defined as drag ‘n drop.
- You can click **anywhere** within the user-defined object’s bounding box to drag it (including clicking on a primitive object or on an empty space).
- Dropping an object **anywhere** within the user-defined object’s bounding box generates the *dropped* event.
- A user-defined object within another user-defined object will be treated like a single object.

User-defined object’s empty space is not transparent

Like groups, the empty space within a user-defined object’s bounding box is **not** transparent. If the parent application has a clickable object, such as a pushbutton, which is located **behind** the user-defined object, you will not be able to click it in the Prototyper.

❖ *NOTE: If the user-defined object is hidden, the clickable object is not affected.*

In the Object Layout, be sure to bring the clickable object to the front (or send the user-defined object to the back).

Choosing Which Properties, Events and Functions to Include in the Application

A user-defined object, like any other object, has a specific set of properties, events and functions that govern its behavior. (You defined these elements, if any, when you built the user-defined object.)

When you add the user-defined object to an application, all its user properties, user events and exported functions are available for use in the parent application’s logic. That is, they appear in the Logic Palette and you can include them in the application’s triggers, actions and activities.

Adding properties, events and functions

If you want to include a logic element that you previously removed, or if you have added a new logic element to the user-defined object and you want to include it in the application, you need to add a user-defined object's property, event or function to the application:

To add a user-defined object logic element to the application:

1. Open the user-defined object's parameter pane and click the More button; the User Object dialog box opens (described on the next page).
2. Select the items you want to add and click the Add button. The logic is now available to the application.

Removing properties, events and functions

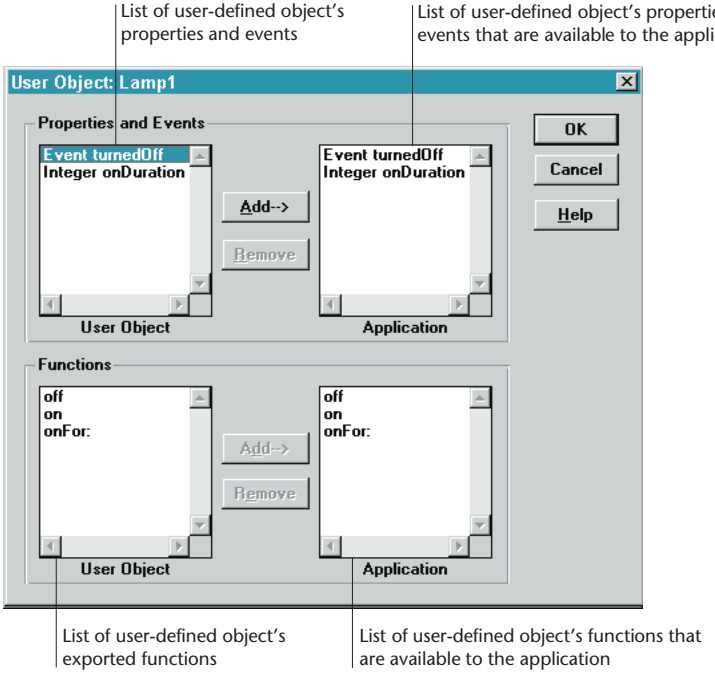
Sometimes, you might not need all of this logic in your application. For example, a function that is useful in other applications that contain the user-defined object, might be irrelevant in the current application.

You can choose to "remove" any or all of these logic elements. "Removed" logic is not available in the Logic Editor. (It is not deleted from the user-defined object, it is just unavailable to the parent application.) For example, if you try and type logic that includes a removed event, Rapid informs you that it does not recognize the event.

To remove logic from the parent application:

1. Open the user-defined object's parameter pane and click the More button; the User Object dialog box opens.
2. In the Application lists, select the items you do **not** want to appear and click the Remove button. The removed logic is no longer available to the application.

❖ *NOTE: If you remove items that are already used by the application's logic, Rapid **deletes the related logic.***



Deciding what logic to include

Usually, you should only include those properties, events and functions that you need, in order to make it easier to change the user-defined object in the future.

Example:

You are building an application with a user-defined object called Math that includes all sorts of useful mathematical functions. All you need in the current application is the *volumeOfSphere:* function, but the user-defined object also has a function *calculateX:* <*Integer: integer1*>. It would be a mistake to include *calculateX:* in the application.

Later, Math might be updated and improved, and *calculateX:* might be redefined to *calculateX:* <*Number: number1*>. Every time you load the application, Rapid will warn you about an incompatible user-defined object.

If you include only what you really need, your application will be more robust in the face of any changes to the underlying user-defined object

(see "Changing a User-Defined Object that is Part of an Application," below).

Changing a User-Defined Object that is Part of an Application

It is possible to edit a user-defined object like any other application, but you must carefully consider the implications of doing so. If the object is already part of one or more parent applications, editing it might create problems in those applications.

Changes that do not affect the parent application

There are several kinds of changes to a user-defined object that do not affect the parent application:

- Adding objects.
- Changing the **internal** logic of the user-defined object.
- Editing bitmaps, changing the position of objects, or any other cosmetic changes.

Adding new properties, events and functions

If you add a new property, event or exported function to a user-defined object, the new logic is not automatically available to a parent application that contains the object. You can add the new logic though, as described on p. 19–19.

❖ *NOTE: If you add a new instance of the user-defined object, all its properties, events and functions (including new ones) are available by default.*

Changing properties, events and functions

When editing a user-defined object, take extra care if you change any properties, events or exported functions. Changing them might have serious consequences on an application that contains this user-defined object.

If you delete or rename a property, event or function, you will not be able to run the application in the Prototyper. Before you are able to run it, you need to adjust either the application's logic or the user-defined object itself.

Making changes to user-defined objects—summary

When you add a new instance of a user-defined object, all its properties, events and exported functions are imported by default. However, you can choose to remove specific logic elements.

Once a user-defined object is part of an application, any changes to its logic elements, such as new properties, are **not** automatically imported to that application. You have to decide for yourself whether or not to import the new features.

Example:

Your user-defined object has a property that is involved in a parent application's logic. If you rename the property, you will not be able to run the parent application. Rapid detects that the property with the old name is no longer present; Rapid treats the renamed property as a completely new property. To enable you to run the application, you need to:

1. Open the parent application and add the object's renamed property (as described on p. 19–19).
2. Modify the parent application's logic so that it depends on the new name instead of the old.
3. "Remove" the property with the old name from the parent application (as described on p. 19–19).

MORE ABOUT USER-DEFINED OBJECTS

User-defined objects in parent applications are actually Rapid applications nested within other applications. Since user-defined objects can themselves contain other user-defined objects, advanced use of this feature can lead to whole hierarchies of nested Rapid applications. This section discusses some of the more complex implications of this hierarchy:

- Is the position of an object within a user-defined object relative to the parent application or to the user-defined object?
- What happens to objects that are generated by user-defined objects (using the *holdCopyOf:* function of a holder or array of objects)? Who "owns" it, the parent application or the user-defined object?
- How do a parent application and a user-defined object view each other's objects? Can they directly manipulate each other's objects?
- How can the keyboard be used within user-defined objects?

The fundamental rule governing the behavior of objects is that **objects always belong to the application that created them.**

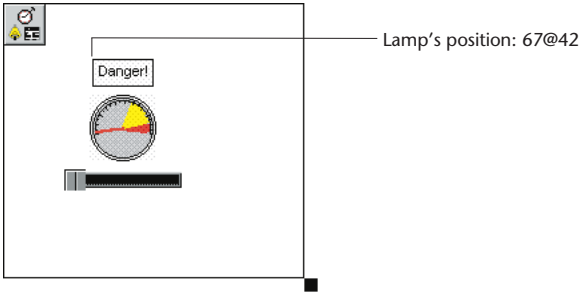
Position of Objects within User-Defined Objects

An object's position is always relative to its parent application, even if the logic that manipulates the object is in another user-defined object. As a result, the position of an object that is within a user-defined object is relative to that user-defined object, not to the parent application.

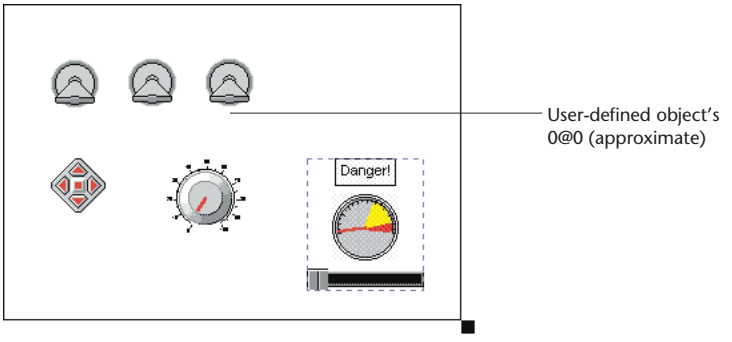
Example:

In the example shown below, the danger lamp's position within the user-defined object, is 67@42. If the lamp was dynamic, and the user-defined object moved it to 0@0, the lamp would not move to the parent application's 0@0, but to the user-defined object's 0@0 (somewhere below the rightmost switch).

❖ *NOTE: The user-defined object's coordinate system takes into account its entire layout frame, which includes empty space around the objects.*



User-defined object



Application containing user-defined object

Example:

An application contains a user-defined object whose top-left corner is at (10@10). The user-defined object changes the position of one of the parent application's objects (with one of its user-defined functions) to (0@0). The object will move to (0@0) in the application—that is, (-10@-10) in the user-defined object's coordinate system.

Scale units of user-defined objects

A user-defined object retains its own scale units, even if it is added to an application that has a different scale. Any of the user-defined object's functions that involve point data (changing an object's position, for example) use the user-defined object's own scale units.

Example:

A user-defined object whose scale units are centimeters has a function *move:upBy:* that moves a specified object by a specified amount. If the user-defined object is added to an application whose scale units are meters, the application's activity *move: Circle upBy: 2* moves the circle by the equivalent of two **centimeters** (not meters).

Objects Generated by User-Defined Objects

A user-defined object that contains a holder or an array of objects can generate objects using the *holdCopyOf:* function. Generated objects belong to the application (or user-defined object) whose holder or array generated them.

For example, an object that was generated by a user-defined object's holder belongs to the user-defined object, even if the object that was copied belongs to the parent application.

"Visibility" of Objects within a User-Defined Object

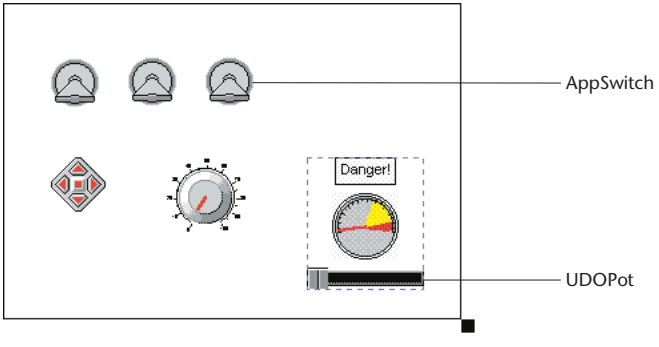
A parent application can only see a user-defined object's properties, events and exported functions. The objects within a user-defined object are invisible to the parent application, that is, a user-defined object is like a "black box" to its parent.

However, a parent application is able to supply its own objects as arguments to an embedded user-defined object's functions. The result is that a parent application's objects are potentially visible to its children, but a child's objects are invisible to a parent.

The same rule applies to nested user-defined objects: a user-defined object within another user-defined object can manipulate its parent's objects as well as the objects of the (grandparent) application. The objects of the (grandchild) user-defined object are invisible to anything higher up the hierarchy.

❖ *NOTE: One of the implications of this limitation is that a parent application's holder or array of objects cannot hold objects within a user-defined object (see next section).*

Example:



In this example, the parent application cannot manipulate the object "UDOPot." Indeed, it is unaware of any of the objects within the user-defined object. The user-defined object can, however, directly manipulate the object "AppSwitch" with a user-defined function such as *turnOnSwitch:*. In the application's logic, the activity would look like:

User_Object1 turnOnSwitch: AppSwitch

Arrays of Objects and Holders in User-Defined Objects

User-defined functions provide an interface between an application and its embedded user-defined objects. When a user-defined object manipulates a parent application's object (via a user-defined function), the object is said to be "passed" from the parent to the user-defined object. As explained above, an application can pass its own objects down to any embedded user-defined objects.

The creative use of user-defined functions in combination with arrays of objects and holders, allows an object in one application (or user-defined object) to be held by an array or holder in another application. In certain cases it also allows a user-defined object to pass one of its objects up to a parent application. This section describes how this is accomplished, and the limitations imposed by Rapid.

How to pass an object from an application to an embedded user-defined object:

A user-defined object has a holder ("UDOHolder") whose default object is a round lamp. The parent application has a round lamp ("ParentLamp"), and a suitable holder ("ParentHolder").

The user-defined object has two functions that allow it to manipulate the parent's lamp—one directly, the other using a holder:

Function: **grab: <Round_Lamp:Lamp1>**
Logic: **UDOHolder hold: <Lamp1>**

Function: **grabHolder: <Round_Lamp_Holder:LampHolder>**
Logic: **UDOHolder hold: <LampHolder>**

After the application performs either of the following actions, the user-defined object's holder will be holding the application's lamp, and can manipulate it in its logic:

UDO grab: ParentLamp

or

UDO grabHolder: ParentHolder

❖ *NOTES: When manipulating the application's object via the user-defined object's holder, the coordinate system in force is that of the application.*

If a generated object is passed through grabHolder, it only ceases to exist if both the parent ***and*** *the child no longer hold the object.*

How to pass an object from an embedded user-defined object to an application:

The above user-defined object also has a function that allows the parent to hold the same object that is held by the child's holder:

Function: **getObject: <Round_Lamp_Holder:LampHolder>**
Logic: **<LampHolder> hold: UDOHolder**

This function allows the parent to supply its own holder as an argument. After the application performs the action *UDO getObject: ParentHolder*, the application's holder and the user-defined object's holder will be holding the same object:

❖ *NOTES: Be careful that the parent's holder never holds an object from the user defined object. Violating this rule results in a runtime error. The only objects that can be passed with this method are objects that have been previously passed to the user-defined object by the parent. For the parent, the user-defined object is a black box, and its objects are not visible.*

Similarly, an application's holder cannot create a ***copy*** *of objects within a user-defined object. Violating this rule also results in a runtime error.*

The Keyboard and User-Defined Objects

Question: Is it possible to have two or more keyboard objects in one application?

Answer: Yes, if the application contains one or more user-defined objects.

Every user-defined object (being itself a Rapid application) has its own keyboard object and keyboard buffer. If the keyboard is not involved in the logic, there is no room for confusion between the parent application's keyboard and the user-defined object's keyboard—the latter simply isn't used.

❖ *NOTE: If you have never used the keyboard object, please read pp. 13-36 to 13-39.*

If you want to use the keyboard to display text in a user-defined object that is part of a parent application, there are two ways to implement it:

- By activating the user-defined object's keyboard.
- By activating the parent application's keyboard.

Activating the user-defined object's keyboard

If you choose to use the user-defined object's keyboard, make sure that the object has the necessary internal logic to recognize keystrokes. That is, its logic must include a transition based on the *keyPress* event.

In addition, you can build user-defined functions that activate and deactivate the user-defined object's keyboard. If you export these functions, you will be able to activate and deactivate the user-defined object's keyboard **from a parent application**.

Activating the parent application's keyboard

If you choose to use the parent application's keyboard, the logic that recognizes keystrokes must be in the parent application. The user-defined object should have the following logic:

- A string property.
- A mode activity that continuously assigns the value of the string property to the text display's contents.

The parent application should assign the value of the keyboard buffer to the user-defined object's string property. The logic to display keystrokes in the user-defined object's text display is summarized below:

User-Defined Object

Mode: **TextDisplay.contents := UDO.contents**

Parent Application

Trigger: **Keyboard keyPress &**
Action: **UDO.contents append: Keyboard getCharacter**

Recording the Events of User-Defined Objects

When recording a user-defined object's events in the Prototyper's Recorder, the object's unique name is given as:

```
<User-defined object name>:<Object name>
```

For example, if a user-defined object called "System" had a pushbutton called "Pushbutton1," the pushbutton being pushed in would be recorded as:

```
System:Pushbutton1 # $self # in
```

The same syntax for identifying the unique name of an object within a user-defined object is used when sending an external event to Rapid from an authoring program (via the CBT channel).

The syntax for nested user-defined objects is:

```
<UDO1>:<UDO2>:...<UDOn>:<Object name>
```

For example, if a user-defined object called "Simulation" contained another user-defined object called "System," which, in turn, contained a pushbutton called "Pushbutton1," the pushbutton being pushed in would be recorded as:

```
Simulation:System:Pushbutton1 # $self # in
```

TUTORIAL: BUILDING A USER-DEFINED OBJECT

In the first part of this tutorial you will build a user-defined object and add it to a parent application. The object will be a "timed lamp" that automatically turns itself off after a set period.

In the second part of the tutorial, you will make changes to the object to see how each change is reflected in the parent application.

Part 1: Description of the User-Defined Object

To comply with the initial definition of the user-defined object, it has to include the following objects:

- Lamp
- Timer. (The lamp will turn off when the timer reaches zero, or ticks.)

It will have the following properties, functions and events:

NAME	DESCRIPTION
onDuration	An integer user property that sets how long the lamp is lit.
off	A user-defined function that turns the lamp off.
on	A user-defined function that turns the lamp on for the length of *onDuration*.
onFor:	A user-defined function that turns the lamp on and will turn it off again after a given period.
turnedOff	A user event that is generated when the lamp turns off.

Building this user-defined object involves several steps:

- Adding the objects and saving the application as a user-defined object.
- Building the basic logic (the lamp turns off when the timer ticks).
- Adding an integer property (*onDuration*) and an event (*turnedOff*) to the root object.
- Creating three new user functions (*off*, *on* and *onFor:*).
- Defining the object's class and its Object Palette button.

Using this user-defined object also involves a number of steps:

- Registering and adding the object.
- Building logic in the parent application using the object's properties and functions.

Adding the Objects and Saving the Application as an Object

To add the objects:

1. Start a new Rapid application.
2. Add a lamp. (You can place it anywhere on the layout.)
3. Add a timer. Set its initial value to one second.

To save the application as a user-defined object:

- Choose File|Save As User Object. Save the user-defined object in the *objects* folder with a name such as *userlamp.udo*.

Be sure to save after you complete each step.

Building the Basic Logic

You want the lamp to switch off whenever the timer generates the *tick* event. The property that you will add in the next step will allow you to define a value for the timer's *initialCount* property—giving you control over how long the lamp is lit.

To turn off the lamp when the timer ticks:

1. In the Logic Editor, add an internal transition.
2. Build the following event trigger:

   Timer1 tick &
3. Build the following action:

   Lamp1 off

Adding an Integer Property

You need to add an integer property to the root object to control the timer's *initialCount* property.

To add the property:

1. Either:
   In the Object Layout, double-click anywhere in the workspace, then click the parameter pane's More button.

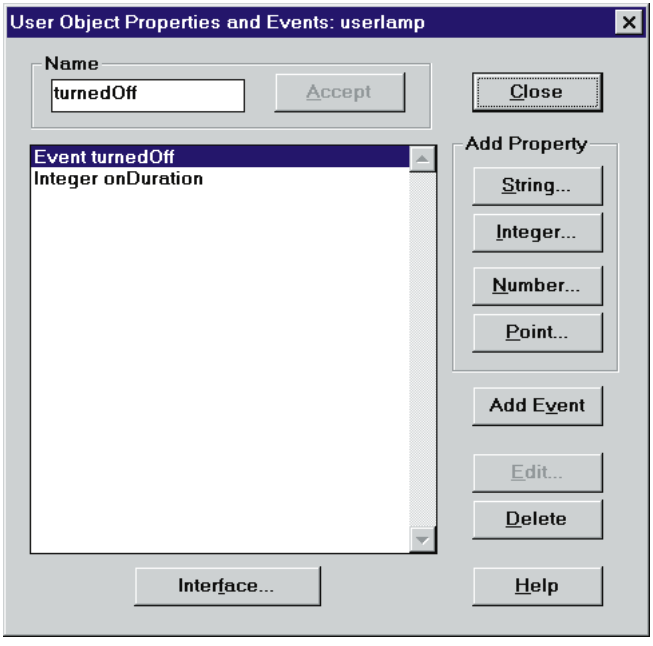
   Or:
   Choose Edit|User Object Properties.
2. In the Add Property group of buttons, click Integer.
3. Set the property's initial value to 1 and click OK.
4. Rename the property "onDuration" and click Accept (do not close the dialog box).

Adding an Event

You need to add an event to the root object that will allow you to trigger a transition when the lamp goes off.

To add the event:

1. Click the Add Event button; Rapid adds a new event, Event1, to the list.
2. Rename the event "turnedOff" and click Accept. The dialog box should look similar to:



3. Close the dialog box.

The *turnedOff* event must be triggered in the user-defined object's logic. If the event is never triggered, the parent application will never receive the event and any transition based on the event will never occur!

You want the event to be triggered when the lamp goes off, that is, when the timer ticks.

To trigger the *turnedOff* event

1. In the Logic Editor, select the *Timer1 tick* & trigger.
2. In the Actions column, build the following new action:

   userlamp.turnedOff trigger

Creating the User-Defined Functions

You now need to build three user-defined functions *(off*, *on* and *onFor:*) and export them all. The exported functions will be available in the parent application's logic as functions of the user-defined object. For details on how to build user-defined functions, refer to Chapter 18: "User-Defined Functions."

To build the *off* function:

1. Open the Function Editor and name the new function *off*.
2. Build the following activity:

   ```
   Lamp1 off
   ```

3. Choose Functions|Exported (to make the function available to the parent application).
4. Accept the function.

To build the *on* function:

1. Start a new function and name it *on*.
2. Build the following activities:

   ```
   Timer1.initialCount := userlamp.onDuration
   Timer1 restart
   Lamp1 on
   ```

3. Choose Functions|Exported.
4. Accept the function.

What's the logic?

The *on* user function you have just built:

- Assigns the value of the *onDuration* integer property to the timer's *initialCount* property.
- Restarts the timer.
- Turns the lamp on.

When the timer ticks, the lamp goes off (the user object's own internal transition).

When you use this function in the parent application, the lamp will switch on for the *onDuration* period. If you do not set a value for *onDuration*, the lamp will light for one second (*onDuration*'s initial value).

To build the *onFor:* function:

1. In the Object Layout, add an integer object and name it *OnLength*. You will use this integer as the argument in the *onFor:* function.
2. In the Function Editor, start a new function and name it *onFor* (without a colon).

3. Build the following activities:

   Timer1.initialCount := OnLength
   Timer1 restart
   Lamp1 on

4. Choose Functions|Add Argument.
5. Select OnLength from the list and click OK. The function name becomes *onFor: <Integer:OnLength>*.
6. Choose Functions|Exported.
7. Accept the function.
8. (Optional.) You can now delete the OnLength integer from the Object Layout—it was only needed to simplify adding the argument. (You can add arguments without adding objects first, see p. 18–18.)

What's the logic?

The function you have just built, *onFor:*, appears as a function of the root object. When you use this function in an activity, Rapid will prompt you to supply an integer argument. This argument determines the value of Timer1's *initialCount* property, which, in turn, determines how long the lamp is lit. This function completes all the user object's logic.

Defining the User-Defined Object's Class and Button

Before registering the user-defined object, you can define its object class and its Object Palette button. If you do not specify otherwise, Rapid will create a new object class that contains only one object — the user object itself.

You can add the object to the lamp class and assign a bitmap to its button.

To add the user-defined object to a specific object class:

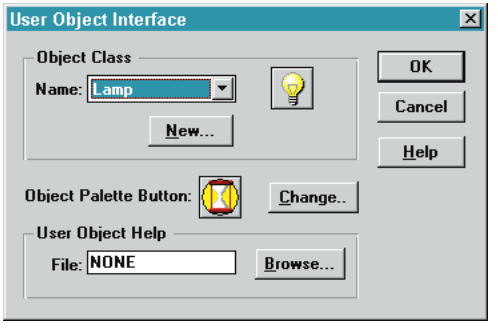
1. In the Object Layout, choose Edit|User Object Properties.
2. Click the Interface button.
3. Scroll down the Object Class list and select Lamp (do not close the dialog box).

To assign a bitmap to the user-defined object's button:

1. Click the Change button (next to the Object Palette button).
2. Select *userlamp.bmp*. You can find this bitmap in Rapid's *objects* folder.



The dialog box should look like:



3. Click OK, then close the User Object Properties and Events dialog box.
4. Save the user-defined object.

Registering and Adding the User-Defined Object

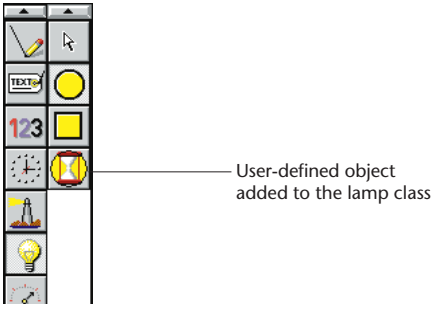
Now that you have completed building the user-defined object, you can register it and add it to an application.

To register the user-defined object:

1. Start a new application and open the Object Layout.
2. Choose Objects|Register.
3. Select USERLAMP from the Available list and click Add, then click OK.

To add the user-defined object to the layout:

1. In the Object Palette, click the lamp class button; notice that the user-defined object's button has been added.



2. Click the object's button; notice that the its name, USERLAMP, appears in the Application Manager's status bar.



3. Place the object anywhere on the layout; remember—you cannot resize it.
4. Double-click the object to open its parameter pane; notice that its default name is Lamp1—a user-defined object's default name is always the name of the object class to which it was assigned.

Using the User-Defined Object's Logic

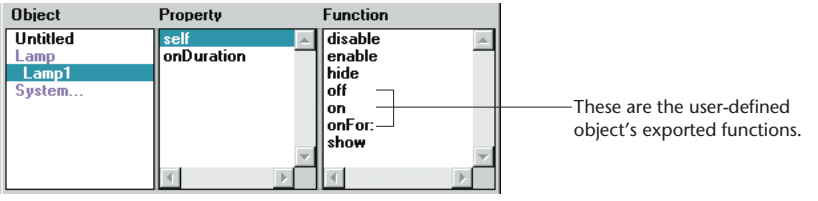
To illustrate how a user-defined object's logic fits in with the logic of standard objects, build an entry activity that uses the *onFor:* function.

To add the entry activity:

1. Open the Logic Editor.
2. Build the following entry activity:

   Lamp1 onFor: 3

   Notice that the user-defined object (Lamp1) appears in the object list just like any other object with its own properties and functions. The fact that this object is itself a mini-Rapid application is completely invisible.



3. Save the parent application with a name such as *parent.rpd*.

Testing the User-Defined Object

To test the user-defined object:

- Open and start the Prototyper; the lamp lights and goes off after three seconds.

Further practice

Build other logic that uses the *onDuration* property and the lamp's *on* and *off* functions.

Remember! The *on* and *off* functions are actually exported functions that you added to the user-defined object in the Function Editor. If you had not added and exported them, the Lamp1 user object in this application would not have *on* and *off* functions.

Part 2: Making Changes to the User-Defined Object

In this section of the tutorial, you will make the following changes to the user-defined object and then examine how the parent application is affected:

1. Add a sound object that beeps when the lamp goes off. This is an example of a change that does not affect the parent application.
2. Add an exported function that starts the sound. This is an example of a change where you need to add the new function in the parent application.

Adding the sound object

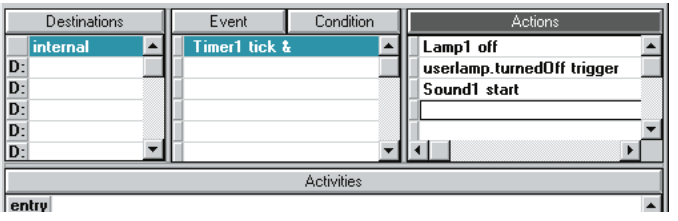
You will add a sound object to the user-defined object that beeps when the lamp goes off.

To add the sound object:

1. Open *userlamp.udo* (or whatever you called it).
2. Add a sound object to the layout.
3. Change its duration to 250 ms.
4. In the Logic Editor, add the following action to the internal event trigger:

   Sound1 start

   The final transition should look like:



6. Save the user-defined object.

To test the modified user-defined object:

1. Open the parent application.
3. Run the Prototyper. The lamp lights, goes off after three seconds and the sound beeps.

   Note that the behavior of the user-defined object has changed without needing to make any changes in the parent application.

Adding a new function

You will now add a new function to the user-defined object that will allow you to start the sound from within the parent application.

To add the new function to the user-defined object:

1. Open the user-defined object and open the Function Editor.
2. Create a new function and name it *startSound*.
3. Build the following activity:

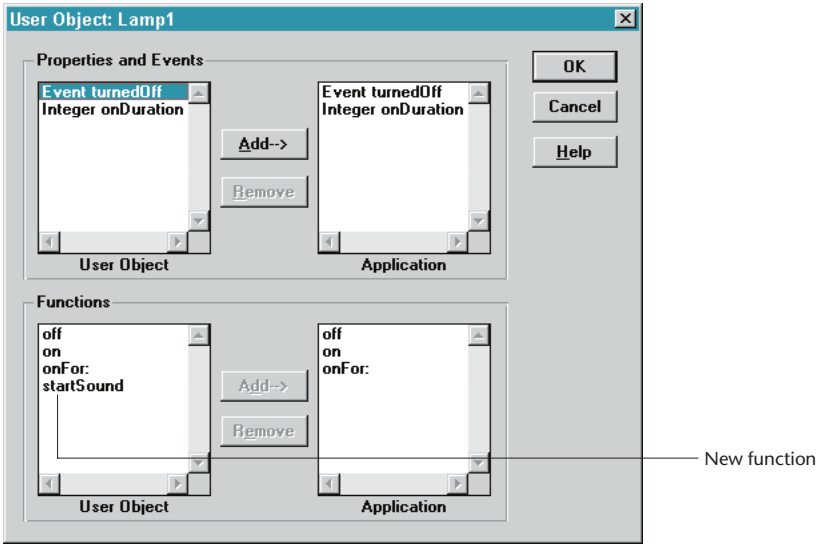
   Sound1 start

4. Choose Functions|Exported.

5. Accept the function and save the user-defined object.

To add the function to the parent application:

1. Open the parent application.
2. Open the Object Layout and double-click the user-defined object.
3. Click the More button; the following dialog box opens:



Note that the new function, *startSound*, is not automatically added to the application.

To add the new function to the application:

1. Select the *startSound* function.
2. Click the Add button; the function is added to the application.

Open the Logic Editor and look at the Logic Palette; you can see that the new function is added to the list of the user-defined object's functions.

To test the new function:

- Build logic that uses the new function, such as an internal action:

```
Lamp1 startSound
```

CHAPTER 20

The Prototyper

The Prototyper is a test bed for running your applications. You can operate user input objects (such as pushbuttons and menus) with the mouse and keyboard, allowing you to simulate the application's operation and logical behavior.

In this chapter you will read about:

- Interactively testing your application's behavior.
- Using the Trace option to follow application logic in the Mode Tree during execution, thereby simplifying debugging.
- Recording a sequence of user inputs and saving the sequence to a file. The file can be replayed in order to closely examine the way in which your application runs. It can also be used to demonstrate a particular series of actions to others.
- Demonstrating the operation of your application to others.

THE PROTOTYPER WINDOW

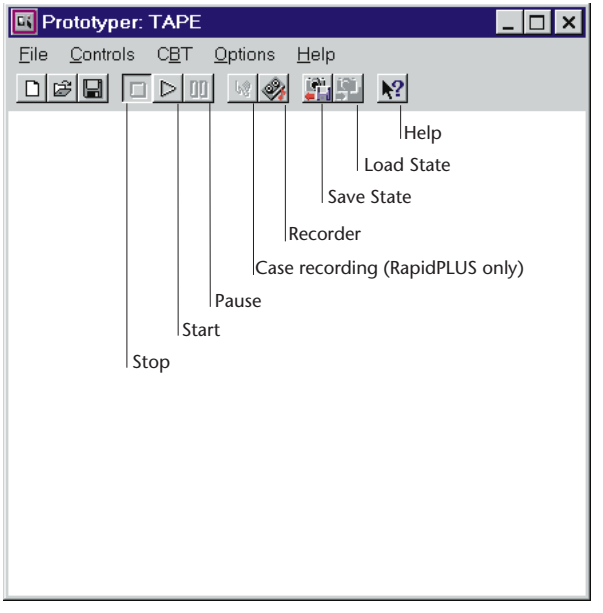


Ctrl+R

To open the Prototyper:

- Choose Tools|Prototyper from the Application Manager, or click the Prototyper button.

When you first open the Prototyper window, it is empty.



The Prototyper window

RUNNING THE APPLICATION

Starting the Application



To start the application:

- Click the Start button or choose Controls|Start.

The application begins to run: the root mode and its default descendants are activated; Rapid executes the entry activities of all the active modes and then

it shows the objects. If the trace option was enabled, the movement between modes is traced in the Mode Tree (see "Tracing the Logic" below).

Testing the Application

You test your application by operating it as your prospective users would: you manipulate pushbuttons, potentiometers, and switches on the layout. Each of these user input objects is animated as you manipulate it; pushbuttons go in and out, rotary potentiometers turn, switches flip, etc. Other objects also react according to application logic: lamps turn on and off, dial needles move and displays update, etc.

To manipulate a user input object:

- Place the cursor on an object's "hot spot" and click.

❖ *NOTE: For a summary of the various object hot spots, see pp. 21-8 to 21-10 in Chapter 21: "The Reviewer".*

Using other Rapid tools while running the application

You can work with other Rapid tools after you start the Prototyper, but there are certain operations that leave the Prototyper in an undefined state. Listed below are the operations that cannot be performed while the Prototyper is running and how Rapid responds when you do perform them:

TOOL	OPERATION	RAPID'S RESPONSE
Object Layout	Adding or deleting an object. Changing properties of an active object in its dialog box. Replacing an active object. Editing, copying or replacing a nongraphic object.	A message box opens informing you that the Prototyper is running and asks if you want to stop it.
Mode Tree	Deleting, moving, or adding a mode.	These commands are disabled.
Logic Editor	Adding, editing or deleting logic.	The Prototyper is automatically stopped.

Updating the Palette

(This section applies if your monitor is in 256-color mode.)

If you run an application that contains 256-color bitmaps, the application may flicker when switching from the Prototyper to a different Windows application and vice versa. The flickering occurs because Windows redraws the Rapid application once or even twice.

❖ *NOTE: This problem occurs between any Windows programs that contain different palettes and is not particular to Rapid.*

To stop the palette flickering:

- Deselect Options|Palette Update.

When deselected, Windows will no longer redraw your application each time you switch to and from the Prototyper. Any 256-color bitmaps, however, might appear with an incorrect palette.

Tracing the Logic

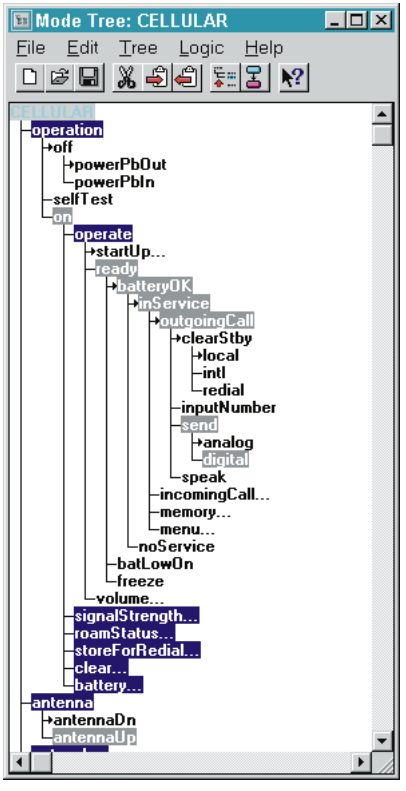
You can follow which of the modes are currently active as the application runs if the Trace option is enabled and the Mode Tree window is open alongside the Prototyper. The modes that are currently active in the Prototyper are highlighted (on a gray background) in the Mode Tree.

In this way you can "trace" the active modes—which is extremely useful when debugging applications.

❖ *NOTE: The application operation slows down when the Trace option is active.*

To enable or disable the Trace option:

- Choose Options|Trace. The Trace option is checked when Trace is enabled.



Active modes are highlighted in gray.

Pausing/Resuming the Application

You may want to pause the application in order to examine and modify the defined destinations, triggers, actions and activities, without having to stop and restart the execution.

To pause the application:



- Choose Controls|Pause, or click the Pause button.

The application halts but the panel is still visible. If Trace is active, the Mode Tree window continues to display the application's currently active mode(s). The Pause option in the menu changes to Resume.

Once paused, you can either stop or resume the application; the Start option is not available. (If you want to start the application from its initial state, you must first stop the application.)

Editing functions that automatically pause the Prototyper

Most editing functions in the Object Layout will automatically pause the Prototyper. These include:

- Changing the name of a graphic object, or any other parameter in the parameter pane.
- Moving or sizing an object with the mouse.

Any changes you make while the Prototyper is paused will come into effect as soon as you resume the application.

Stopping the Application

You may want to stop the application to make major revisions in the application's layout or logic.

To stop the Prototyper:



- Choose Controls|Stop, or click the Stop button. The application disappears from the Prototyper window. If "Trace" was enabled, the highlight disappears from the Mode Tree window.

To restart the application:

- Choose Controls|Start, or click the Start button.

❖ *NOTE: When you stop an application, Rapid executes the exit activities of all the active modes.*

RECORDING IN THE PROTOTYPER

The Prototyper's Recorder utility allows you to record a series of user actions and to play back the resulting sequence. The playback may be at either normal or fast speeds, or "step-by-step" (i.e., one action at a time). This utility allows you to:

- Examine in close detail the way your application runs in order to locate bugs.
- Record a particular procedure as a file and play it back at a future date or demonstrate a recorded series of actions to others on a different computer. A Rapid record file is a readable text file which you can edit through a text editor.

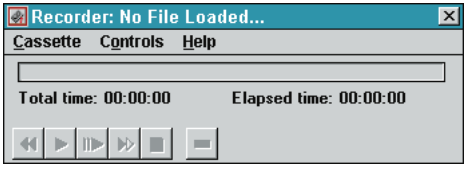
Recording a Sequence

To record a sequence:

1. Start the Prototyper.



2. Choose Controls|Recorder, or click the Recorder button; the Recorder panel opens:



3. Move the Recorder panel so that it does not obscure the application in the Prototyper.



4. Click the Record button; the Elapsed time starts counting.
5. Operate the application in the Prototyper—click pushbuttons, connect switches, etc.



6. Click the Recorder's Stop button.
7. You can continue recording by clicking the Record button again.

If you want to record a further sequence without losing the current recording, you must first save it (see "Saving the Recorded Sequence," below).

Playing the Recorded Sequence

You can immediately play back the sequence that you have just recorded.

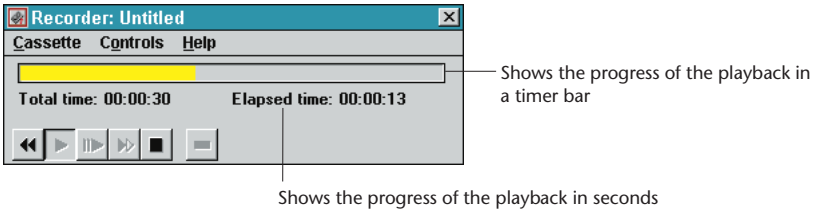
To play back the recorded sequence:



1. Choose Controls|Rewind or click the Recorder's Rewind button.
2. Initialize the application in the Prototyper, i.e., click the Stop and Start buttons.



3. Choose Controls|Play or click the Recorder's Play button; the recorded sequence is played back in the Prototyper and followed in the Recorder.



If you click the Rewind button during the playback sequence, the playback stops and the recording rewinds back to the start.

Playing the recording at different speeds

You can play your recording at speeds other than the normal Play speed:



- "Step" plays the recording one action at a time. Clicking "Step" once (or choosing Controls|Step) plays the first action in the recording, clicking a second time plays the next action, and so on. You can continue playing at normal Play speed by clicking "Play." When playing at "step" speed, timers are ignored.



- "Cue" plays the recording at a fast speed. That is, Rapid ignores the time breaks between actions, but retains the standard speed of moving objects (potentiometers, etc.). When playing at "cue" speed, timers might not behave as they should.

Managing Recorder Files

Saving the Recorded Sequence

You can save the recording as an *.rcd* file and play it back at a later time.

To save the recording:

- Choose Cassette|Save As; a dialog box opens for you to enter a file name and location (*\recorder* is the default folder). Rapid automatically adds the *.rcd* extension.

Recording a New Sequence

To record a new sequence:

- Choose Cassette|New. If you have not saved the current recording, Rapid prompts you to save.

Deleting a Saved Recording

To delete an *.rcd* file:

- Choose Cassette|Delete; a dialog box opens with a listing of the *.rcd* files in the current directory.

Playing a Saved Recording

To play back a saved sequence:

- Choose Cassette|Load; a dialog box opens with a listing of the *.rcd* files in the current directory.

  ❖ *NOTE: Rapid's recorder will not play files from version 2.2 applications.*

Possible errors

When loading and/or replaying the file, Rapid checks for one of the following conditions:

- All objects referenced by the recording still exist in the application.
- The events are still appropriate for the named objects. For example, SW1 referenced by the recording was originally a 6-position rotary switch and one of the events involves Position6. In the meantime, SW1 has been changed to a 4-position switch. The recorded event, therefore, is no longer legal.

If either of the conditions is not fulfilled, Rapid terminates the load and/or playback operation.

The importance of initializing the application

For the playback to be effective, the application must be in the same state as when you began the recording. Otherwise, situations such as the following will arise.

Example:

A rotary switch was in Position1 when the recording began and the first recorded event was to click Position6. If the rotary switch is already in Position6 when you initiate the playback, the first event is lost.

You should initialize the application (i.e., click the Stop and Start buttons) before beginning to record and then re-initialize the application before starting the playback.

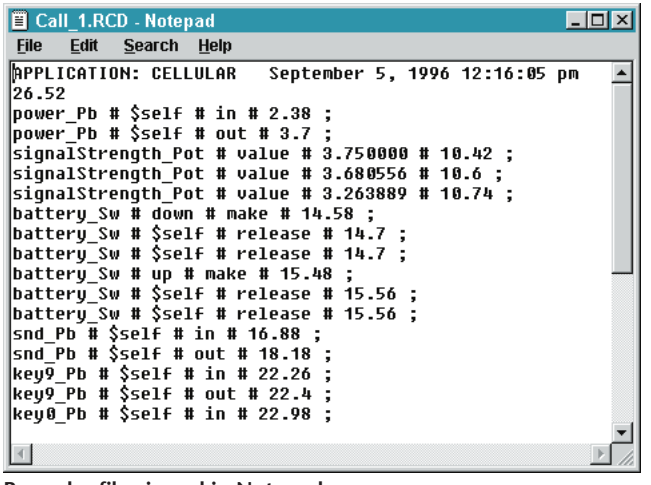
Editing the Recording File

Rapid recording files are readable and you can edit them using any text editor. In this way you can introduce changes into the recording without having to re-record the whole sequence.

To view the Recorder file:

1. Open your editor, e.g., Notepad.
2. Open the relevant *.rcd* file.

A typical recording file viewed through Notepad is shown below.



Recorder file viewed in Notepad

The *.rcd* file format

The first section of the file, the header, consists of two lines. An example of a header is:

```
APPLICATION: CELLULAR   September 5, 1996  12:16:05 pm
26.52
```

The first line includes the application name and the date and time of the recording. The second line contains the total recording time in seconds.

Every line in the body of the file contains one recorded user action. Each line contains four fields that are separated by a hash sign (#) and ends with a semicolon. An example of a line from a recorder file is:

```
On_Off_Switch # up # make # 1.263 ; turns tape recorder on
```

The fields that make up each line are (in order):

1. Event object: the object's name as it appears in the application, e.g., "On_Off_Switch."
2. Event property: the name of the object's property that was affected by the user action, e.g., "up."
3. Event symbol: the name of the action itself (usually an event), e.g., "make."
4. Time stamp: the time elapsed (in seconds) since the start of the recording when the operation took place, e.g., 1.263 seconds.
5. Optional comment: following the semicolon, you can add a comment (which Rapid ignores).

❖ *CAUTION! You should be very careful when manually editing a recording file. Disrupting the file's format will prevent Rapid from understanding the file and might cause unpredictable results.*

Editing tips

- Ensure that the value of the time field increases as you move down the file.
- Do not delete the semicolon at the end of a line.
- Rapid ignores white space before and after the fields' text so you can add tabs or spaces to improve readability.
- Object and property names should be identical to the ones that appear in the Logic Editor, but *self* should be "$self."
- Rapid only records switches' *make* events, not their *break* events.
- You can edit the file with a word processor, but the file must be saved in "Text Only" format, and with the extension *.rcd*.

Recording "Special Cases"

Switch connections

- When Rapid records the movement of a switch from one position to another, it records only the *make* event, not the *break* event.

  For example, when a switch is moved from position1 to position2, Rapid generates two events: *Switch.position1 break &* and *Switch.position2 make &*. When Rapid records this action, though, it records just *Switch.position2 make &*:

  ```
  Switch # position2 # make #
  ```

- When Rapid records the movement of a switch from position 1 to position3, it does not record the intermediate *Switch.position2 make &* event.

  ❖ *NOTE: These exceptions do not change a recording file's effect on an application, you still get an accurate playback.*

Auto repeat pushbuttons and stepper switches

A press on a pushbutton is recorded as two separate events: *in* and *out* (*inUp*/*inDown* and *release* for a stepper switch). If the button is auto repeat, a long press is recorded as two events with a delay between them. Because Rapid's state machine may work at different speeds (depending on computer load, amount of available memory, etc.), the button or stepper switch could send a different number of events each time the recording is played.

Potentiometer changes

Rapid records potentiometer inputs as changes in value, for example:

```
Potentiometer2 # value # 31.565050 # 22.026 ;
```

In this example, the event symbol gives the potentiometer's value at the time stamp. The next recorded input might read:

```
Potentiometer2 # value # 32.522865 # 22.191 ;
```

Drag 'n drop movements

You can record the movements of a drag 'n drop object as you drag it in the Prototyper, just as you can record changes in any other object. When you play back a recording of a drag 'n drop object being moved, you might see that the object's movement is not smooth.

This occurs because Rapid only records stages of the object's movement, not the entire movement. (Recording the entire drag movement would generate a file that contains too many object changes to handle.) Other than the unsmooth movement that might appear, the recording is accurate.

User-defined objects' events

When recording a user-defined object's events in the Recorder, the object's unique name is given as:

```
<User-defined object name>:<Object name>
```

For example, if a user-defined object called "System" had a pushbutton called "Pushbutton1," the pushbutton being pushed in would be recorded as:

```
System:Pushbutton1 # $self # in
```

The syntax for nested user-defined objects is:

```
<UDO1>:<UDO2>:...<UDOn>:<Object name>
```

For example, if a user-defined object called "Simulation" contained another user-defined object called "System," which, in turn, contained a pushbutton called "Pushbutton1," the pushbutton being pushed in would be recorded as:

```
Simulation:System:Pushbutton1 # $self # in
```

Generated objects' events

When recording a generated object's events in the Recorder, the object's unique name is given as:

```
x$<Name of copied object>
```

where x is an integer that indicates which of the copies of a particular object generated the event.

For example, if a pushbutton called "Pushbutton1" was copied, and the generated pushbutton was pushed in, the event would be recorded as:

```
1$Pushbutton1 # $self # in
```

If a second copy of the pushbutton was generated and pushed, its event would be recorded as:

```
2$Pushbutton1 # $self # in
```

Events generated by an external program

The same syntax for identifying the unique name of an object within a user-defined object is used when sending an external event to Rapid from an authoring program (via the CBT channel).

EXTERNALLY CONTROLLING THE PROTOTYPER

If you are building an application that will be controlled by an external program, such as an authoring tool, you will need to check your progress from time to time to see how your application behaves. To do so, you need to carry out two steps:

1. Define the communication parameters of the CBT channel that will link the Prototyper to an external managing program. (The CBT channel is a specific kind of DDE channel used for sending commands to Rapid.)
2. Switch your Prototyper to student mode, allowing an external program to control it.

When the Prototyper runs in student mode, it behaves exactly like a standalone Reviewer module. The external managing program you use may be an authoring program or another Rapid application that uses the simulation manager object.

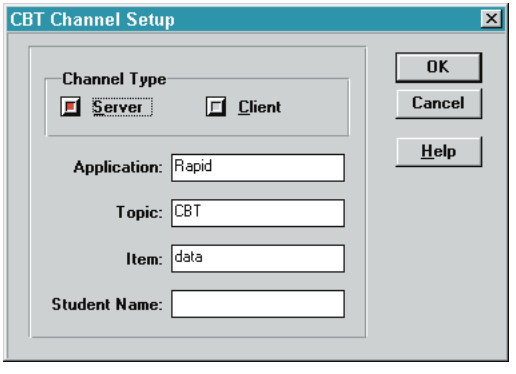
(For details about applications that communicate with other programs, see Chapter 7: “Communicating with Other Programs.”)

Setting up the CBT Channel

Before you switch your Prototyper to student mode, you must define the parameters of the DDE communication channel that Rapid opens between the Prototyper and the external program.

To set up the CBT channel:

- Choose CBT|Setup:



In this dialog box, you can define the Prototyper as either the DDE server or client. If you define the Prototyper as a server, then the external program must be a client and vice versa. Whether to set up the Prototyper as a server or as a client depends on the external program. The external program may only be able to function as a server, in which case your Prototyper should be a client, or vice versa.

The information in the Application, Topic and Item fields must be identical for both the server and client applications.

To identify the student:

- In the Student Name field, enter the name of the student who will be using the Prototyper in its student mode (e.g., your name).

  This name identifies the Prototyper to the external program. You can leave the student name empty.

Rapid in Student Mode

When the Prototyper is in student mode, Rapid opens a DDE channel with the parameters that you defined above. In student mode, the external program at the other end of the channel can control the Prototyper.

To switch to student mode:

- Choose CBT|Student Mode; the option appears with a check mark and Rapid opens the CBT channel.

  The Prototyper will now behave exactly like a student Reviewer.

  You can control the application from your external program: minimize the window, start and stop the Prototyper, send events, etc. If you discover any problems in the way your authoring application interacts with the Rapid simulation, make the necessary adjustments and test again. If necessary, switch Rapid back from student mode to make any changes in the application.

CONTROLLING THE PROTOTYPER'S STATE

Rapid allows you to save the current state of the Prototyper at any point while the Prototyper is running. Rapid saves this "snapshot" of the Prototyper as a readable text file which you can edit with a text editor.

The CBT menu has options that allow you to save the current state of the Prototyper to a file, load any state file into the Prototyper or load the application's default state. You can also carry out the first two of these options by clicking buttons in the toolbar.

This feature allows you to:

- Save the state of an application before taking a break, and continue the session at a later stage.
- Prepare a specific state of the application as a starting point for a student's practice session.
- Exchange application states between teacher and student.

Saving the Prototyper's State

To save the Prototyper state:



1. Save the application.
2. Pause the Prototyper (recommended but not essential).
3. Choose CBT|Save State or click the Save State button; a Save As dialog box opens.
4. Enter a name for the file and select the folder where Rapid is to save the file; Rapid automatically assigns the extension *.rst* to your name.
5. Click OK; Rapid saves the state to the specified file. If you paused the Prototyper, it remains paused. If Rapid paused the Prototyper, it resumes.

For details on the contents of the state file and how to edit it, see "The State File Format" below.

❖ *NOTE: If you save a state and restart the Prototyper, the application starts at the saved state, not at its default state.*

The simulation manager object's *saveState:* function produces a state file of the student simulation.

Loading a State File

Once you have saved a state file, you can load it into your Prototyper—or into any other Prototyper or Reviewer that is running the same application (e.g., a student's Reviewer).

You can load a state file into the Prototyper at any point:

- If the Prototyper is running, Rapid loads the file, and resumes the Prototyper from the new state.
- If the Prototyper is paused, Rapid loads the new state and the Prototyper remains paused in the new state.
- If the Prototyper is stopped, Rapid loads the new state and uses the new state as the starting point when you start the Prototyper.



To load a state file:

1. Choose CBT|Load State or click the Load State button.
2. Select a state (*.rst*) file and click OK; Rapid pauses the Prototyper, loads the state file and resumes the Prototyper from the new state.

Note that the new state becomes the starting point each time you start the Prototyper.

To start the Prototyper from its default state
(as originally set out in the Object Layout):

- Choose CBT|Default State.

The simulation manager object's *loadState:* function loads a state file into the student's Prototyper or Reviewer.

The State File Format

Rapid state files are readable and you can edit them using any text editor.

❖ *CAUTION! Take great care when editing a state file! Any wrong data could lead to Rapid behaving unexpectedly.*

The state file contains three sections: header, modes and objects.

Header

The header contains the application's name, and the date and time of the application's last update. An example of a header:

```
Rapid application saved state - all objects
APPLICATION NAME:TAPE
UPDATE:June 11, 1996 8:00:39 pm
: START APPLICATION SEGMENT - TAPE :
```

Mode Section

An example of part of a mode section is:

```
: START MODES :

[TAPE]; Root Mode
{
     state = active
                  activeChild = On
}

[Off]; Exclusive
{
     state = inactive
                  activeChild = $none
}

[On]; Exclusive
{
     state = active
                  activeChild = $all
}
```

This section contains a list of all the application's modes. For each mode, it gives the mode's type, its current state and its last active child mode.

The mode's status is either active or inactive. Its last active child is one of:

- A mode name (if a child mode other than the default mode was active).
- $all (if the child modes are concurrent).
- $default (if the default child mode was active).
- $none (if there are no child modes).

The mode section ends with the line:

```
: END MODES :
```

Object Section

An example of part of an object section is:

```
: START OBJECTS :

[TAPE]
{
     hidden = NO
      disabled = NO
     dragTo = 0@0
}

>[On_Off_Switch]; Rocker Switch
{
      up = active
     down = inactive
     hidden = NO
                  disabled = NO
}

>[On_Off_Lamp]; Round Lamp
{
          blinkPeriod = 500
         hidden = NO
      disabled = NO
     dragTo = 83@147
      blinkStatus = OFF
     state = ON
}

>[Tape_Count]; Stopwatch
{
  time = 413
}

[Beep]; Sound
{
     frequency = 5000
                  duration = 500
      modulationPeriod = 500
         dutyCycle = 50
}
```

This section lists the details of all the application's objects. These details include the object's unique name, the type of object and the object's current status. A > symbol that precedes an object indicates that its current state is different from its default state.

The mode section ends with the line:

```
: END OBJECTS :
```

Editing a state file

If you do decide to edit a state file by hand, bear the following points in mind:

- You should only edit the values of each property, not the property itself.
- Each object's state description must be enclosed by curly brackets.
- White space before and after the fields' text is ignored, so you can add tabs or spaces to improve readability.
- Rapid ignores any text that follows a semicolon (;) until the end of the line. You can add your own comments by typing a semicolon after the line's last character and then typing text.
- You can edit the file with a word processor, but you must save the file in "Text Only" format and with the extension *.rst*.

Compressed State File

When you save a state, Rapid saves all the application's modes and objects. When Rapid recovers a state, it does not need the complete description of the state.

You can write your own state file that contains only the specific objects which have changed from the default state. When Rapid loads such a state file, only those objects are affected. This state file is known as a compressed state file.

❖ *NOTE: A compressed state file must include* ***all*** *the application's modes.*

The simulation manager object's *saveCompressedState:* function produces a compressed state file of the student simulation.

CHAPTER 21

The Reviewer

The Reviewer is a stand-alone Prototyper that allows you to run Rapid applications independent of the Rapid development environment. Because the Reviewer does not include the development tools (such as the Object Layout and the Logic Editor), it loads more quickly and requires much less RAM (a minimum of 8 MB instead of 16 MB). It also safeguards your distributed application from unwanted modifications.

You can use the Packager utility (see Chapter 22, “The Packager”) to prepare self-installing versions of your Rapid application for distribution. The package includes the Reviewer files, the Rapid application file and any external files which may be required to run the application (such as multimedia files, specialized fonts and data files).

As in the Prototyper, in the Reviewer you can:

- record and/or play back a sequence of user actions.
- save and load application states, i.e., application “snapshots” saved to a text file.
- run the Reviewer in various modes so that the Rapid application can be controlled by an external program (such as a CBT authoring tool) or by a Rapid application using the Simulation Manager object.

In this chapter you will read about:

- Opening and closing the Reviewer
- Working with applications
- Manipulating application objects
- Running the Reviewer in various modes.

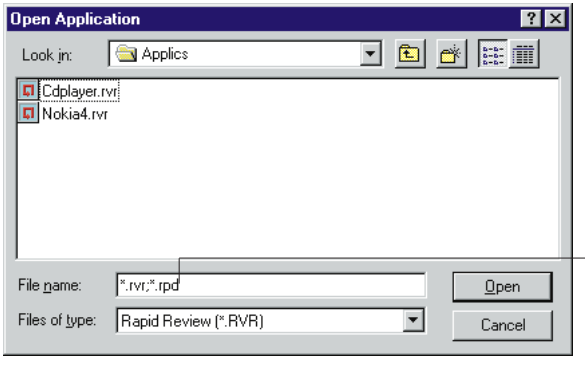
OPENING AND CLOSING THE REVIEWER

When you use the default method for opening the Reviewer, you select an application during the load procedure. However, you can configure the Reviewer so that it automatically runs a specific application each time you open it.

To open the Reviewer (default method):



1. Click the Reviewer shortcut in the Rapid folder of the Start menu.
2. The Reviewer begins loading and the Open Application dialog box appears:



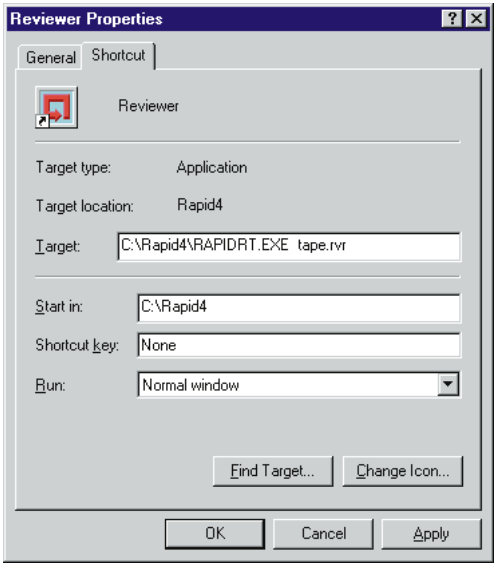
3. Select a file type in the Files of type list box. Two types of files can be opened in the Reviewer:
   - Regular Rapid application files (*.rpd*)
   - Runtime versions of Rapid applications (*.rvr*)

   The difference between the two is that the *.rvr* files are somewhat compressed and are "read-only" files which cannot be opened in the Rapid development package.
4. Select a file, then click Open.
5. The Reviewer completes loading and the application appears in the Reviewer window.

❖ *NOTE: If you do not choose an application file, the Reviewer will not load.*

To configure the Reviewer so that it opens with a specific application:

1. Right-click the Reviewer shortcut and choose Properties from the popup menu. Click the Shortcut tab.
2. In the Target text box, add the application's path and file name immediately after RAPIDRT.EXE [space]. For example:



3. Click OK.



4. Double-click the Reviewer shortcut (shown to the left as a Desktop shortcut); the Reviewer loads with the specified application.

To close the Reviewer:



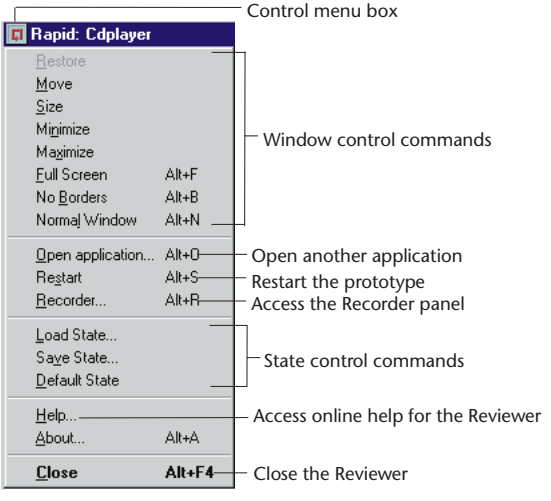
- Choose Close from the Control menu or double-click the Control menu box (see the following figure).

WORKING WITH THE APPLICATION

When an application is first opened, the Reviewer presents it in normal window view. If the application is larger than the window, scroll bars appear. You can resize the window using the usual Windows procedures, or by using the Resize command in the Control menu (see "Sizing the Reviewer window", p. 21-8).

To open the Control menu:

- Click the Control menu box in the upper-left corner of the Reviewer window. The following menu opens:



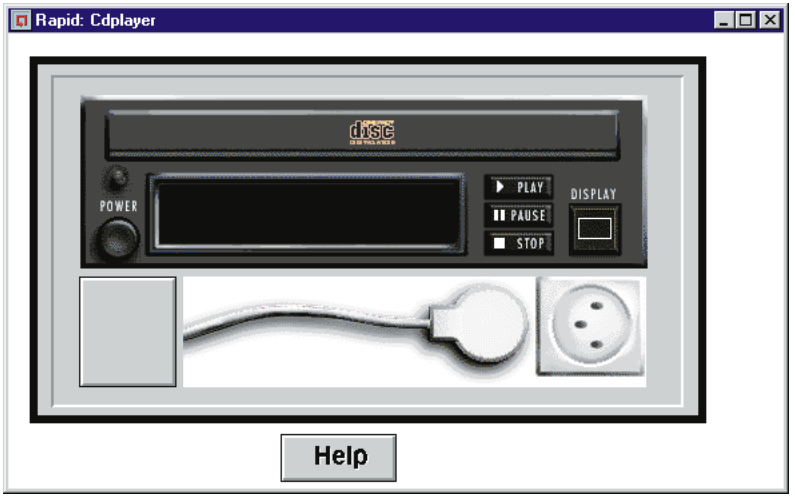
❖ *NOTE: Do not double-click the Control menu box or you will close the Reviewer.*

The Reviewer Window

You can work with the Reviewer in three different views: normal window, no borders and full screen.

Normal Window View

When the Reviewer opens, it is always in normal window view. The illustration below shows an application in normal window view.



An application in normal window view

To return to normal window view (from another view):

- Choose Control|Normal Window.

Alt+N

No Borders View

In this view, the window remains the same size as the normal window view, but the title bar is removed.

To display the application in no borders view:

- Choose Control|No Borders.

Alt+B



An application in no borders view

To access the Control menu in no borders view:



- With the focus on the Reviewer window, press Alt + Spacebar.

❖ *NOTE: To close the menu, press Esc or click anywhere outside the menu.*

To exit no borders view:

- Choose another view in the Control menu.

Full Screen View

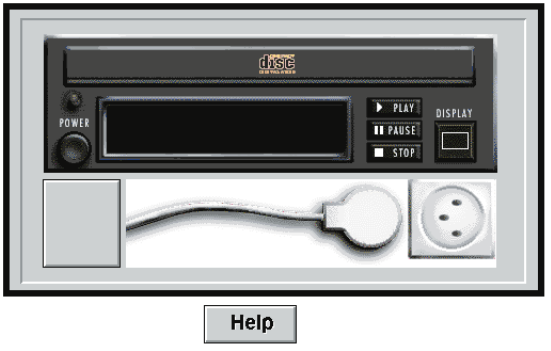
In full screen view, the Reviewer window fills the entire screen, without the title bar.

❖ *NOTE: To get a full screen view* **with** *a title bar, choose Control|Maximize.*

To display the application in full screen view:



- Choose Control|Full Screen.



An application in full screen view

To exit full screen view:

- Press Alt + Spacebar. The Control menu reopens for choosing another view.

Moving the Reviewer window

You can not use the usual Windows procedures to move the Reviewer window when it is in no border view since the title bar is not displayed. As an alternative, in both normal and no border views, you can use the Move command in the Control menu.

1. Choose Control|Move.
2. Press the spacebar.
3. Drag the window to the desired location and click.

Sizing the Reviewer window

In normal view, you can either use the usual Windows procedures to resize the Reviewer window or you can use the Size command in the Control menu.

1. Choose Control|Size.
2. Press the spacebar.
3. Move the mouse horizontally or vertically to change the window's width or height, respectively. Click the mouse to change the size.
4. Repeat steps 1-3 until you have achieved the desired window size.

Running the Application

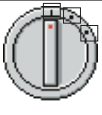
As in the Prototyper, you run the application by manipulating the objects on the screen.

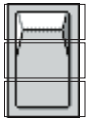
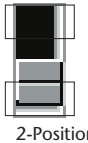
❖ *NOTE: In order to activate two buttons or switches concurrently, shift+click one then click the other.*

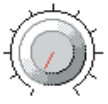
To manipulate an object:

Place the cursor on an object's "hot spot" (see illustrations below) and click as directed. Where no specific hot spot is indicated (as for a pushbutton), the boundary around the object delineates the sensitive area.

Since the application designer can change the size and location of hot spots, the hot spots shown below illustrate the default positions and sizes only.

HOT SPOTS	OBJECT TYPE	USAGE
	**Pushbutton**	**Toggle and Momentary:** Click and release at the center of the pushbutton (or anywhere within the pushbutton boundary). **Momentary with Auto Repeat:** Click and hold at the center of the pushbutton (or anywhere within the pushbutton boundary).
	**Rotary Switch**	Click and release on the dot at the desired switch position.

HOT SPOTS	OBJECT TYPE	USAGE
 2-position  3-Position	**Switch (toggle or rocker)**	**Constant Position:** Click and release the top or bottom (or center) of the switch. The switch stays in the clicked position. **Momentary Position:** Click and release the top or bottom of the switch. The switch leaves the clicked position when you release the mouse button. *NOTE: In 3-position switches, the center position is always constant.*
	**Stepper Switch**	Click and release the UP arrow (or anywhere on the top part of the switch) to increment; click and release the DOWN arrow (or anywhere on the lower part of the switch) to decrement. If Auto Repeat is on, click and hold for repeated increment/decrement action.
 2-Position	**Slider Switch**	**2-position switch:** Click the top or bottom of the switch (or on the left/right side if the switch is horizontal). **3-position switch:** As above, with an additional hot spot in the center of the switch.
 	**Cross Switch or Joystick**	Click and release one of the outer buttons or on the center button.

HOT SPOTS	OBJECT TYPE	USAGE
	**Menu Object**	To select an item: a) Click a menu item; b) Type a mnemonic (one of the underlined letters); or c) Use the up and down arrows on the keyboard to scroll to the desired item, then press Enter.
 	**Potentiometer**	**Rotary:** Click and drag the pointer. However, if you click and drag anywhere within the object's boundary, the indicator jumps to the clicked position. **Linear:** Click and drag the slider bar. However, if you click and drag anywhere within the object's boundary, the slider knob jumps to the clicked position.
	**Valve**	Click and release on one of the black connecting positions.

Palette Colors

If your application contains 256-color bitmaps, you may experience color palette problems when switching from a different Windows application back to the Reviewer and vice versa. Each time you switch from the Reviewer to the other application and back again, Windows might redraw the Reviewer application once or even twice, causing your application to flicker.

❖ *NOTE: This problem occurs between any Windows programs that contain different palettes and is not particular to the Reviewer.*

Restarting an Application

You can restart an application from its initial state. The Restart command is particularly useful if you are recording your steps.

To restart an application:



- Choose Control|Restart.

Opening a Different Application

To open a different application:

1. Choose Control|Open Application; the Open Application dialog box opens.
2. Select an application, then click Open; the application appears in the Reviewer window.

RECORDING

As in the Prototyper, the Reviewer's Recorder utility allows you to record a series of user actions and to play back the resulting sequence. The playback may be at either normal or fast speeds, or "step-by-step," i.e., one action at a time. This utility allows you to:

- Examine in close detail the way your application runs.
- Record a particular procedure as a file and play it back at a future date or demonstrate a recorded series of actions to others on a different computer.

Choose Control|Recorder to open the Recorder panel. See p. 20-6, "Recording in the Prototyper" for detailed explanations on recording and playing sequences, as well as managing recorded sequence files.

CONTROLLING AN APPLICATION'S STATE

As in the Prototyper, the Reviewer allows you to save the current state of the application at any point while it is running. This "snapshot" of the application is saved as a readable text file which you can edit with a text editor.

This feature allows you to:

- Save the state of an application before taking a break, and continue the session at a later stage.
- Prepare a specific state of the application as a starting point for a student's practice session.
- Exchange application states between teacher and student.

The Reviewer Control menu has options that allow you to save the current state of the application to a file, load a state file into the application or load the application's default state. See p. 20-17, "Controlling the Prototyper's State", for a full discussion of saving and loading states, and managing state files.

USING THE REVIEWER'S COMMAND LINE OPTIONS

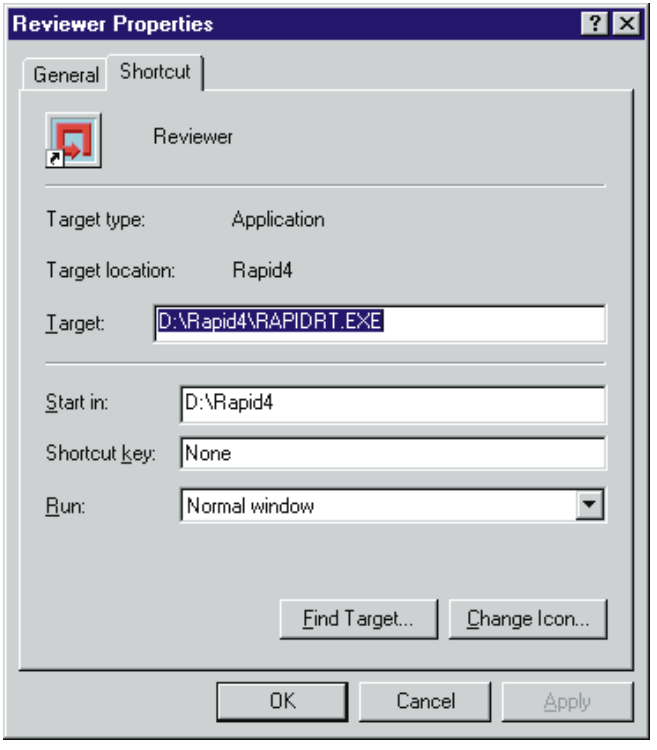
The Rapid Reviewer has several command line options that allow you to control how it behaves when it opens. These options allow you, for example, to:

- Run the Reviewer in student mode.
- Run the Reviewer as a DDE client or server.
- Set application, topic and item names.
- Load a specific application.
- Open the Reviewer in the background (hidden).

Adding Options to the Command Line

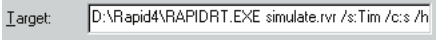
To add options to the Reviewer's command line:

1. Open the Reviewer's shortcut Properties dialog box at the Shortcut tab.



2. In the Target text box, add the option(s) at the end of the line. Leave a space between *rapidrt.exe* and the options, and an additional space between each option. (The options are listed on the following page.)

   For example:



This command line runs the Reviewer:

- Loading the s*imulate.rvr* application.
- In student mode, with the student name "Tim".
- As a DDE server.
- In hidden mode.

You might want to create several Reviewer shortcuts, each one of which would run the Reviewer in a different mode or under a different student name, load a different application, etc.

The Command Line Options

Loads a specific application when you run the Reviewer:

`<Rapid application name>`

where "Rapid application name" is the name of the application that the Reviewer loads.

❖ *NOTE: If the application is not located in the Rapid directory/folder (where rapidrt.exe is located), enter its entire pathname.*

Sets the Reviewer to student mode:

`/s:name`

where "name" is a unique name that identifies the Reviewer (the student name).

Opens the Reviewer as a server or client:

`/c:s` for DDE server

`/c:c` for DDE client

Sets Application/Topic/Item names:

`/a:application name`

`/t:topic name`

`/i:item name`

If any of the above flags are not used, the Reviewer opens with the following default values:

- Application: Rapid
- Topic: CBT
- Item: data

Enables the external events mode:

`/e`

Sets the routing modes (for the CBT channel):

`/r:1` (normal mode—default)

`/r:2` (monitor mode)

`/r:3` (rerouting mode)

Opens the Reviewer in the background (hidden):

`/h`

CHAPTER 22

The Packager

The Packager is an application distribution utility that creates a self-installing disk set for either an application alone, or for an application together with the Rapid stand-alone Reviewer.

In this chapter you will read about:

- Installing and launching the Packager.
- Creating a package.
- Editing a package.
- Installing a package

OVERVIEW

Using dialog boxes, the Packager guides you, step by step, through the procedure of choosing the application itself, as well as its associated external files (such as multimedia files, special fonts and readme files).

It then builds a compressed installation version of the application, either with or without the Reviewer — organizing the files in a specified location into folders labelled Disk1, Disk2, etc.

Disk1 always contains the *setup.exe* and *setup.ini* files, as well as the Readme file (if there was one). If there are special fonts to be packaged, they are always written to the last diskette.

The Packager also creates an <*application name*>*.pak* file which it writes to the application directory. You can double-click this file to edit the package.

The person receiving the diskettes just double-clicks on the setup file to automatically install the application (and the Reviewer, if applicable). During the packaging process, you specified a location for the installed files, but the end-user can override this suggestion.

The end-user can also choose whether to install the fonts (if any) as part of the setup, or to install them later from the last distribution diskette.

When the installation is complete, icons for launching the application and the Readme file (optional) appear in a program folder that you specified during the packaging process There's also an Uninstall icon for removing the installed files.

GETTING STARTED

Installation

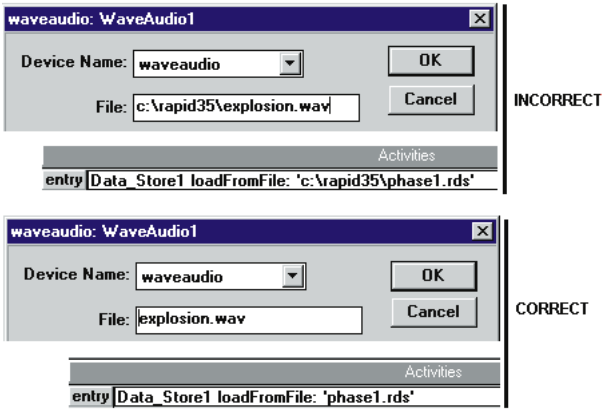
To install the Packager from the Rapid CD-ROM:

1. Double-click *setup.exe* in the root directory of the Rapid CD-ROM.
2. Check the Packager and Reviewer option in the list of Rapid modules to be installed.
3. The Packager files are installed in the \PACKAGER subfolder in the Rapid working directory and its icon is added to the Rapid program folder.

Preparing the Application for Packaging

There are two important issues to keep in mind when preparing an application for packaging:

- You must have a complete list of external files used by the application, including specialized RPXs, specialized DLLs, UDOs, and multimedia files. See p. 22-6, “Adding external files.”
- When the end-user “unwraps”’ the Rapid package, all files will be copied into the same directory. Thus, you must ensure that your application objects and logic do **not** use paths to call external files. For example:

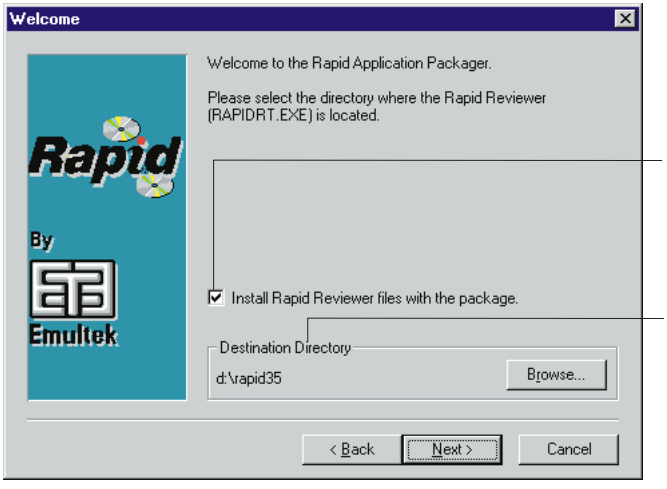


Launching the Packager



To launch the Packager:

- Click the icon in the Rapid program folder, or a shortcut, or double-click *Pack.exe*. The following dialog box appears:



If necessary, browse to the directory where the Reviewer execution file (RAPIDRT.EXE) is located.

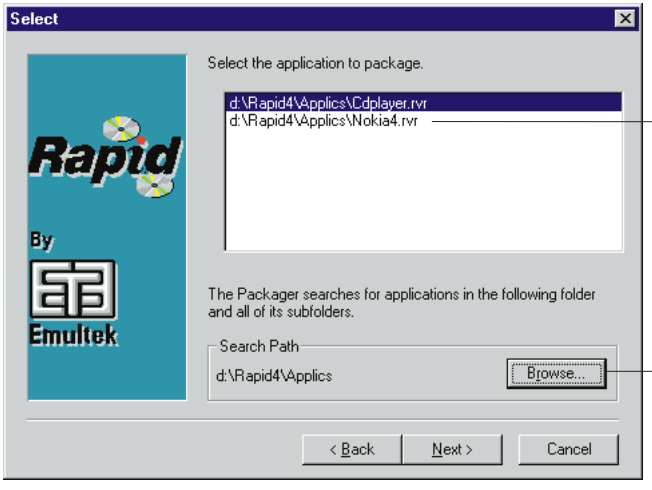
CREATING A PACKAGE

The Packager is a wizard which guides you, step-by-step, through the process of creating a package. The steps, each of which is described below, are:

- Selecting the Rapid application
- Selecting the application icon
- Specifying application files which are external to Rapid
- Adding a Readme file (optional)
- Adding specialized fonts
- Specifying the default directory in which to install the package
- Specifying the default name for the Start menu folder
- Choosing the directory in which Rapid builds the package

Selecting the Application

You specify the Rapid application to be packaged in this dialog box:

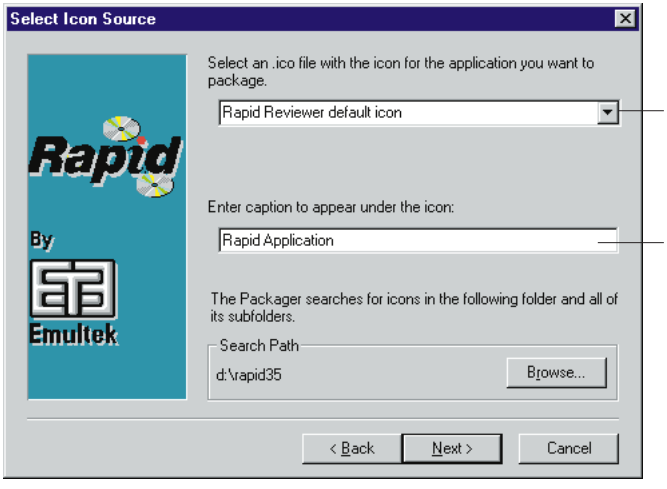


Browse as necessary to the desired directory and click on a listed application to select it.

❖ *NOTE: The Packager does not support long paths to an application. If you get an error message about exceeding the length of the string variable, you will have to save your application in a folder closer to the root.*

Selecting an Icon

You specify the application icon, that is, the icon in the program folder and in created shortcuts, in this dialog box:

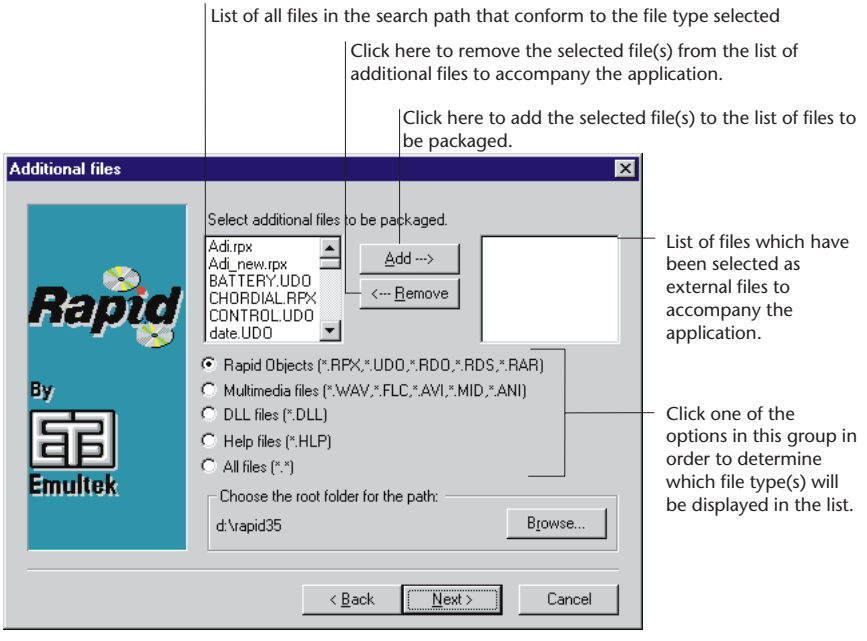


Either accept the default Rapid icon or open the drop-down box to display *.ico* files

Enter the caption which will appear with the icon in the application's program folder.

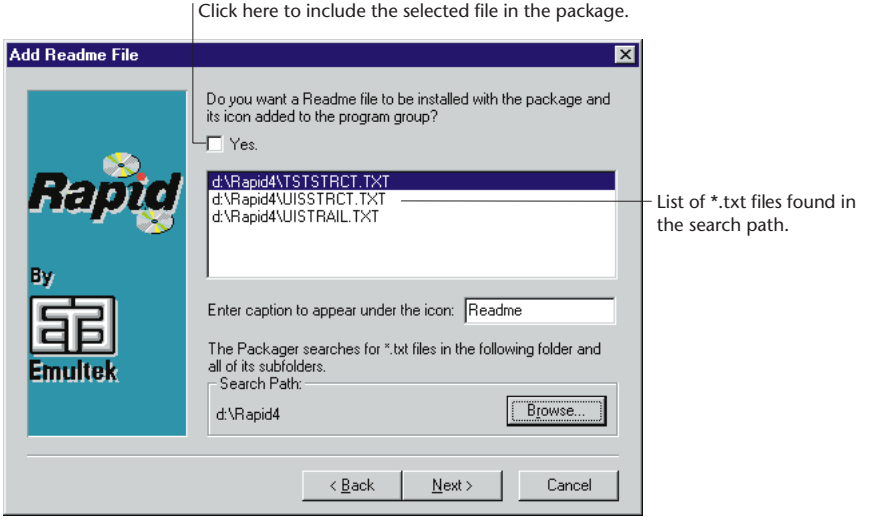
Adding External Files

If your Rapid application uses external files, including UDOs or specialized RPXs, you must be sure to include **all** such files in the compressed package. Readme files and fonts, however, are handled by other dialog boxes:



❖ *NOTE: You can Ctrl+Click or Shift+Click to select multiple files.*

Adding Readme Files

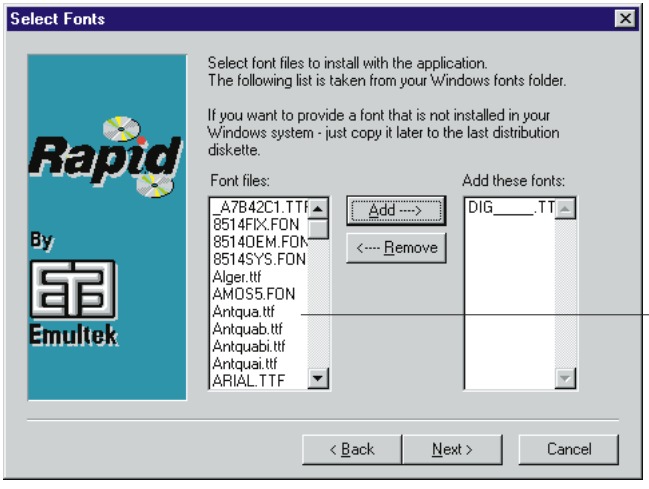


If the Yes option is checked, the selected file in the list will be accessible on the first distribution diskette and accessed by icon in the application's program folder.

Adding Fonts

If your application uses a specialized font, you must send it with the application. The end-user will be given the option of having the font installed during setup, or of installing it manually later. For this reason, font files are not included in the compressed files and are copied separately onto the last distribution diskette.

You specify fonts in this dialog box:

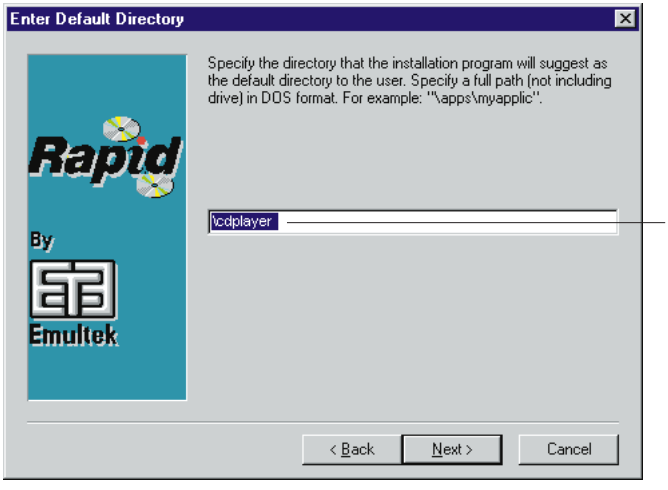


❖ *NOTE: You can Ctrl+Click or Shift+Click to select multiple files.*

Specifying the Default Directory

When the end-user "unwraps" the package, the setup program will suggest that the files be copied to the directory that you specify in the dialog box shown below. During setup, the end-user can over-ride this suggestion.

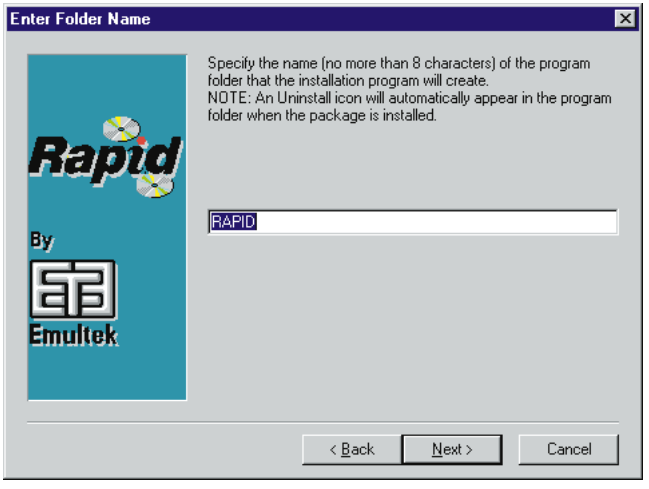
❖ *NOTE: You can also modify this parameter in the setup.ini file written to Disk1 of the disk set. See p. 22-13 for information on editing this file.*



Specifying the Folder Name

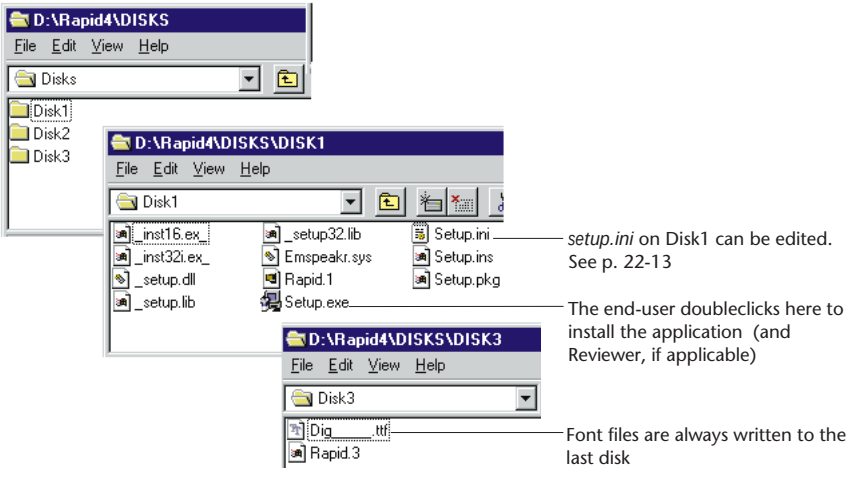
When the end-user installs the application, the setup program automatically adds icons to the program folder that you specify in the dialog box shown below. By default, the Packager suggests 'RAPID' as the program folder in which to place the three application icons:

- the application itself
- the readme file (if you specified one)
- the uninstall icon.



Specifying the Package Build Directory

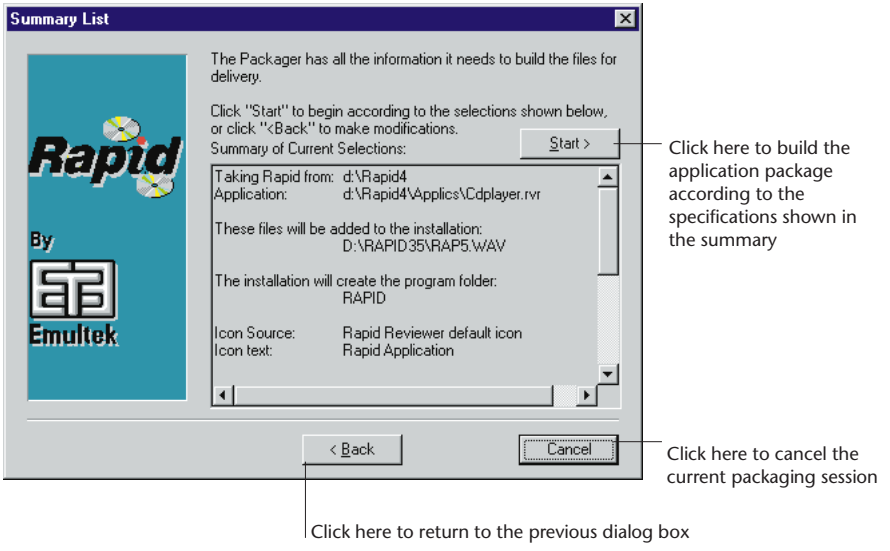
Rapid creates the subfolder "Disks" in the directory specified in the Select Package Directory dialog box and then creates a series of numbered subfolders, each one of which contains no more than 1.44 MB of data. You then copy the contents of each Disk*xx* folder (e.g., Disk1, Disk2, etc.) to a floppy disk.



Verifying Summary List

Before building the package, verify that the information you provided is accurate. Use the scroll bar, if necessary, to see the full list. However, if you realize that you made a mistake after clicking Start, don't worry: you can always use the *.pak* file to edit the package.

❖ *NOTE: Did you remember to remove all the paths from your application objects and logic?*



EDITING THE PACKAGE

There are two methods for editing the package after it has been created:

- When the Packager builds the package it writes an *<applicationName>.pak* file to the Rapid application directory. You can use this file to edit the package
- When preparing the self-installing disk set, the Packager writes the *setup.ini* file to Disk1. It contains information concerning the setup parameters that you set through the dialog boxes. You can use this file to edit the package, but do so carefully.

Editing Via the *.pak* File.

❖ *NOTE: When you launch the Packager, it automatically creates a file type association for PAK files.*

To edit the package using the *.pak* file:

1. Double-click on the *.pak* file. The Welcome dialog box opens.
2. Click Next. The Packager jumps to the Summary List dialog box, where it displays the current package selections.
3. To change a selection, click Back to move to the appropriate dialog box.
4. Return to the Summary List dialog box and click Start to create the edited package.

❖ *NOTES: Any time you open the Packager and select a previously packaged application, its .pak file automatically provides the previous package selections as defaults in the Packager dialog boxes.*

The .pak file is an ASCII file which can be edited directly in any text editor.

Editing Via the *setup.ini* File

Here are the relevant sections of a typical *setup.ini* file:

```
[general]

DefaultDirectory=\CDPLAYER (installation directory)
TemporaryDirectory=ZZZRAPID.TMP (erased when Packager is done)
DefaultGroup=CDPLAYER (program folder)

[application]

ApplicationFileName=cdplayer.rpd

[program]

InstallProgram=YES (Reviewer files to be included in the package)
RunApplication=RAPIDRT.EXE (see note below)
RequiredDiskSpace=5000KB
RapidVersion=4.0

[icon]

IconTitle=Rapid Application
IconSourceFile=RAPIDRT.EXE (the default Reviewer icon)
SetAsDDEServer=No (see note below)
```

```
[readme]

ReadmeFileExists=No
ReadmeTitle=Readme
ReadmeFileName=readme.txt

[extensions] (for internal use by the Packager; do not edit)

RapidObjects=*.RPX,*.UDO,*.RDO,*.RDS,*.RAR
Multimedia=*.WAV,*.FLC,*.AVI,*.MID,*.ANI
```

❖ *NOTES: You can modify the RunApplication parameter in order to specify an execution program other than Rapid Reviewer. If you change this setting, however, you should also consider the implications for the IconSourceFile and the ApplicationFileName parameters.*

If SetAsDDEServer in the setup.ini file is changed to 'Yes', the Rapid Reviewer will automatically open a CBT channel when the application is launched.

MODIFYING THE REVIEWER FILE LIST

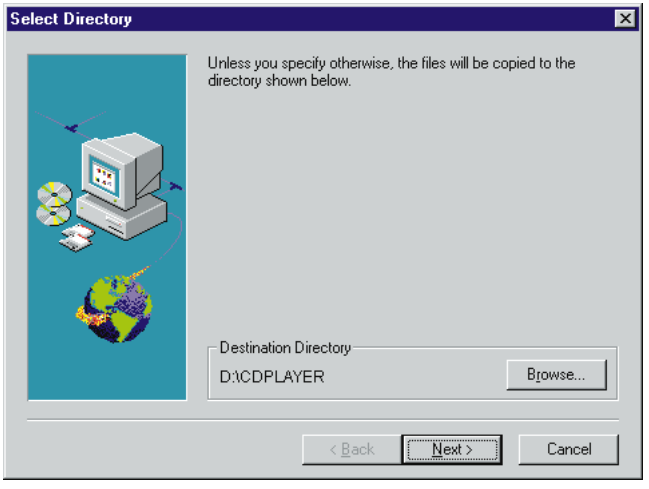
Among the Packager files you'll find *files.txt* — a list of the Rapid Reviewer files to be installed together with the application. The list included with the Packager is compatible with the most recent version of Rapid at the time of distribution. You can adapt the Packager to different Rapid versions by opening *files.txt* in an ASCII text editor and modifying the list.

In order to reduce the size of the package, you can also eliminate unused RPXes from the *files.txt* list. You should, however, take great care when changing this file.

INSTRUCTING THE END-USER

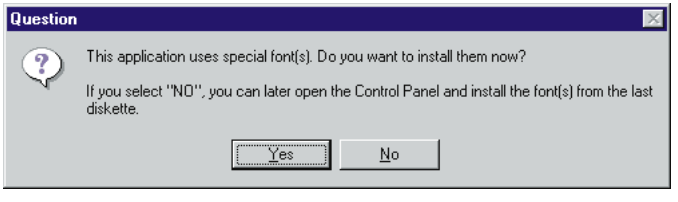
Installing and launching the application is a simple four-step procedure:

1. Double-click *setup.exe* on Disk1.
2. Either accept the default directory, or browse to another one.



❖ *NOTE: If you wish, you can explain to the user how to modify the default directory in the setup.ini file.*

3. Either install the fonts (if any) as part of the setup — or later from the last disk.



4. You should tell the user in which program folder he/she will find the icon for launching the application in the Reviewer. There will also be an Uninstall icon for removing the application files and, if applicable, an icon for opening the readme file.

CHAPTER 23

Plug-In for Netscape

Rapid's Internet plug-in lets you embed your specially-packaged Rapid applications in an HTML page, making them available to anyone using Netscape 3.0 or higher. Some examples of potential usages would be: letting the UI design team publish their virtual prototypes to the whole organization, always showing the latest version; promoting your product by making a free-play simulation available on your website; or including Rapid applications in CBT programs being written for Internet and Intranet distribution.

The Plug-in consists of two main components:

- the plug-in module itself (*irapid.exe*), which the end-user installs into his/her Netscape services. Available on the Rapid CD-ROM or on Emultek's website (http://www.emultek.com).
- a Packager utility (*ipack.exe*) which you use to bundle the Rapid simulation and any accessory files (such as data files, help files, and specialized fonts) into a compressed file with a ZRP extension.

  When you embed the ZRP file in an HTML page, you include information on the placement and size of the Rapid simulation window. You also specify whether the simulation should run inside the HTML page or in a standalone window.

In this chapter you will read about:

- Preparing a Rapid application for embedding.
- Packaging the Rapid application.
- Writing the HTML code that embeds the application.
- Using JavaScript to add buttons to the page that control the application window (optional).
- Putting the HTML page and the application on your web server.

PREPARING THE RAPID APPLICATION

See p. 22-3, “Preparing the Application for Packaging” for general guidelines on preparing a Rapid application for packaging. In addition, you should keep certain guidelines in mind when designing the application so that it will look good on a Web page. For example, ensure that it is the right size to fit on a page comfortably and make the application’s background color the same as the page’s.

PACKAGING THE APPLICATION

The Internet Packager (*ipack.exe*) produces a single, compressed file that includes the Rapid application and all its associated files (multimedia clips, fonts, and so on).

The main steps for packaging an application are:

- Launch the Internet Packager.
- Select the application to be packaged.
- Specify additional files to be packaged with the application.
- Specify font files to be included in the package.
- Specify a name and location for the package file.

To open the packager:

- Launch *ipack.exe* and click the Setup button in the Welcome dialog box.

The Select Application dialog box appears.

To select the application to be packaged:

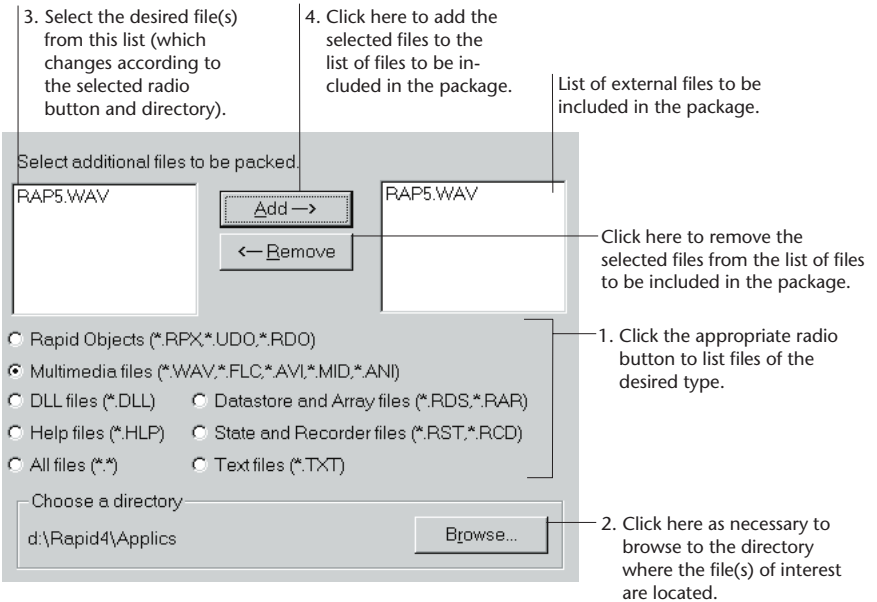
1. In the Select Application dialog box, browse to the directory in which the application is located and select the application from the drop-down list.
2. Click Next to advance to the Additional Files dialog box.

To select the external files to be included in the package:

- Repeat steps 1 through 4 (in the illustration below) until you have built a complete list of files to be included in the package.

  ❖ *NOTE: Make sure that you include ALL the files needed by the application.*

- Click Next to advance to the Select Fonts dialog box.



Partial view of Additional Files dialog box

To add specialized fonts to the package:

- The procedure for adding fonts is exactly the same as for the Packager utility supplied with Rapid. See p. 22-9, “Adding Fonts”.
- Click Next to advance to the Select Package Directory dialog box.

To specify a package name and location:

1. Enter a package name in the edit box. The default name is the application name. The Packager automatically adds the **.zrp* extension.

2. Click Browse as necessary to select the destination directory in which the **.zrp* file will be written.
3. Click Next to advance to the Summary List dialog box.

To confirm the package specifications and create the package:

- If the package specifications displayed in the Summary List dialog box are satisfactory, click Next to create the package.
- If necessary, click Back as many times as necessary in order to return to a dialog box and change the package specifications.

PREPARING THE HTML PAGE

There are two ways you can embed a Rapid application in an HTML page:

- "built-in": When the reader opens the page, the application begins to run in a predefined area of the page itself.
- standalone: When the reader opens the page, the application begins to run in a separate window.

In addition, you can include a parameter that allows the reader to jump directly to a page where s/he can download the plug-in.

To embed the application in the page itself:

- Add the following HTML code to your page:

```
<EMBED NAME=rapid
        SRC="applic.zrp"
        WIDTH=200 HEIGHT=200>
```

WHERE:	IS:
NAME	Any (optional) name you give to the embedded object. If (using JavaScript) you later add buttons that control the application window, you will need to refer to this name.
SRC	The filename of the package that you created earlier. You can give a complete URL

WHERE:	IS:
WIDTH *HEIGHT*	Give the size (in pixels) of the area in the page where the application will run. If you define an area that is smaller than the application, scrollbars will appear.

To embed the application in its own window:

1. Add the following parameter to the embed tag:

   STANDALONE=TRUE

2. Assign 0 to the width and height values.

Example:

```
<EMBED NAME=rapid
        SRC="applic.zrp"
        WIDTH=0 HEIGHT=0 STANDALONE=TRUE>
```

To help the reader to download the plug-in if it is not already installed on his/her computer:

- Add the following parameter to the embed tag:

  PLUGINSPAGE="*url*"

  where *url* is a page from where the Rapid plug-in can be downloaded.

Example:

```
<EMBED NAME=rapid
        SRC="applic.zrp"
        WIDTH=0 HEIGHT=0 STANDALONE=TRUE
        PLUGINSPAGE="http://www.emultek.com/plugin.htm">
```

❖ *NOTE: If a reader does not have the plug-in installed and s/he opens a page that contains a Rapid application* **without** *the PLUGINSPAGE parameter, s/he will be sent to Netscape's plug-in locator page.*

ADDING CONTROL BUTTONS (OPTIONAL)

You can add buttons to your page that control the Rapid application window. The buttons send commands to Rapid using JavaScript.

The example below inserts a button that stops the application when pressed:

```
<INPUT      type=button
            onclick="document.rapid.stop()"
            value="Stop">
```

WHERE:	IS:
rapid	The name given to the embedded object in the EMBED tag (see p. 23-4).
type	The argument of the INPUT tag that defines the object being added to the page.
onclick	The argument of the INPUT tag that defines the action to be taken when the button is clicked.
value	The text that appears on the button itself.

Controlling the Application Window in all Modes

The following commands can be sent to Rapid in both "built-in" and "stand-alone" modes.

To start the application:

```
<INPUT      type=button
            onclick="document.rapid.start()"
            value="Start">
```

❖ *NOTE: The name in the onclick string (in the above example, "rapid") must match the name in the EMBED tag.*

To stop the application:

```
<INPUT      type=button
            onclick="document.rapid.stop()"
            value="Stop">
```

To pause the application:

```
<INPUT      type=button
            onclick="document.rapid.pause()"
            value="Pause">
```

To resume the application:

```
<INPUT      type=button
            onclick="document.rapid.resume()"
            value="Resume">
```

❖ *NOTE: You can combine more than one command on one button. For example, the following button restarts the application (that is, stops and starts it):*

```
<INPUT      type=button
            onclick="document.rapid.stop(); document.rapid.start()"
            value="Restart"
```

Controlling the Application Window in Standalone Mode Only

To show the application window:

```
<INPUT      type=button
            onclick="document.rapid.show()"
            value="Show">
```

To hide the application window:

```
<INPUT      type=button
            onclick="document.rapid.hide()"
            value="Hide">
```

To put the application window in always-on-top mode:

```
<INPUT      type=button
            onclick="document.rapid.alwaysOnTop()"
            value="On Top">
```

To cancel always-on-top mode:

```
<INPUT      type=button
            onclick="document.rapid.cancelAlwaysOnTop()"
            value="Not On Top">
```

PUTTING THE HTML PAGE AND PACKAGE ON THE WEB SERVER

The HTML page and the Rapid package should be copied to the web server like any other HTML pages.

❖ **NOTE: The server must be configured to recognize the Rapid package file. This is done by setting the MIME type of files with the extension *.zrp to: application/vnd.rapid**

WHAT HAPPENS WHEN THE PAGE IS OPENED?

When a reader opens a page that contains a Rapid application, Netscape checks to see whether the Rapid plug-in is installed. If not, the reader is directed to the page from where the plug-in can be downloaded and installed. (This is either the the page you defined with the PLUGINSPAGE parameter in the EMBED tag, or Netscape's Plug-In Developers page.)

If the plug-in is installed, the package is downloaded and the files are extracted. The files are placed in a subdirectory of iRapid\Applics (where the plug-in files were installed). The name of the subdirectory contains the package's name.

After the files are extracted, the application begins to run in the page (or in its own window).

If the end-user wants to keep the application, s/he should copy the *.*rpd* or *.*rvr* file (and all other application files) from the subdirectory to a permanent location, while the page is still open.

The total number of downloaded applications on the end-user's disk never exceeds a predefined cache size. When the page is closed, the earliest of the downloaded applications will be deleted until the total is less than the cache size.

The end-user can, of course, delete them manually.

CHAPTER 24

Generating Reports

Rapid enables you to generate detailed text or graphic reports about various aspects of your application. To give just two examples, you can generate a text file that summarizes your application's logic, mode by mode, or print the layout of your objects as they appear in the Object Layout.

You can send any of the reports directly to a printer. When you generate a text report, you can tell Rapid either to print it directly (without creating a file), or to create an ASCII file. If you tell Rapid to create a file, you have the option of viewing it in a text editor.

The following information summarizes which types of reports can be sent to which destination(s):

REPORT TYPE	POSSIBLE DESTINATIONS
Object layout	Printer only
Prototyper layout	Printer only
State Chart plot	Printer only
Mode and object data *Mode and object tree*	Printer or file

You generate reports using the Application Manager's Reports menu.

OBJECT REPORTS

Rapid can generate several types of reports about the application's objects:

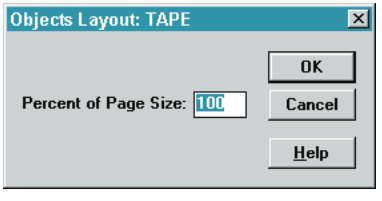
- Layout report
- Tree report
- Data report
- Prototyper layout report

Object Layout Report

The object layout report is a printed, graphical representation of the physical objects that make up the system, i.e., a copy of what you see in the Object Layout (without the nongraphic objects icon).

To print an object layout report:

- Choose Reports|Objects|Layout; the following dialog box opens:

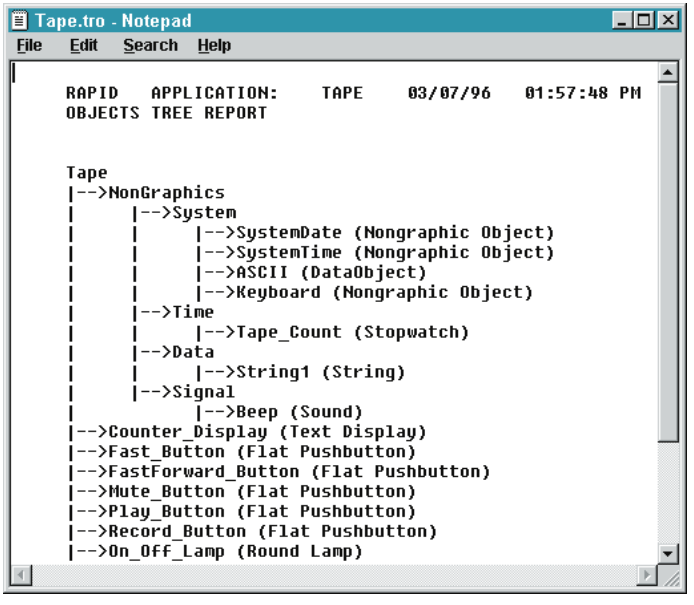


You can specify the size of the printed image in terms of the percentage of maximum size. The default is 100%, i.e., the layout image will be scaled to print as large as possible on a single sheet, while maintaining the same aspect ratio and orientation[1]. The smaller the image (i.e., the smaller the percentage of paper size exploited), the faster the printout; for a complex layout, the time difference may be substantial.

1 To get the largest possible printout, you may want to switch your printer setup from portrait to landscape or vice versa.

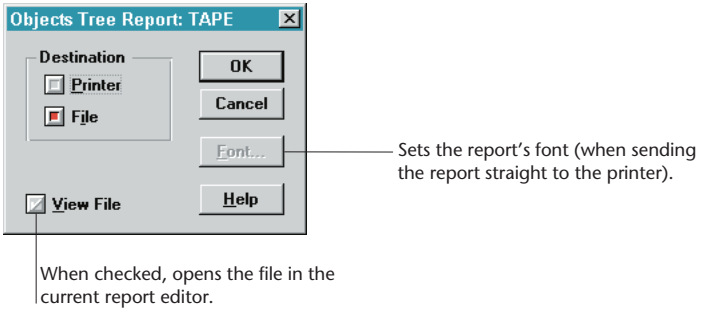
Object Tree Report

The object tree report is an ASCII text file that shows the hierarchy of the application's objects. A typical report (as viewed in Notepad) is shown here:



To generate an object tree report:

- Choose Reports|Objects|Tree; the following dialog box opens:



The default option is to generate a file and to view it in the report text editor.

To print the report:

1. Click the Printer radio button in the Destination group.
2. Click the Font button to set the report's font.

To save the report as a file:

1. Click the File radio button in the Destination group.
2. Check the View File checkbox to view the file after Rapid has generated it.

Rapid generates a file called <*Application name*>.*tro* and places it in Rapid's Reports folder. If there is no folder called "Reports," the file is placed in the Rapid folder. The report is written as an ASCII text file, which can be opened in any text editor.

Object Data Report

The object data report is a text summary of object information, including parent, type, parameters and properties. The following is part of a typical object data report, showing the data of a text display object called Counter_Display:

```
5. OBJECT: Counter_Display
    Parent: Tape   Type: Text Display

    Parameters:
   Position (pxl):'143 @ 64'  Size (pxl):'136@19'  Dynamic: 'false'
   Drag 'n Drop:'false'

            Properties:
      5.1.  Name: contents    Type: Data
            Parameters:
                  Text:''

    5.2.  Name: cursor    Type: Data
              Parameters:
                   Type:'Underline' Show state:'Hidden' Text blink period:500

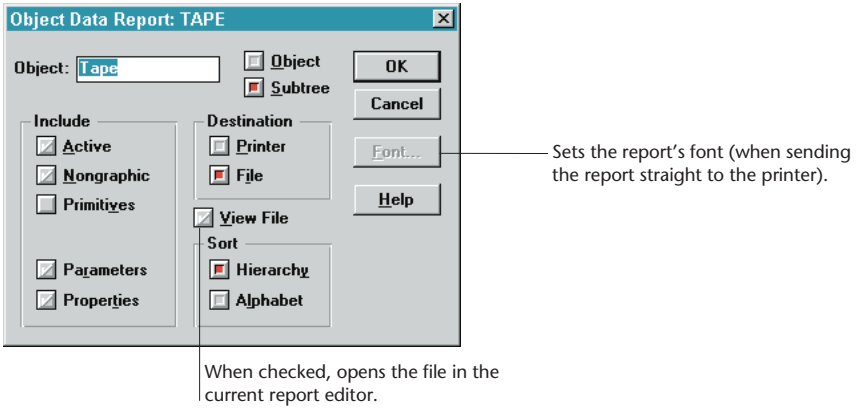
     5.3.  Name: cursorPosition    Type: Data
          Parameters:
              Cursor position:1

    5.4.  Name: fillColor    Type: Data
                Parameters:
                 Color:'gray'

    5.5.  Name: lineColor    Type: Data
                  Parameters:
               Color:'black'
```

To generate an object data report:

- Choose Reports|Objects|Data; the following dialog box opens:



The default object name is the selected object or, if no object is selected, the root object. The default scope is to generate report information for the object's hierarchical subtree (i.e., for the object and all its children).

To change the default object, type another object name in the edit box. If you want data on the named object only, click the Object radio button.

You can select any combination of check boxes in the Include group to determine which object types (active, nongraphic, primitive) and which object information (parameters and/or properties) will be included in the report.

The object parameters are parent, size and position, as seen in the object's parameter pane. The object properties are different for each object type. For example, each switch position is a property of switch objects. The object properties are described in detail in the Object Reference manual.

If you have chosen a subtree report, by default the objects will be sorted according to hierarchy. To sort the objects by alphabetical order of object name, click the Alphabet radio button.

The default option is to generate a file and to view it in the report text editor.

To print the report:

1. Click the Printer radio button in the Destination group.
2. Click the Font button to set the report's font.

To save the report as a file:

1. Click the File radio button in the Destination group.
2. Check the View File checkbox to view the file after Rapid has generated it.

Rapid generates a file called <*Application name*>.*trd* and places it in Rapid's Reports folder. If there is no folder called "Reports," the file is placed in the Rapid folder. The report is written as an ASCII text file, which can be opened in any text editor.

Prototyper Layout Report

This report is similar to the object layout report, except that it shows what is seen in the Prototyper, e.g., any text in the displays, the dial positions, etc. This report is only available if the Prototyper is running or paused. If the Prototyper is running, this report pauses it. Click the Prototyper's Resume button to continue once the report is printed.

MODE REPORTS

Rapid can generate several types of reports about the application's modes:

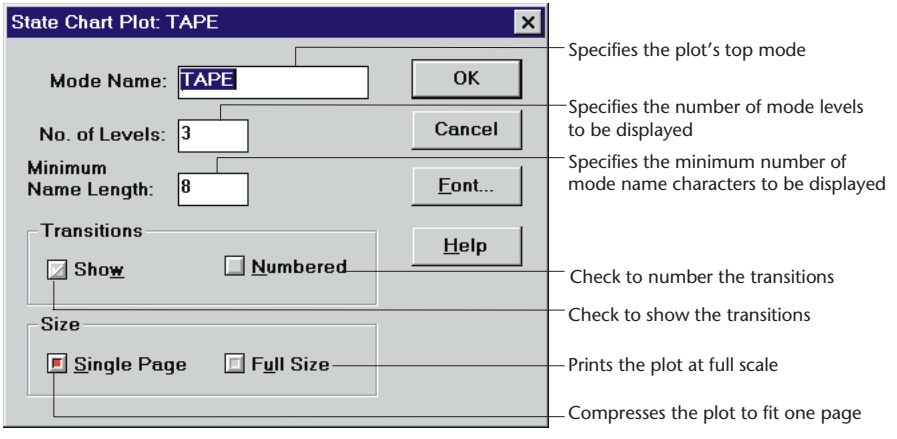
- State Chart plot
- Tree report
- Data report

State Chart Plot

The State Chart plot is a printout of the State Chart, showing the application's modes in a nested chart form.

To print a State Chart plot:

- Choose Reports|Modes|State Chart Plot; the following dialog box opens:

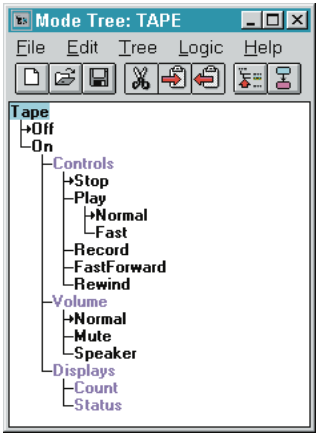


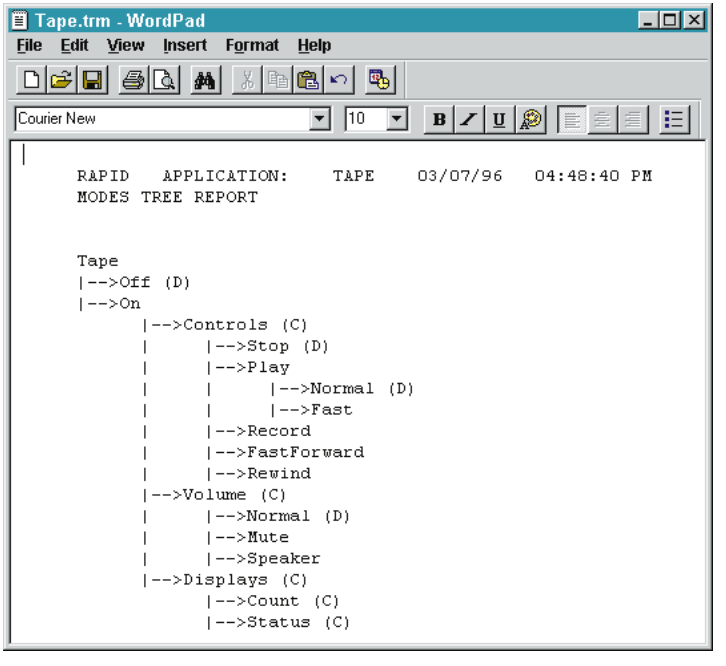
The default mode in the Mode Name text box is the root mode. To select a different top mode, type its name in the text box.

If you choose to print at full size, Rapid will break the State Chart into pages if necessary. The column and line number is printed at the top of each page and there is enough overlap for you to align the pages and paste them together.

Mode Tree Report

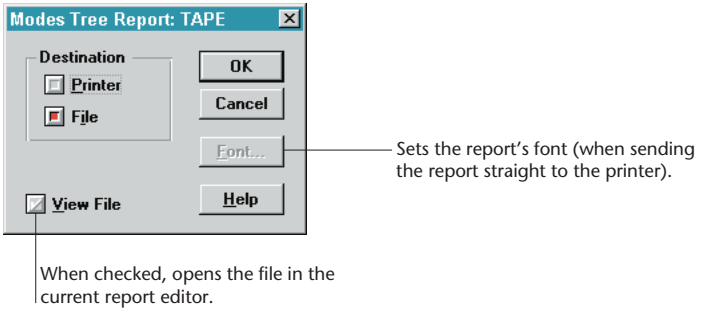
The Mode Tree report is an ASCII text file that shows the hierarchy of the application's modes. An application's Mode Tree, and its report (as viewed in WordPad) are shown here:





To generate a Mode Tree report:

- Choose Reports|Modes|Tree; the following dialog box opens:



The default option is to generate a file and to view it in the report text editor.

To print the report:

1. Click the Printer radio button in the Destination group.
2. Click the Font button to set the report's font.

To save the report as a file:

1. Click the File radio button in the Destination group.
2. Check the View File checkbox to view the file after Rapid has generated it.

Rapid generates a file called <*Application name*>.*trm* and places it in Rapid's Reports folder. If there is no folder called "Reports," the file is placed in the Rapid folder. The report is written as an ASCII text file, which can be opened in any text editor.

Mode Data Report

The mode data report is a text summary of mode information, including: parent, type, transitions, triggers, actions and activities. Mode data reports can also include the application's user-defined functions.

The following is part of a typical mode data report, showing the data of a mode called "watchout":

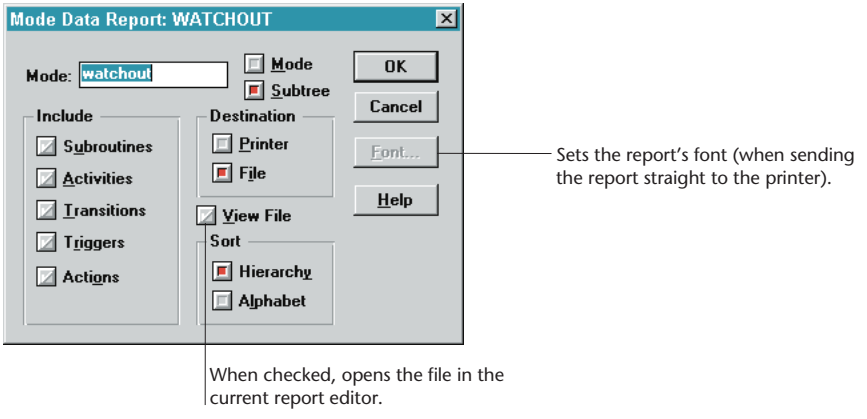
```
1.  MODE: watchout
       Parent: none     Type: root
       Note:

                        Mode Activities:
         a. Indicator1.reading := Pot1.value
         b. Pot1.value := watchout.value

       Transitions:
            a. Destination: safe      Entry type: Default
                     a1. Trigger:  & Pot1.value < 60
          b. Destination: warning      Entry type: Default
               b1. Trigger:  & Pot1.value >= 60 and Pot1.value < 90
        c. Destination: danger      Entry type: Default
           c1. Trigger:  & Pot1.value >= 90
```

To generate a mode data report:

- Choose Reports|Modes|Data; the following dialog box opens:



The default mode name is the selected mode or, if no mode is selected, the root mode. The default scope is to generate a report for the subtree. To change the default mode, type another mode name in the text box. If you want data on the named mode only, click the Mode radio button.

You can select any combination of check boxes in the Include group to determine which features will appear in the report. The default is to include all.

❖ *NOTES: Actions cannot be included unless triggers are included; triggers cannot be included unless transitions are included.*
Transition listings also show entry type.
If you include the application's user-defined functions, they are listed at the start of the report.

By default the modes will be sorted by hierarchy. To sort the modes in alphabetical order of mode name, click the Alphabet radio button.

To print the report:

1. Click the Printer radio button in the Destination group.
2. Click the Font button to set the report's font.

To save the report as a file:

1. Click the File radio button in the Destination group.
2. Check the View File checkbox to view the file after Rapid has generated it.

Rapid generates a file called <*Application name*>.*trl* and places it in Rapid's Reports folder. If there is no folder called "Reports," the file is placed in the Rapid folder. The report is written as an ASCII text file, which can be opened in any text editor.

EDITING REPORTS

Rapid generates four kinds of text reports:

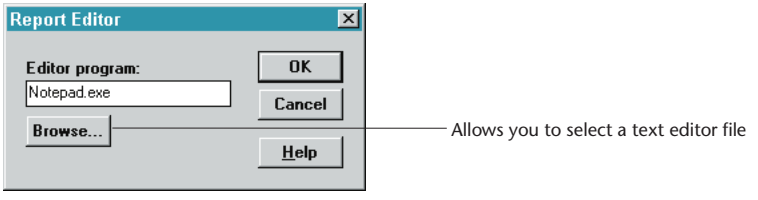
- Objects tree (TRO files)
- Objects data (TRD files)
- Mode tree (TRM files)
- Mode data (TRL files)

You can view and edit these reports in any text editor. By default, Rapid displays the files in Notepad—but you can choose to use whatever editor you like.

Choosing the Report Editor

To set the report text editor:

1. Choose Options|Report Editor; the Report Editor dialog box opens.



2. Type in the filename of a text editor (e.g., Wordpad.exe), or click the Browse button to select the file.

Editing Text Reports

To edit a report:

1. Choose Reports|Edit.
2. Select the report file that you want to edit; Rapid opens the report in whatever text editor you chose. (If you did not set a specific editor, Rapid opens the reports in Notepad.)

APPENDIX A

Graphic Performance in Rapid

Rapid has full support for 16, 24 and 32 bit color bitmaps (”true color”). This appendix explains the affect that different kinds of bitmaps have on Rapid’s overall performance.

Rapid Bitmap Objects

When reviewing the behavior of bitmaps in Rapid, take into account that there are several kinds of Rapid bitmaps:

- Bitmap objects (created by importing a bitmap into the Object Layout).
- All objects that are displayed by drawing a bitmap (like switches and potentiometers).
- Objects that were edited in the Object Editor.

Display Modes in Windows

Windows supports several display modes: 16 colors, 256 colors and more than 256 colors. Even though there are several levels of display modes above 256 colors, they are all referred to as “true color” mode.

256 color mode

256 color mode uses a palette of 256 colors that can be displayed at any one time. One of the side effects of working in 256 color mode is the “palette shift” that occurs when passing the focus from one window to another. This

happens because when a window becomes active, the system palette (which is a shared resource) changes to match the internal palette of the active window. All other windows have to match their colors (using their own internal palettes) to the newly selected system palette.

True color mode

In true color mode, there are no palettes. You can display in different windows as many color bitmaps with as many colors in them as you like without palette shift occurring.

The disadvantage of true color mode is that it requires a lot of memory (typically, three times as much memory as 256 color mode). The displaying of bitmaps is also slower (about 3 to 6 times slower, depending on the type of bitmap). This is caused by the representation of bitmaps in true color, which takes more memory than 256 color mode.

Because it is slower, true color mode is not so appropriate for fast animation, but it gives you better graphic quality while eliminating the palette shift problem.

Bitmap Formats

Windows supports several kinds of bitmaps:

- Monochrome bitmaps — black and white bitmaps, also called 1bpp (bits per pixel) bitmaps.
- 16 color bitmaps — also called 4bpp bitmaps.
- 256 color bitmaps (8bpp).
- 16, 24 and 32 bpp bitmaps, which use different levels of true color. Of all the true color levels, Windows 3.1 only supports 24bpp bitmaps.
- RLE bitmaps — bitmaps which are compressed using the RLE method. These come in two types: RLE4 (16 color) and RLE8 (256 color).
- 16 and 32 bitfield bitmaps — only supported by Windows NT.

  You can import all of these bitmap formats into Rapid except for the bitfield format.

Palette Mode vs. True Color Mode

Rapid's graphic performance is the speed with which it draws bitmaps in the Prototyper or Reviewer window. An application's graphic performance depends on two factors:

- The display mode.
- The bitmaps that were imported into the application.

Rapid divides all the display modes into two types: palette mode (16 and 256 color) and true color mode. Bitmaps are also divided to two groups: 256 colors or less (monochrome, 16 colors, 256 colors, RLE4 and RLE8) and true color bitmaps (16, 24, and 32 bpp). Rapid does not support bitfield bitmaps.

Rapid in palette mode

The **fastest** performance will be achieved when using only bitmaps with 256 or less colors in palette mode. This is also least demanding on memory.

Using true color bitmaps in palette mode is equally fast during runtime (once the Prototyper has already started) but the startup time will increase (the time that the Prototyper takes to start running the application). This is because at startup (or when the true color bitmap is first displayed) the bitmap is dithered to a 256 color bitmap. Dithering a true color bitmap (converting it to 256 colors) decreases the quality of the bitmap.

Rapid in true color mode

True color mode can be up to 6 times slower than palette mode, depending on the bitmaps you use.

256 color bitmaps are drawn 5 to 6 times slower than the same bitmaps in palette mode. True color bitmaps are drawn 3 to 4 times slower, but demand 3 times more memory than 256 color (or less) bitmaps. If you want speed in true color mode, use true color bitmaps. If you're short on memory, use 256 color bitmaps. You always have to pay for speed with memory.

The differences between the different types of bitmaps in the different display modes is summarized in the following table:

	PALETTE DISPLAY	TRUE COLOR DISPLAY
256 (or less) color bitmap	Fastest and least demanding on memory.	5 to 6 times slower. Demands less memory.
True color bitmap	As fast as a 256 color bitmap, more demanding on memory.	3 to 4 times slower, demands 3 times more memory.

Aspects of Rapid's Graphic Performance

Opaque bitmaps vs. transparent bitmaps

In Rapid 3.0x, bitmaps that had a transparent color were drawn slower than bitmaps without a transparent color. In Rapid 3.5, there is no difference.

Replacing the application palette

When you import a bitmap into the Object Layout and your display is in palette mode, Rapid' asks you: "Replace the application palette with the object palette?" There are three options:

- Yes. Rapid builds a new palette, based on the palette of the bitmap. This new palette replaces the previous palette of the Rapid application. All the bitmaps that were previously imported into Rapid will be redisplayed using the new palette. If the imported bitmap is a true color bitmap, Rapid's default palette will be used.
- No. The new bitmap will be displayed with the application palette.
- Cancel. Cancels the import bitmap operation.

Rapid's default palette is a balanced palette with a wide variety of colors. It should be appropriate if you intend to use several bitmaps with very different palettes. In this case, always answer No when you import a new bitmap.

❖ *NOTE: If you are building an application in true color display mode, this question will* **not** *appear.*

Stretching bitmaps

Stretching or shrinking a bitmap is a very slow operation and it reduces the quality of the bitmap.

When stretching a bitmap in the Object Layout the quality of the bitmap may be reduced but it will not affect the runtime performance. It is better to use a third-party graphics utility to stretch the bitmap and only then to import it into Rapid.

Stretching a bitmap during runtime (by changing the size or radius of a dynamic object) will decrease the application's graphic performance drastically. Try to avoid stretching bitmap objects during runtime whenever possible.

True color applications on a palette display

If you run an application that was designed in true color display mode on a display in palette mode, the application will use the default Rapid palette. This will look acceptable if the bitmaps in the application have a wide variety of colors. You might want to see how the application looks in palette display mode before distributing it.

To change the application palette, import a bitmap with the palette that you want, answer yes to the replace palette question, and delete the bitmap. This procedure will not affect the way the application will appear in true color mode because the palette is not used.

Application file size

All bitmaps are saved in the application file in their original format, exactly the way they were saved on disk before you imported them. All palette translation and dithering is made during editing and runtime and does not affect the way these bitmaps are saved.

Because of this, the application file size depends on the size of the bitmap files that are imported into Rapid. True color bitmap files are larger than 256 color bitmap files, and will increase the size of the Rapid application file in the same ratio.

If the size of the file is important, you can use RLE bitmaps. They slightly increase the startup time, but decrease the size of the file.

Multimedia

Video clips and animation files have palettes of their own. When a video clip starts and Rapid is in palette display mode, the behavior of Rapid is defined by the status of the Palette Update menu item in the Prototyper's Options menu (or in the Reviewer's System menu). The status is saved in the application file and determines the default state of the Prototyper or Reviewer.

If the menu item is checked (default) all the bitmaps are redrawn with the video clip's palette as soon as the clip starts. They are redrawn again with the application palette when the clip stops. This will make the bitmaps look relatively good while the clip is playing, but the window will blink when the bitmaps are redrawn. The redrawing might take some time if there are many bitmaps.

If the menu item is not checked, the application palette will not change. The video clip will be displayed with the application palette which will reduce the quality of the video.

In true color display mode, all these problem are not relevant, but again you might want to see how your true color application looks when running in palette display mode.

APPENDIX B

CBT Interface Message System

Once a CBT channel has been opened (in either the Prototyper —see p. 20-16, or the Reviewer—see p. 21-14), Rapid can exchange messages with any external program that supports DDE (dynamic data exchange).

❖ *NOTE: In order to exchange* **data** *(as opposed to messages), use the DDE Server or DDE Client object, neither of which require a CBT channel. For a full description of these objects, see Chapter 7: "Communication Objects".*

The Rapid CBT message system receives all of the messages (known as "packets") generated by the external program that is linked to Rapid. If the packet is in the proper format, the Message system forwards the packet to the CBT system. If the packet is invalid, it is sent to the Channel Manager.

PACKET FORMAT

Valid packets are made up of five parts, three mandatory and two optional. The parts are separated by delimiters:

PACKET PART	DESCRIPTION
Packet header	An asterisk (*).
Message handler address	An ASCII digit 0 - 6.
Command Number	ASCII digits or characters. The command number is a message to the handler. The command number can also contain text.
Parameters (optional)	ASCII text. If there is more than one parameter then a hash mark (#) is used as a separator. Most messages do not need data. For those that do, the data must be an ASCII string. **File names** - File names can be with or without a path or extension. - If no path is given, then the current directory is used. - If no extension is given, then the default extension for the particular command category is used.
End Of Transmission (optional)	ASCII 4 (EOT)

Delimiter

All parts of the packet are delimited by the character “!”.

If the delimiter symbol appears as part of the text of a string, then it must appear twice: “!!”.

Syntax

"*!message handler address!command number![parameters!EOT]".

(The brackets [ ] around the parameters and EOT parts indicate that they are optional).

Example:

***!1!8!\04**

WHERE:	IS:
*	packet header
1	Message Handler Address (Reviewer)
8	Command Number 8 (Show window)
\04	End of packet

Invalid Messages

If the CBT Message system receives an invalid packet, the handler that received the message sends a "NAK" packet in reply.

The syntax for a NAK packet is:
*!message handler address!the string 'NAK'!an error number or error description string.

Examples

A NAK packet sent after a poorly formed packet was received (format problem):

***!0!NAK!Invalid format!**

The 0 is the handler address. See the section "Message Handler Addresses" below.

An invalid event number would send:

***!4!NAK!Invalid event message number!**

MESSAGE HANDLER ADDRESSES

HANDLER	AD-DRESS	DESCRIPTION
Channel manager	0	Communication problem with the packet
Reviewer/Prototyper	1	Defines the Reviewer CBT mode (normal, reroute, etc.) and window state (hide title bar, etc.).
State	2	Transfers the application state.
Recorder	3	Controls the Rapid Recorder.
Event Packet	4	Events sent to or from Rapid.
Announcements	6	Messages sent to be displayed in Rapid.

Channel Manager

The Channel Manager handles any errors in the format of a received packet by sending a NAK packet to the sender's Channel Manager whenever such errors occur.

The Channel Manager (Handler !0!) supports a number of commands:

COMMAND NUMBER	COMMAND	DATA
1	Use numbers to define the error type ***	
2	Use strings to define the errors (default)	
3	Set response level for commands to 0	
4	Set response level for commands to 1	

COMMAND NUMBER	COMMAND	DATA
5	Set response level for commands to 2	

*** For internal use by Emultek only

Reviewer/Prototyper Commands

A Reviewer/Prototyper message (Handler !1!) is generally a simple command number. In some cases, it may be a command number and a data string, separated by a hash mark (#).

COMMAND NUMBER	COMMAND	DATA
1	normal event routing	
2	monitor events	
3	reroute events	
4	restore window	
5	iconize window	
6	maximize window	
7	fill screen	
8	show window	
9	hide window	
10	start application	
11	stop application	
12	pause application	
13	resume application	

COMMAND NUMBER	COMMAND	DATA
14	load application (from file)	name of application
15	query application	student responds with name of application
16	accept external (events)	
17	do not accept external (events)	
18	notify student channel opening	
19	notify student channel closing	
20	disable recorder controls	
21	enable recorder controls	
22	disable Prototyper controls	
23	enable Prototyper controls	
24	always on top	
25	cancel always on top	
26	set window position	x#y#width#height
27	slave window (no borders)	
28	release slave window	
29	query student name	student responds with student name
30	query position	student responds with x#y#width#height
31	query if present	If present, student responds with no data
32	close Reviewer	

State Manager Commands

A State Manager message (Handler !2!) is made up of two parts: a command number and a data string. The two parts are separated by a hash mark (#).

COMMAND NUMBER	COMMAND	DATA
1	load state from file	name of file
2	save state to file	name of file
3	get state from Rapid	the state organized in packets, each packet maximum 128 characters
4	send state to Rapid	the state organized in packets
5	state packet	packet no. + packet; all state packets need to be concatenated (the last packet number being 0).
6	state packet ack	packet number
7	reset to default state	
8	save compressed state to file	name of file

Recorder Commands

A Recorder message (Handler !3!) is generally a simple command number. In some cases, it may be a command number and a data string separated by a "#".

COMMAND NUMBER	COMMAND	DATA
1	recorder stop	
2	recorder play	

COMMAND NUMBER	COMMAND	DATA
3	recorder cue	
4	recorder record	
5	recorder rewind	
6	recorder step	
7	load recording	file name (with path)
8	save recording	file name (with path)
9	show recorder	
10	hide recorder	
11	at end	sent by recorder when playback reaches end and response level is 1 or higher

EVENT PACKETS

Event Rerouting

Event rerouting is achieved by returning the event messages generated by the Rapid Student back to the student. Such message use the event ID numbers 100 - 119. These messages are, of course, sent by Rapid with all the data needed to identify the event. However, it is not necessary to return this data to the Rapid Student - only the event ID number is necessary.

Rapid Student, in rerouting mode, continuously buffers the last 20 events that were sent out. This is done to enhance performance. If a message with one of the event ID numbers is returned to the Rapid Student, even without any of the data associated with that event, Rapid uses its copy of that event in the buffer and releases it to the state machine for processing.

This means that there can never be more than 20 outstanding events that can be activated by rerouting. However, response time of the simulation is greatly improved.

Event Monitoring and External Events

Messages sent by Rapid in Monitor mode always have the event ID number 0. A messages sent to the Rapid Student with message number 0 are interpreted as externally generated events and, if the Student has the external events enabled, these events will be parsed and passed on to the state machine for processing. Messages with message number 0 are handled independently from the buffered events.

Event Message Structure

Event messages are made up of eight parts. The parts are separated by delimiters:

- Packet header
- Message handler address
- Event ID number
- Object name
- Object property
- Property event
- Time stamp
- End of transmission

The parts are described below.

PACKET PART	DESCRIPTION
Packet header	An asterisk (*).
Message handler address	An ASCII digit 4.

PACKET PART	DESCRIPTION
Event ID number	A number from 100 to 119 or 0, specifying the event number. Numbers 100 to 119 specify rerouted events. When these arrive from the Rapid Student, they contain data specifying the event. The Rapid Student in rerouting mode buffers the last 20 events that were sent out. ID number 0, if received from a Rapid Student, specifies an event sent in monitor mode.
Object name	The unique Rapid object name. This is the same name that would appear in the recorder file.
Object property	The object's property that follows the object name. (i.e., $self, up, down, position1, size, etc.)
Property event	The event that the object generated (i.e., in, out, make, break, 100, etc.). This has the same format as that of the Recorder.
Time stamp (in seconds)	The number of seconds that have elapsed since the start of the simulation. This number is a floating point expressed as a string. It is optional for messages numbered 0 which are sent to Rapid.
End Of Transmission	ASCII 4 (EOT)

Delimiters

All parts of the packet are delimited by exclamation marks (!) and data parts by hash marks (#).

Example:

***!4!115!button1#$self#in#456.32!\04**

WHERE:	IS:
*	packet header
4	Message Handler Address (event)

WHERE:	IS:
115	Event ID number
button1	Rapid object name
$self	Object property
in	Property event
456.32	time stamp
\04	End of packet

ANNOUNCEMENTS

Announcements are windows with a message to the student, together with a button to enable the student to close the window or respond to the message.

The format of these messages are generally title and message text separated by “#”.

Use **(Handler !6!1!Title#Message text!)**

COMMAND NUMBER	COMMAND	DATA
1	display message****	title#message text

**** If Rapid is in always on top mode, the message appears behind the Rapid window!

RESPONSE LEVEL

The CBT channel can be set to various levels of response. The default response level is zero. At this level NAK and requested data will be the only possible response to commands.

Response level 1 enables notification messages being sent by Rapid whenever the user manually changes the state on one of the control functions. These notification messages are identical to the command that would have had to be sent to Rapid in order to achieve the same result. For example, if the user stops the Prototyper, then the message “*!3!11!” will be sent by the Prototyper.

Response level 2 is the same as response level 1 with the addition, that it also responds to any command (successful only) even if it does not require a response, by returning that command.

APPENDIX C

Writing DLLs for Rapid

DLLs used with Rapid are normal DLLs. The source code should be compiled and linked like every other DLL. The examples given in this appendix are in C, but you can use other languages.

The functions in the DLL can use the following types of variables:

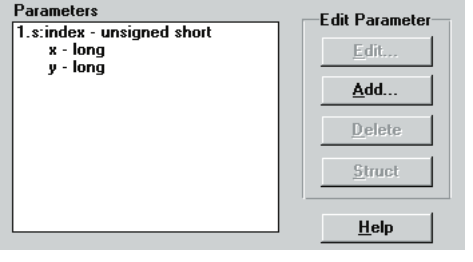
RAPID NAME	ALTERNATIVE NAME	STORAGE	RANGE OF VALUES
char	CHAR	1 byte	-128 to 127
unsigned char	UCHAR	1 byte	0 to 255
short	short int, SHORT	2 bytes	-32,768 to 32,767
unsigned short	USHORT, WORD	2 bytes	0 to 65,535
long	int, LONG, INT	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	unsigned int, UINT, DWORD	4 bytes	0 to 4,294,967,295
float	FLOAT	4 bytes	3.4e-38 to 3.4e38
double		8 bytes	1.7e-308 to 1.7e+308
string	char *	4 bytes	

When writing your DLL, bear in mind the following points:

- A function that does not return any value (void) should define “none” as its return type.
- A function that doesn’t receive any parameters should simply not define any parameter.
- If a function changes a string parameter (in C, char *) in such a way that the string is lengthened, an appropriate space must be allocated in Rapid before calling the function. This is done by assigning the string parameter an empty string, long enough for the returned value. If this is not done, Rapid may crash when calling the function.
- A pointer to a variable should be defined as a structure. For example a long*, is defined as “s: long”.
- Structures are always passed by reference (pointer to the structure).
- Rapid expects structures to be byte aligned. If your compiler aligns structures in another way, make sure to adjust the compiler options accordingly.
- Nested structures can be passed to the DLL by simply spreading the parameters as if the structure is not nested. For example, the following nested structure:

```
struct {
  unsigned short index;
  struct {
    long x;
    long y;
  } Point;
} IndexedPoint;
```

can be defined in Rapid as:



- If the parameters defined in the DLL object don’t exactly match the parameters of the called function, Rapid might crash.

- (C programmers.) The DLL functions must be declared as: `__declspec(dllexport)` and `WINAPI` (in Borland C).
- An entry point for the DLL should be defined. A template of the entry point function for Borland C is:

```
BOOL WINAPI DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason,
LPVOID lpvRsvd) {
  switch (fdwReason) {
  case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into the process's address space.
    // Put necessary initialization here
     break;
  case DLL_PROCESS_DETACH:
    // The DLLs bei from the process's address space.
    // Put necessary cleanup here.
     break;
  case DLL_THREAD_ATTACH:
    // A thread is being created.
    break;
  case DLL_THREAD_DETACH:
     // A thread is exiting cleanly.
    break;
  }
  return TRUE;
}
```

  For Visual C, use the following header:

```
BOOL WINAPI DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvRsvd)
```

- Never return a float, a double or a structure from a function. It you need to return such a type, pass a pointer to it as a parameter to the function, and have the function set it. For example, change the following function:

```
double distance(double a, double b) {
  return sqrt(a*a + b*b);
}
```

  to:

```
void distance(double a, double b, double *c) {
  *c = sqrt(a*a + b*b);
}
```

- Types defined by Windows and their Rapid equivalents:

WINDOWS TYPE	RAPID TYPE
PUCHAR	s: unsigned char
PCHAR, PCH, LPCH, PCCH, LPCCH	string
PSTR, NPSTR, LPSTR, PCSTR, LPCSTR	string
PSHORT	s: short
PUSHORT	s: unsigned short
INT	long
PINT, LPINT	s: long
UINT	unsigned long
PUINT	s: unsigned long
LONG	long
PLONG, LPLONG	s: long
ULONG	unsigned long
PULONG	s: unsigned long
FLOAT	float
PFLOAT	s: float
DWORDLONG , LONGLONG	double
PDWORDLONG , PLONGLONG	s: double
VOID	none
LPVOID, LPCVOID	string
PSZ	string
BOOL	long
PBOOL, LPBOOL	s: long
BYTE	unsigned char
PBYTE, LPBYTE	s: unsigned char
WORD	unsigned short
PWORD, LPWORD	s: unsigned short
DWORD	unsigned long

WINDOWS TYPE	RAPID TYPE
PDWORD, LPDWORD	s: unsigned long
HANDLE, HWND, HGDIOBJ	unsigned long
WPARAM, LPARAM	unsigned long
LRESULT	long
ATOM	unsigned short
HGLOBAL, GLOBALHANDLE	unsigned long
HLOCAL, LOCALHANDLE	unsigned long
HFILE	long
COLORREF	unsigned long
LPCOLORREF	s: unsigned long

APPENDIX D

Rapid Language Syntax

Token Specifications

A token class is defined by giving its token class terminal, followed by a colon (:), a regular expression, an optional scanner directive, and terminated by a semicolon. For example:

```
<identifier>    : [A-Za-z][A-Za-z0-9]*    ;
 <number>       : [0-9]+                  ;
<whitespace>    : [\s\t\r]+               ;
```

defines three token classes. Tokens in the class **<identifier>** begin with a letter and are followed by zero or more letters or digits. Tokens in the class **<number>** are composed of one or more digits. Tokens in the class **<whitespace>** consist of one or more sequential space, tab or carriage-return characters.

The language of regular expressions for token class specifications is defined as follows:

c — The character c itself, can be most any printable character, but regular expression metacharacters must be escaped (see below).

\c — The (escaped) character c itself, used to specify characters that normally would be interpreted as regular expression operators, such as [, *, and +, also used to specify certain nonprintable characters. The following are defined:

SPECIFICATION	ASCII VALUE	CHARACTER
\t	9	horizontal tab
\r	13	carriage return (Smalltalk newline cr)
\s	32	space

\ddd	The character whose decimal ASCII value is ddd, where d is a decimal digit (0-9).
a|b	Either of the regular expressions a or b (alternation).
ab	The regular expression a followed by the regular expression b (concatenation).
*a**	Zero or more repetitions of the regular expression a (Kleene Closure).
(a)	The regular expression a as an indivisible unit (parentheses are used in the traditional fashion to group operators of lower precedence together—closure has the highest precedence, followed by concatenation, and then alternation).
a+	One or more repetitions of the regular expression a (shorthand for aa*).
a?	An optional occurrence of the regular expression a (shorthand for (a|)).
[rangelist]	Shorthand for the alternation of all characters specified by the *rangelist*. A *rangelist* is a list of range expressions of the form: c a simple (or escaped) character. c-d shorthand for a list of all the characters between c and d inclusive based on ASCII character values) For example, **[abd-g]** is equivalent to **a | b | d | e | f | g**.
~[rangelist]	The alternation of all printable nonwhitespace characters in the complement of rangelist (i.e., all printable nonwhitespace characters except those specified).

Grammar Specifications

A grammar specification is a sequence of grammar rule specifications similar to those used by YACC. A grammar rule specification consists of a nonterminal, followed by a colon (**:**), one or more right-hand-side specifications separated by vertical bars (**|**), and is terminated with a semicolon (**;**). A right-hand-side specification is a regular expression.

Token Specification for English (1)

```
<id>          : [A-Za-z\128-\255][A-Za-z0-9_\128-\255]*(/[A-Za-z\128-
                \255][A-Za-z0-9_\128-\255]*)* ;

<argument>    : \<[A-Za-z\128-\255][A-Za-z0-9_\128-\255]*[\s\t]*\:[\s\
                t]*[A-Za-z\128-\255][\s\tA-Za-z0-9_\128-\255]*[\s\t]*\> ;

<parameter>   : \<[A-Za-z\128-\255][A-Za-z0-9_\128-\255]*[\s\t]*\> ;

<property>    : .[A-Za-z\128-\255]([A-Za-z0-9_\128-\255]*) ;

<keyword>     : ([a-z] | ([a-z]([A-Za-z0-9_]*)))\: ;

<integer>     : ([\+\-]?)[0-9]+  ;

<float>       : ([\+\-]?)[0-9]+.[0-9]+((e([\+\-]?)[0-9]([0-9]?))?) ;

<string>      : '(~['] | ' ')(~['] | ' ')+' | '' ;

<character>   : '(~['] | ' ')' ;

<space>       : [\s\t\r\n]+ ;

<comment>     : \" [\s\t\r\033\035-\255]*\" ;

<condDest>    : C\: ;

<defDest>     : D\: ;

<histDest>    : H\: ;

<deepDest>    : P\: ;

<notEqual>    : \<\> ;

<remark>      : \\\\[\s\t\r\033-\058\060-\255]*  ;
```

Token Specification for Japanese (81)

```
<id>          : ([A-Za-z\160-\223]|([\129-\159\224-\252][\064-
                \252]))([A-Za-z0-9_\160-\223]|([\129-\159\224-\252][\064-
                \252]))*(/([A-Za-z\160-\223]|([\129-\159\224-\252][\064-
                \252]))([A-Za-z0-9_\160-\223]|([\129-\159\224-\252][\064-
                \252]))*)* ;

<argument>    : \<([A-Za-z\160-\223]|([\129-\159\224-\252][\064-
                \252]))([A-Za-z0-9_\160-\223]|([\129-\159\224-\252][\064-
                \252]))*[\s\t]*\:[\s\t]*([A-Za-z\160-\223]|([\129-
                \159\224-\252][\064-\252]))([A-Za-z0-9_\160-\223]|([\129-
                \159\224-\252][\064-\252]))*[\s\t]*\> ;
```

```
<parameter> : \<([A-Za-z\160-\223]|([\129-\159\224-\252][\064-
            \252]))([A-
            Za-z0-9_\160-\223]|([\129-\159\224-\252][\064-
            \252]))*[\s\t]*\> ;

<property>  : .([A-Za-z\160-\223]|([\129-\159\224-\252][\064-
            \252]))([A-
            Za-z0-9_\160-\223]|([\129-\159\224-\252][\064-\252]))* ;

<keyword>   : ([a-z] | ([a-z]([A-Za-z0-9_]*)))\: ;

<integer>   : ([\+\-]?)[0-9]+  ;

<float>     : ([\+\-]?)[0-9]+.[0-9]+((e([\+\-]?)[0-9]([0-9]?))?) ;

<string>    : '(~['] | ' ')(~['] | ' ')+' | '' ;

<character> : '(~['] | ' ')' ;

<space>     : [\s\t\r\n]+ ;

<comment>   : \" [\s\t\r\033\035-\255]*\" ;

<condDest>  : C\: ;

<defDest>   : D\: ;

<histDest>  : H\: ;

<deepDest>  : P\: ;

<notEqual>  : \<\> ;

<remark>    : \\\\[\s\t\r\033-\058\060-\255]*  ;
```

Token Specification for Korean (82)

```
<id>        : ([A-Za-z\161-\254]|([\161-\172\176-\200\202-
            \253][\064-\252]))([A-Za-z0-9_\161-\254]|([\161-\172\176-
            \200\202-\253][\064-\252]))*(/([A-Za-z\161-\254]|([\161-
            \172\176-\200\202-\253][\064-\252]))([A-Za-z0-9_\161-\254
            ]|([\161-\172\176-\200\202-\253][\064-\252]))*)* ;

<argument>  : \<([A-Za-z\161-\254]|([\161-\172\176-\200\202-
            \253][\064-\252]))([A-Za-z0-9_\161-\254]|([\161-\172\176-\
            200\202-\253][\064-\252]))*[\s\t]*\:[\s\t]*([A-Za-z\161-
            \254]|([\161-\172\176-\200\202-\253][\064-\252]))([A-Za-z0
            -9_\161-\254]|([\161-\172\176-\200\202-\253][\064-
            \252]))*[\s\t]*\> ;

<parameter> : \<([A-Za-z\161-\254]|([\161-\172\176-\200\202-
            \253][\064-\252]))([A-Za-z0-9_\161-\254]|([\161-\172\176-
            \200\202-\253][\064-\252]))*[\s\t]*\> ;
```

```
<property>   : .([A-Za-z\161-\254]|([\161-\172\176-\200\202-
               \253][\064-\252]))([A-Za-z0-9_\161-\254]|([\161-\172\176-
               \200\202-\253][\064-\252]))* ;

<keyword>    : ([a-z] | ([a-z]([A-Za-z0-9_]*)))\: ;

<integer>    : ([\+\-]?)[0-9]+  ;

<float>      : ([\+\-]?)[0-9]+.[0-9]+((e([\+\-]?)[0-9]([0-9]?))?) ;

<string>     : '(~['] | ' ')(~['] | ' ')+' | '' ;

<character>  : '(~['] | ' ')' ;

<space>      : [\s\t\r\n]+ ;

<comment>    : \" [\s\t\r\033\035-\255]*\";

<condDest>   : C\: ;

<defDest>    : D\: ;

<histDest>   : H\: ;

<deepDest>   : P\: ;

<notEqual>   : \<\> ;

<remark>     : \\\\[\s\t\r\033-\058\060-\255]*  ;
```

Syntax Definition

```
Executable             : Method
                       | Function
                       | EntryBlock
                       | ExitBlock
                       | ActionBlock
                       | <remark>;

Function               : 'subroutine' SubroutineDeclaration StatementBlock;

EntryBlock             : 'entry' StatementBlock;

ExitBlock              : 'exit' StatementBlock;

ActionBlock            : 'action' StatementBlock;

SubroutineDeclaration  : FunctionName ReturnValue
                       | FunctionName;

ReturnValue            : 'return:' <parameter>
```

```
StatementBlock      : 'begin' Stmts 'end'

FunctionName        : MacroName
                    | 'is' MacroName
                    | 'has' MacroName
                    |  SubroutineName
                    |  'is' SubroutineName
                    |  'has' SubroutineName;

MacroName           : <id>;

SubroutineName      : SubroutineName ArgumentPair
                    | ArgumentPair ;

ArgumentPair        : keyword <argument>

Stmts               : StatementList
                    | ReturnSign Condition ';'  ;

StatementList       : StatementList Stmt ';'
                    | Stmt ';';

Stmt                : <space> ';';
                    | Activity ';'
                    | ReturnSign Expression ';'
                    | <remark> ';';

Method              : Activity
                    | Transition ;

Transition          : Destination ';' Triggers
                    | Destination ';'
                    | Destination ';' <remark> ;

Triggers            : TriggerList ReturnSign Condition
                    | TriggerList ReturnSign
                    | TriggerList
                    | ReturnSign Condition ;

ReturnSign          : '&'

TriggerList         : TriggerList '!' Trigger
                    | Trigger;

Trigger             : Object Event
                    | Property Event ;

Event               : <id>;
```

```
Destination          : 'Internal'
                     | 'internal'
                     |
                     | DestinationName
                     | <defDest> DestinationName
                     | <condDest> DestinationName
                     | <histDest> DestinationName
                     | <deepDest> DestinationName ;

DestinationName      : <id> ;

Condition            : Condition 'and' BooleanExpression
                     | Condition 'or' BooleanExpression
                     | BooleanExpression ;

BooleanExpression    'not' BooleanExpression
                     | BooleanRelation
                     | BooleanElement 'is' UnaryFunction
                     | BooleanElement 'is' KeywordList
                     | BooleanElement 'has' UnaryFunction
                     | BooleanElement 'has' KeywordList

BooleanElement       : '-'BooleanElement
                     | '(' Expression ')'
                     |  Object
                     |  Property

BooleanRelation      : Expression '=' Expression
                     | Expression <notEqual> Expression
                     | Expression '<' Expression
                     | Expression '>' Expression
                     | Expression '<=' Expression
                     | Expression '>=' Expression
                     | '(' Condition ')' ;

Act                  : LValue ':=' Expression
                     | LValue 'x' ':=' Expression
                     | LValue 'y' ':=' Expression
                     | KeywordExpression
                     | LValue UnaryFunction;

Element              : '-'Element
                     | Element ;
                     | Element UnaryFunction
                     | Element 'x'
                     | Element 'y' ;
```

```
BinaryFunction      : BinaryFunction '+' Element
                    | BinaryFunction '-' Element
                    | BinaryFunction '*' Element
                    | BinaryFunction '/' Element
                    | BinaryFunction 'div:' Element
                    | BinaryFunction 'modulo:' Element
                    | BinaryFunction '@' Element
                    | BinaryFunction '|' Element
                    | Element;

KeywordExpression   : Element KeywordList ;

KeywordList         : KeywordList <keyword> Element
                    | <keyword> Element ;

Expression          : BinaryFunction
                    | KeywordExpression ;

Object              : IndexList ']'

IndexList           : IndexList ',' Index
                    | ArrayStart Index

ArrayStart          : Object '['
                    | Property '['
                    | String '[' ;

Index               : Expression ;

Element             : '(' Expression')'
                    | Object
                    | Property
                    | String
                    | <float>
                    | <character>
                    | <integer>

Object              : <id>
                    | <parameter>

Property            : Object <property>

String              : <string>

UnaryFunction       : <id>
```

Glossary

accept external events — A command given to Rapid to enable it to accept—and react to—events sent from an external program.

action — An application operation that is performed on an object when a particular transition is triggered. The action is coupled to the trigger.

active area — An area of an object that has special characteristics, for example a hot spot.

active mode — As the application runs in the Prototyper, Rapid goes from mode to mode, executing all of the activities defined within each mode. An active mode is a mode whose activities are currently being executed.

active object — A graphic element that plays an active role in an application's behavior. Examples of active objects are switches and displays.

active primitive object — A primitive object that has been named. An active primitive can be manipulated in the Prototyper.

activity — An application operation performed on an object (using its properties and functions) when a particular mode is active.

Append (button) — The Logic Editor button that appends the selected object/property/function combination to the edit box.

application — A prototype or simulation designed with Rapid.

Application Manager — The Rapid window from which you can open all the other modules: Object Layout, Logic Editor, etc.

argument — A variable that is needed by some functions to be complete. For example, the := function of the text display object's *contents* property needs to know what string to assign. The string is the function's argument.

background — Any area within the object's enclosing rectangle that is not actually occupied by the object; the color of lighter areas (default: white).

breakpoint — A logic statement or a mode entry/exit tagged as a point at which the Debugger automatically pauses the application in the Prototyper.

CBT channel — The name of the DDE channel that links a Rapid simulation in student mode with a CBT application.

child — A mode or an object that was allocated a particular parent. In Rapid's hierarchical structure, every object or mode is a child of another object or mode. The only exceptions are the root object and root mode.

commlink object — A communications object that facilitates the passing of data between a Rapid application and an external device, via RS-232 serial communications.

compound multimedia object — A multimedia object which relies on data stored in a file for playback. For example, the wave audio object only plays data stored in *.wav* files.

concurrent modes — Modes that are always active, together with all their sibling modes, when their parent is active.

condition — A logic phrase that tests the status of an object (or a mode). Conditions are used, alone or together with events, to trigger transitions.

constant — A type of switch or pushbutton that remains in its new position after you press it once.

DDE channel Dynamic Data Exchange channel. A mechanism that allows separate Windows programs to exchange data between them.

DDE client One side of a DDE link. The client waits for the server to initiate the exchange.

DDE server One side of a DDE link. The server initiates the exchange with the client.

Debugger An integrated utility that pauses the Prototyper at specified breakpoints and/or executes the application's logic one step at a time. When the Prototyper is paused, the logic statement about to be executed is highlighted in the Logic Editor.

deep history transition When a deep history transition is made to a mode with multiple generations of descendants, the most recently active descendant at each generational level becomes active.

default mode An exclusive child mode that becomes active when a default transition occurs to its parent. A default mode is marked by an arrow in the Mode Tree.

default state The application state as defined in the Object Layout, before you make any changes in the Prototyper.

default transition When a default transition is made to a parent mode, its default child mode becomes active.

default scale The scale that the next new application will adopt.

delimiter An ASCII value recognized by Rapid as signalling the end of a package of incoming data, when passing data via the commlink object.

destination window A frame object on which you project a multimedia image as part of your application's layout.

device element A data file used by a compound multimedia object.

drag 'n drop object A graphic object that can be dragged in the Prototyper using the mouse.

dynamic object	A graphic object whose position can be changed by the application's logic.
entry activity	An activity that occurs when the mode becomes active.
event	A momentary change in the status of an object or mode that is used to trigger a transition from one mode to another. For example, when you press a pushbutton or a switch.
exclusive modes	Sibling modes, only one of which can be active while their parent mode is active.
exit activity	An activity that occurs at the instant the mode becomes inactive.
external device	Any device that is able to receive data from a serial port via a serial cable. A Rapid application can communicate with an external device using the commlink object.
external event	An event that is generated by a CBT application and sent to the Rapid simulation via the CBT channel. If the Rapid simulation has received the "Accept External Events" command, it will react to the external event.
exported function	A function of a user-defined object that is available for use in a parent application.
field (data store)	A column of a data store record that contains either string, number or integer data.
fill color	The color of the object's flat surfaces (default: light gray).
Find and Replace	A window in which you can search for logic statements of specified types that contain a specified string and/or object. Once found, you can go directly to a statement in the Logic Editor or carry out replace operations.
function	A "command" that tells Rapid to alter one of the properties of an object.
function list	The section of the Logic Editor that displays the functions, conditions and events of the selected object property.

group	A collection of objects in the Object Layout that you can move and resize together.
history transition	When a deep history transition is made to a mode, its most recently active child becomes active.
hot spot	An area of an active object that is "mouse sensitive." Click this area to activate the object.
indication	The color of an object's pointer, a lit lamp and a stepper switch's text (default: red).
inspector	A window which displays the contents and/or state of an object while the Prototyper is running or paused.
internal transition	A type of transition that allows execution of actions that are triggered by conditions and/or events, without leaving the mode. A mode can have several internal triggers, each with its own associated action.
Item	One of the three parameters that you need to define when setting up a DDE channel.
Librarian	The utility that allows you to manage your object library files and to manipulate the objects that they contain.
line color	The color of the thin line that outlines the various surfaces of the object, any text and external calibration markers (default: black).
line width	The width of the line, circle and frame object's outline.
Logic Editor	The window in which you build an application's logic. In the Logic Editor you define events, conditions, actions and activities.
Logic Info pane	The pane at the foot of the Logic Chart that provides information about the selected mode's transitions or the selected transition's actions.
Logic Palette	The Logic Editor's floating palette that provides a list of all the properties, functions, conditions and events of an application's objects.

logic validity check	When using the Find and Replace utility, Rapid only carries out a replacement after verifying that the new logic statement that would be created is logically valid.
MCI	Media Control Interface provided by Microsoft® Windows for controlling multimedia devices such as audio hardware and video players.
mode	A collection of activities that are grouped together and activated when the mode is active.
mode activity	An activity that continuously occurs while the mode is active.
mode condition	A condition that tests whether a particular mode is active or not.
Mode Tree	The window in which you build the application's hierarchical tree of modes, and define the transitions between them.
momentary	A type of switch or pushbutton that remains in its new position only as long as you press it.
multimedia device	A utility installed on a multimedia personal computer and controlled by an MCI driver; some examples of MCI device types are animation, CD audio, digital video and MIDI sequencer.
nongraphic object	An invisible, "behind the scenes" object that plays a role in the application's behavior, such as a timer or a data store.
normal mode	The status of a Rapid simulation when its event flow is entirely internal.
notes	Text comments that you can attach to any object or mode. Notes will appear on any printed report alongside their objects. You can directly add comments to logic in the Logic Editor.
object	The basic building-block of a Rapid application. Each object has a pre-defined set of properties and functions that you use to give your application its logical behavior.

object class A collection of objects of a similar type, e.g. switches.

Object Editor The window that is used to edit objects' bitmaps, pointers and active areas.

Object Layout The module used to add the application's objects and set up their physical layout.

object palette The section of the Object Layout window that contains the object icons. You select objects from the palette to add them to the application.

orphan An object or mode whose parent has been deleted or cut.

Packager An application distribution utility that creates a self-installing disk set for either an application alone, or for an application together with the Rapid stand-alone Reviewer.

parameter One of an object's basic characteristics that is defined in its parameter pane, e.g., name, parent and position.

Parameter Pane The dialog box that defines the basic parameters of a graphic object in the Object Layout.

parent A mode or object that transmits certain characteristics to its child. In Rapid's hierarchical structure, every object or mode has a parent object or mode. The only exceptions are the root object and root mode.

parent application An application that contains a user-defined object.

paste buffer Rapid's internal memory where it places cut or copied items (objects, activities, modes, etc.) Each type of item has its own buffer.

primitive object A graphic element used to enhance the application's physical appearance, but which has no active role in its logic. Examples of primitive objects are frames, lines and bitmaps, You can make primitive objects active by assigning them a name.

property	A characteristic of an object that describes a specific aspect of an object's behavior. Each property holds a set of functions and can be used to define triggers and activities.
property list	The section of the Logic Editor that displays the properties of the selected object.
Prototyper	The module used for running the application and testing its behavior.
record (data store)	A subsection (a line) of a data store, made up of a series of fields.
Recorder	The Prototyper utility that enables you to record, play back and save to file sequences of the application's behavior as it runs.
recorder file	The file that results from saving a recorded Prototyper sequence. The recorder file is an editable text file.
report	A printed or on-screen account of various aspects of the application.
reroute mode	The status of a Rapid simulation when it sends all its events to the CBT application via the CBT channel and does not respond unless it receives an "acknowledge" signal.
Reviewer	Emultek's runtime module that acts as a stand-alone Prototyper.
root mode	The highest mode that bears the name of the application, automatically created by Rapid. As you add modes, the mode tree grows downwards from the root mode.
root object	The highest object (the layout itself) that bears the name of the application, automatically created by Rapid. All active objects are descended from the root object.
RPX object	Rapid external object—a Rapid-compatible "plug-in" object, created outside Rapid. They are written by a programmer in a language such as C++.

shading The color of the highlighted areas that give the object its 3-dimensional appearance (default: dark gray).

sibling An object or mode that shares the same parent as another object or mode.

state A "snapshot" of the application's current situation as it runs in the Prototyper, including the status of all its objects and modes.

State Chart The module that presents a graphical, "bird's eye view" representation of the application's modes and the transitions between them.

state file The file that results from saving the Prototyper's state. The state file is an editable text file.

student mode The status of a Rapid simulation when it is linked to a CBT application via the CBT channel.

system cursor A cursor automatically generated by Rapid and always available for use in the Logic Palette, in conjunction with the mouse system object.

system object Nongraphic objects automatically generated by Rapid and always available to the application logic via the Logic Palette. These objects, which cannot be edited or deleted, include: keyboard, mouse, ASCII, system data and system time.

Topic One of the three parameters that you need to define when setting up a DDE channel.

Trace option The Prototyper option that allows you to follow the application's logic as it runs. When this option is activated, the currently active modes are highlighted in the Mode Tree.

transition A path from one mode to another. A transition is comprised of a destination and a trigger. Transitions determine the flow of the application's behavior.

trigger A set of events or conditions (or both) that causes a transition to take place.

user cursor	A cursor defined via the cursor object in the Object Layout, to be used in conjunction with the mouse system object.
user-defined function	A block of Rapid logic that can be invoked as a single function.
user-defined object	An application that is used as a discrete object within other applications. A collection of objects and logic that behaves as a single object.

Index

@ coordinate symbol 13-4

acceptExternalEvents function 7-42
Actions 10-16
  building 12-27–12-30
  column in Logic Editor 12-5
  compared to entry activities 10-16
  defined 10-6
  in data reports 24-11
  internal 10-16
  moving to activities column 12-34
  order of execution 12-30
Active areas
  editing 9-10, 9-26–9-27
Active areas-editing mode 9-26
Active objects 4-1–4-56
  adding to layout 2-14–2-18
  animated object 4-50–4-51
  automatically selecting children 2-26
  defined 2-2
  displays 4-39, 4-41, 4-43–4-47
  flow objects 4-52–4-55
  hierarchy 2-11
  hot spots 20-3
  indicators 4-3–4-12, 4-16
  inspecting contents during runtime 15-8
  lamps 4-2
  logic of, general 13-2–13-3
  making drag ‘n drop 2-20
  making dynamic 2-20
  menu 4-47–4-49
  orphans 2-30, 2-32
  parent parameter 2-22
  pipes 4-52–4-55
  pointers 4-17–4-18, 4-20–4-21
  position parameter 2-22
  potentiometers 4-25–4-29, 4-31
  pushbuttons 4-31–4-32
  sector dial 4-22, 4-24
  selecting 2-24
  size parameter 2-23
  switches 4-32–4-39
  tanks 4-56
  valves 4-56
Activities 10-13–10-15
  building 12-27–12-30
  column in Logic Editor 12-6
  moving to actions column 12-34
  order of execution 12-30
  pasting by type 12-32
  type, changing 12-29
  user-defined 18-2
Add command 2-15
Add Members [to Group] command 2-34–2-35
Adding
  bitmaps 2-50–2-51
  fields in a data store 5-18
  graphic objects 2-14–2-15, 2-17, 2-19
  label objects 3-4
  line objects 3-3
  modes 11-3
  properties to primitive objects 3-9
  user-defined objects to an application 19-14
addMore: copiesOf: function 13-30
Align commands 2-36
  Bottom 2-37
  Left 2-36
  Right 2-37

Align commands *(cont.)*
Top 2-37
Aligning objects 2-36–2-38
and logical operator 12-23
Animated object 4-50–4-51
changing bitmaps 4-50
logic 13-58–13-60
moving 13-59
running 13-58
setting colors 4-52
antialiasing 4-21
Application Manager 1-1–1-10, 7-25–7-51
Application objects 9-5
saving to Object Layout 9-8
selecting from Object Layout 9-6
Application parameter
CBT channel 20-16
DDE 7-2
Simulation Manager object 7-39
applicationName property 7-44
Applications
creating 1-3
debugging 15-1, 15-10
opening 1-3
optimizing 16-17
packaging 22-1, 22-15
preparing for packaging 22-3, 23-2
running in the Prototyper 20-2–20-6
running in the Reviewer 21-9–21-10
runtime 1-5
saving 1-4
saving as user-defined objects 19-5
testing 20-3
Arguments
changing the number of in user-defined functions 18-20
for multimedia objects 6-9
SEE Functions, arguments
generating 18-14
in user-defined functions 18-13
typing in user-defined functions 18-17
Array object 5-11–5-13
addressing elements 13-26
SEE ALSO Array of objects
as a property 3-9
logic 13-26–13-27
queries and searches 13-27
Array of objects
as argument in user-defined functions 19-26
deleting generated objects 13-30
events of 13-31
generating objects with 13-30
in user-defined objects 19-26
lastEventIndex function 13-31
logic 13-29
ASCII object 5-28
ASCIIValue function 13-11
Auto repeat
mouse object 13-45
pushbutton objects 4-32
stepper switch object 4-36
Automatic type conversion 13-13

Background color element 2-43
menu object 4-48
Background color, Object Editor 9-15
Backups, application 1-4
Bar indicator objects 4-11
Barrel dial object 4-12–4-14, 4-16
borrow event 13-49
carry event 13-49
changing rotation rate 13-48
editing colors 4-16
logic 13-47–13-49
rotation, determining 4-14
rotation, starting/stopping 13-47
setting size 4-13
setting value dynamically 13-48
Baseline height 4-9
Baseline offset 4-5, 4-9
Baud rate 7-26
Bitmap Selector box 9-4, 9-14, 9-17
Bitmaps
256-color 2-51
adding to layout 2-50–2-51
behavior in different display modes A-1
editing 9-10, 9-17–9-23, 9-25
effect on application size A-5
stretching A-5
blink function 4-2
break event 4-32–4-33, 4-35
Break Group command 2-34
Breakpoints
(un)setting 15-7
brief definition 15-7
display 15-7
remove all 15-7

Call function 7-19
Cancel Adding command 2-16, 2-19

Cancel Move command 11-9
CBT channel 20-15
Center command, State Chart 17-10
Center command, State Chart 17-10
Center commands, Object Layout 2-38
  Horizontal 2-38
  Vertical 2-39
Center parameter 2-22
Centering commands, Object Editor
  Horizontal 9-13
  Vertical 9-13
  when creating/editing a pointer 9-28
Centering objects 2-36–2-39
centerPosition property 13-4
character function 13-37
Child mode 10-3
Child object 2-11
Choosing object colors
  in 256 color mode 2-45
  in TrueColor mode 2-46
Circle objects 3-2
  center parameter 2-22
  radius parameter 2-23
  size 2-18
Circle tool 9-18
Clashing object names 2-30
Clear Bitmap command 9-22
click function 13-53
Climb command 17-12–17-13
Color
  background 2-43, 4-48, 9-15
  changing in runtime 13-7
  customizing in the Object Editor 9-15
  fill 2-43, 4-48
  foreground 9-15
  indication 2-43, 4-48
  line 2-43, 4-48
  selecting in the Object Editor 9-15
  shading 2-43, 4-48
  transparent 2-45
Color palettes A-1
Colors command 2-44
  label object 3-5
  text display object 4-42
Command line options 21-12–21-14
Comments
  adding to logic 12-36
Commlink object 7-25–7-37
  (dis)connecting 7-29
  baud rate 7-26
  brief definition 7-25
  data bits 7-26
  delimiter 7-26, 7-30, 7-32
  error codes 7-32
  events 7-33
  flow control 7-26
  parity 7-26
  port 7-26
  receiving data 7-30
  sending data 7-30
  stop bits 7-26
  testing serial link 7-27–7-29
  timeout 7-26, 7-33
Communication objects
  Commlink 7-25– 7-37
  DDE client 7-3–7-6
  DDE server 7-6–7-7
  DLL 7-15–7-23
Compound triggers
  SEE Triggers, compound
Concurrent modes 10-4
  general rules 10-5
  in State Chart 17-3
  SEE ALSO Modes
  transitions to 10-8, 16-16
Conditions 10-11
  building 12-20–12-24
  building condition triggers 12-23
  compared to events 13-54–13-56
  multi-line 18-9
  multiple 12-23
  negative 12-23
  on modes 10-11
  syntax 12-21
  to test coordinates 13-6
  user-defined 18-2
Constant position 4-32
Continous Adding command 2-19
Continously adding graphic objects 2-19
Coordinates
  assigning values to position and size 13-4
  moving objects using 2-28
  testing position and size using conditions 13-6
  working with 13-4
Copy command
  Mode Tree 11-8
  Object Editor 9-21
  Object Layout 2-30
Copy row/column commands 9-23

Copying
graphic objects 2-30
lines of logic 12-32
modes and subtrees 11-8
nongraphic objects 5-4
rows/columns of a bitmap 9-23
sections of a bitmap 9-21
Create Group command 2-33
Creating
groups 2-33
new objects 9-3
Cross switch object 4-35
Cue 20-8
Cursor object 13-39–13-45
assigning cursors 13-41
display in Logic Palette 13-42
logic 13-39–13-45
priorities during runtime 13-43
reset to default 13-42
Cursor styles, text display object 4-44
cursorEntered/Exited events 13-40
Customizing
colors in the Object Editor 9-15
dial scales 4-3
linear indicator scales 4-8
rotary potentiometer scales 4-26
Cut command
Mode Tree 11-7
Object Editor 9-21
Object Layout 2-29
Cutting
graphic objects 2-29
lines of logic 12-31
modes and subtrees 11-7
sections of a bitmap 9-21

Data bits 7-26
Data objects 5-6–5-19
array 5-11–5-14
data store 5-15–5-19
holder 5-10, 13-29
integer 5-8–5-9
number 5-6–5-7
point 5-20
random number 5-9
string 5-6
Data Store object 5-15–5-19
addressing an index 13-18
assigning values to elements 13-19
assigning values to records, fields, elements 13-19, 13-21
conditions, in 13-25
data store file, editing 13-23
field 13-17, 13-21
loading from a file 13-21, 13-23
logic 13-16–13-25
overwriting one data store with another 13-19
overwriting one record with another 13-19
queries and searches 13-27
record 13-17
removing records 13-24
saving to a file 13-21, 13-23
Data type
DDE client object 7-5
DDE server object 7-7
dataReady event 7-5
Date object 5-22
logic 13-32
DDE 7-2–7-3
DDE Client object 7-3–7-6
link type 7-5
logic 7-5–7-6
DDE Server object 7-6–7-7
logic 7-7
Debugger 15-1–15-10
(un)setting breakpoints 15-7
application flow display 15-4
breakpoint display 15-7
breakpoint, running to 15-3
breakpoints, removing all 15-7
brief description 1-6
opening 15-2
overview 15-2
status messages 15-5
step into 15-3
step over 15-3
stepping mode 15-3
stopping 15-4
synchronizing with Logic Editor 15-6
Deep history transitions
SEE Transitions, deep history
Default mode 10-7
example of subtree with no default mode 16-5–16-7
making a mode the default 11-5
removing default designation 11-6
Default state 20-18
Default State command 20-18

Default transition
SEE Transitions, default
Degrees and radians 13-15
degreesToRadians function 13-15
Delete command
Logic Editor 12-35
Object Layout 2-32
Recorder 20-9
Delete row/column commands 9-25
deleteFrom: function 13-24
deleteFrom:to: function 13-24
Deleting
graphic objects 2-32
lines of logic 12-35
menu items 4-49
modes 11-11
nongraphic objects 5-5
recording file 20-9
records from a data store 13-24
rows/columns from a bitmap 9-25
switch positions 4-38
Delimiter 7-26, 7-30–7-32
Deselecting objects 2-25
destination window 6-15
defining size 6-16
Destinations 11-13–11-14
column in Logic Editor 12-5
creating in Logic Editor 12-14
creating in State Chart 17-18
defined 10-6
device element 6-3
Dial objects 4-3–4-5
disable function 13-2
Display Logic Info command 17-5
Display objects 4-39, 4-41–4-46
text display 4-39–4-44
text window 4-44–4-46
Display Transitions command 17-8
displayMessage: function 7-50
Distribute commands 2-39
Horizontal 2-40
Vertical 2-40
Distributing objects 2-36, 2-39–2-40
Dive command 17-11
DLL object 7-15–7-23
call 7-19
examples 7-20–7-23
parameters 7-17
structures 7-18
DLLs, writing for Rapid C-1
Drag ‘n drop objects 2-10
active objects, making drag ‘n drop 2-20
groups 2-35
in user-defined objects 19-4
primitive objects, making drag ‘n drop 3-11
recording movements 20-13
within a group 2-33
Dragging
lines of logic 12-33
Duplicating
graphic objects 2-32
lines of logic 12-34
modes and subtrees 11-8
Dynamic Data Exchange
SEE DDE
Dynamic objects 2-9
active objects, making dynamic 2-20
groups 2-35
in user-defined objects 19-4
logic 13-4–13-8
primitive objects, making dynamic 3-11

Edit Active Areas command 9-26
Edit Bitmaps command 9-17
Edit command
Group, Object Layout 2-34
Reports 24-12–24-13
Edit line, Logic Editor 12-4
Edit Object command 2-47, 9-6
Edit Pointer command 9-28
Editing
active areas 9-26
application objects 9-5
array elements 5-13
bitmaps of objects 9-17
data store elements 5-19
data store fields 5-18
data store file 13-23
library objects 9-7
logic 12-13
mode names 11-4
nongraphic objects 5-4
object colors 2-43
objects in a group 2-34
objects in the Object Editor 2-47
pointers 9-28
recording file 20-10
root mode’s name 11-5
transition destinations 11-14
user-defined functions 18-9

Ellipse objects 3-2
position 2-18, 2-22
size 2-18
size parameter 2-23
enable function 13-2
enableExternalEvents function 7-47
End Edit [Group] command 2-34
End parameter 2-23
Entry activities 10-13, 12-27
compared to actions 10-16
occur when application starts 10-13
Error codes 7-32
Event flow 7-40
Event object 5-24
examples of uses 13-33
in recorder files 20-12
logic 13-32–13-35
Event property 20-12
Event symbol 20-12
eventObject property 7-44, 7-47
eventProperty property 7-44, 7-47
Events 10-11
adding to user-defined objects 19-9
building event triggers 12-17–12-20
compared to conditions 13-54–13-56
logic 16-9
multiple 12-20
syntax 12-18
user 16-9
eventSymbol property 7-44, 7-47
Exclusive modes 10-4
in State Chart 17-3
transitions to 16-9–16-15
Exit activities 10-13, 12-27
occur when application stopped 10-13
Exit Rapid command 1-5, 1-10
Exporting
user-defined functions 18-10
External program
recording events 20-14

Field
SEE Data Store object, field
File menu 1-3
Fill color element 2-43
menu object 4-48
Fill tool 9-20
fillColor property 13-7
dynamic primitive object 9-7
Filled circle tool 9-18
Filled frame tool 9-19
Filled polygon tool 9-20
Find and Replace
accessing 11-15, 12-11
deleting lines from search list 14-9
going to a logic line 14-9
opening 14-2
references to objects 14-1
replacing 14-6, 14-8
replacing, undo 14-8
replacing, validity check 14-7
search options 14-3
search results 14-4
search, defining scope 14-3
search, defining string 14-3
search, interrupting 14-9
searching for logic types 14-8
searching for object/property 14-9
wild cards in search string 14-4
Finding
mode names 11-15
objects in Logic Palette 12-11
Fit command 17-17
Flow control 7-26
Flow objects 4-52–4-56
pipe 4-52–4-55
valves 4-56
Flow pattern 4-54–4-55
Flow speed 4-54
flowOff function 13-60
flowOn function 13-60
follow execution
SEE Debugger, synchronizing with Logic Editor
Font command
Application Manager 1-7
Fonts
adding to packaged application 22-9
changing Rapid's font 1-7
dial scales 4-5
label object 3-5
linear indicator scales 4-10
text display 4-41
text window object 4-46
Foreground color 9-15
Frame command 17-16
Frame objects 3-2
Frame tool 9-18
Function Editor 18-7
accepting a function in 18-8
building a function in 18-8

SEE ALSO Logic Editor
Functions 2-8, 10-13
adding to user-defined objects 19-10
arguments 10-14, 12-22
binary 12-27
SEE ALSO user-defined functions

Generate Runtime command 1-5
Generated objects
deleting 13-30
events of 13-31
generated by user-defined objects 19-25
Generating
arguments 18-14
objects 13-30
random numbers 13-15
runtime applications 1-5
getCharacter function 13-37
Grid
Object Editor 9-12, 9-14
Object Layout 2-13–2-14
Grid command 2-13
Grouping objects 2-33
Groups
adding objects to existing group 2-34
creating 2-33
drag 'n drop 2-35
editing objects within 2-34
removing objects from 2-35
resizing 2-29
ungrouping 2-34

Help
getting Help in Rapid 1-xxi
user-defined objects, for 19-13
Help ID numbers
assigning to user-defined functions 18-12
Hide command
Mode Tree 11-6
Object Layout 2-42
Hiding
modes in Mode Tree 11-6
Object Palette 2-13
objects on the layout 2-42
Highlight Transitions command 17-9
History transitions
SEE Transitions, history
hold: function
Holder and array of objects 13-29
Pipe object 13-61
holdCopyOf: function 13-30
Holder object 5-10, 13-29
as argument in user-defined functions 19-26
deleting generated objects 13-30
events of 13-31
in user-defined objects 19-26
logic 13-29
Hot link 7-5
Hot spots 4-31, 20-3, 21-8–21-10
HSB color 2-47

Import Bitmap command 2-50
Importing
bitmaps 2-50
Indication color element 2-43
menu object 4-48
Indicator objects 4-3–4-23
bar indicators 4-11
barrel dials 4-12–4-16
dials 4-3
editing pointer of 9-28
linear indicators 4-7
sector dial 4-22, 4-24
user pointers 4-17–4-21
Insert (text entry) mode 4-43
Insert command 11-9
Insert row/column commands 9-25
Inserting
lines of logic 12-34
modes 11-9
rows/columns into a bitmap 9-25
Inspector 15-8
instancesOf functions 13-27
Integer object 5-8–5-9
as a data store field 5-15
as a property 3-9
formatting as a string 13-13
logic 13-13–13-15
random integer, generating a 13-16
Internal actions 10-16
Internal paste buffer 2-29
Internal transitions 10-10
compared to transition from mode to itself 10-10
conflicting 16-11
continuously true 16-15
creating 12-16
Inverse [Grid] command 9-12
is connected function 4-32, 13-56
is controlKey function 13-39

is formattedAs: function 13-11–13-12
is open function
DDE client 7-5
is overlapping: function 13-2
is placedOn: function 13-3
is shiftKey function 13-39
Item parameter
CBT channel 20-16
DDE 7-3
Simulation Manager object 7-39

JavaScript
adding plug-in control buttons 23-6–23-7
Jogdial object 4-27–4-31
click function 13-53
defining fast rotation 13-53
fast rotation 4-28
hot spots 4-31
initial angle 4-29
logic 13-51–13-53
pulse resolution 4-29
rotating 13-52
scale definitions 4-30
setting value dynamically 13-54
Joystick object 4-37–4-38
logic 13-57–13-58
moving a bitmap (example) 13-57

Keyboard object 5-28
entering text with 13-37
for entering text into text display 4-40
for entering text into text window 4-45
logic 13-36–13-39
principles of operation 13-36
user-defined object and 19-27
keyPress event 13-38

Label object 3-4–3-5
adding to layout 3-4
editing 3-5
Lamp objects 2-8, 4-2
lastEventIndex function 13-31
Layout frame
resizing 2-13
Levels
SEE State Chart
Librarian 8-2
opening 8-2
selecting library object from 9-8
Libraries
creating 8-2–8-3
managing objects 8-4
opening 8-3
saving objects to 9-5
Library objects 8-4–8-5
adding to application 8-6
creating 9-3
editing 9-5, 9-7, 9-9
Line color element 2-43
menu object 4-48
Line objects 3-3
adding to layout 3-3
dynamic 13-8
end parameter 2-23
logic 13-8
start parameter 2-22
Line tool 9-18
Line width 3-6–3-7
marker 3-6
no affect on Label objects 3-4
setting 3-6
Line width box 3-6–3-7
Line Width command
Object Editor 9-12
Object Layout 3-6–3-7
Linear indicator objects 4-7–4-10
lineColor property 13-7
dynamic primitive object 9-7
Link type 7-5
Load Backup command 1-4
loadFromFile: function 13-21
Loading
data store from a file 13-23
Logic
adding comments to 12-36
advanced topics 16-1–16-18
SEE ALSO Debugger
debugging 15-1–15-10
SEE ALSO Debugger
finding and replacing 14-1
going to a line from Find and Replace 14-9
Rapid logic basics 10-1–10-15
removing temporarily 12-37
searching by logic type(s) 14-8
stepping through with Debugger 15-3
syntax, summary 16-2–16-4
tracing in the Prototyper 20-4
typing manually 12-35, 12-37
validity check when replacing 14-7

verify 12-42
Logic Editor 12-2–12-37
actions 12-27–12-30
actions column 12-5
activities 12-27–12-30
activities column 12-6
brief definition 1-6
changing a column's width 12-14
conditions 12-20–12-24
copying lines 12-32
cutting lines 12-31
deleting lines 12-35
description 12-2–12-7
destinations column 12-5
display during debugging 15-6
dragging lines 12-33
duplicating lines 12-34
edit line 12-4
editing logic 12-13
events 12-17–12-20
external transitions, creating 12-14
guidelines on cutting/copying/pasting 12-33
inserting lines 12-34
inspecting object contents during runtime 15-9
internal transitions 12-16
lines of logic 12-6
Logic Palette 12-7–12-8
mode list 12-3
moving lines 12-33
navigating in 12-14
opening 12-2
pasting activities by type 12-32
pasting lines 12-32
selecting lines 12-12
synchronizing with Debugger 15-6
transition destinations, creating 12-14
transition type 12-15–12-16
triggers 12-17–12-27
triggers column 12-5
typing logic 12-35, 12-37
zooming in 12-3
Logic Information pane 17-5
Logic Palette 12-7
description 12-8–12-10
inspecting object contents during runtime 15-8
logical operators (not, and, or) 12-10
navigating in 12-9
object list 12-8
opening 12-8
sorting the objects 12-8
Logical operators (not, and, or) 12-10, 12-23

Make Default command 11-5
make event 4-32–4-33, 4-35, 13-55
Marquee tool 9-20
MCI 6-2
Menu object 4-47
Message object 5-26–5-28
Middle graduations 4-5, 4-9
Mnemonic, menu object 4-47
Mode activities 10-13
compared to entry & exit activities 12-27
effect of pausing an application on 12-28
timing of 10-15
when they occur 10-15
Mode activities 10-14–10-15
Mode Browser
temporarily removing logic 12-37
Mode list, Logic Editor 12-3
Mode Tree 11-2–11-12
adding a new mode 11-3
adding breakpoints 15-7
brief definition 1-5
description 11-2
display during debugging 15-6
editing mode names 11-4
finding mode names 11-15
logic verification test 12-38–12-42
printing 24-8
selecting a mode 11-2
Modes
adding a new mode 11-3
chart 10-3
child 10-3
concurrent 10-4
SEE ALSO Concurrent modes; Exclusive modes
copying 11-8
cutting 11-7
deleting 11-11
description of 10-2–10-5
duplicating with mouse 11-8
editing names of 11-4
exclusive 10-4
finding in tree 11-15
hiding in Mode Tree 11-6
hierarchy 10-3
in conditions 10-11
in State Chart 17-3
in triggers 12-24

Modes *(cont.)*
inserting 11-9
levels in State Chart 17-7
logic warnings 12-41
moving 11-9
names in State Chart 17-6
pasting 11-8
removing 11-11–11-12
root 10-3
selecting in the State Chart 17-17
showing in Mode Tree 11-6
sibling 10-3
tree 10-3
types of 10-4
Momentary position 4-32
Momentary pushbuttons 4-32
Monitor mode 7-42
Mouse coordinates
showing 2-14
Mouse object 5-29, 13-39–13-45
activating 13-40
auto repeat 13-45
cursorEntered/Exited events 13-40
logic 13-39–13-45
mouseMove event 13-41
position property 13-44
mouseMove event 13-41
Move command 11-9
Move to Back command 2-41
Move to Front command 2-41
when moving obscured objects 2-25
Moving
lines of logic 12-33
marked area in Object Editor 9-20
modes and subtrees 11-9
objects 2-27–2-28
objects to back 2-41
objects to front 2-41
Moving scales 4-5, 4-9
Multi-line conditions 18-3, 18-9
Multimedia A-6
Multimedia objects 6-1–6-22
adding 6-4
auto open 6-7
closing 6-22
defining properties 6-5
destination window 6-15
device element 6-3
function summary 6-10
getting image source size 6-17
getting information 6-20
listing 6-3
MCI 6-2
memory limitations 6-7, 6-22
notify parameter 6-20
opening 6-12
pausing 6-22
playing 6-19
playing 6-15
presenting image with border 6-18
runtime errors 6-12
stopping 6-22
syntax 6-8, 6-10
time formats 6-19
wait parameter 6-21
window style 6-13
Multiple triggers
SEE Triggers

Name clashes
when cutting objects 2-30
when pasting objects 2-31
Name parameter 2-21
Naming
active and nongraphic objects 2-11
active objects 2-21
primitive objects 3-8
user-defined functions 18-17
New command 1-3
New Mode command 11-3
New Object command 9-3
next function 13-15
Nongraphic icon 5-3
Nongraphic objects 5-1–5-29
adding to layout 5-2
array 5-11–5-13
ASCII 5-28
copying 5-4
cursor 13-39–13-45
data store 5-15–5-19
date 5-22
DDE client 7-3–7-6
DDE server 7-6–7-7
defined 2-2
deleting 5-5
DLL 7-15–7-23
editing 5-4
event 5-24
holder 5-10, 13-29
inspecting contents during runtime 15-8

integer 5-8–5-9
keyboard 5-28
message 5-26–5-28
mouse 13-39–13-45
number 5-6–5-7
patterns 4-54
point 5-20
random number 5-9
renaming 5-4
replacing 5-5
simulation manager 7-38–7-51
sound 5-24
stopwatch 5-21
string 5-6
SystemDate 5-29
SystemTime 5-29
time 5-23
timer 5-21
Nongraphic Objects dialog box 5-3
Nongraphics command 5-3
Normal [Grid] command 9-12
Normal command 17-17
Normal mode 7-41
not logical operator 12-23
Notes
adding and viewing 2-49
on nongraphic objects 5-2–5-3
on objects 2-49
notify parameter 6-20
Number object 5-6–5-7
as a data store field 5-15
as a property 3-9
formatting as a string 13-13
logic 13-13–13-15

Object Editor 9-1–9-29
brief definition 1-6
creating a new object 9-3
opening 9-2
testing the objects 9-29
toolbar 9-3
Object Layout 2-1–2-52
brief definition 1-5
customizing 2-12
grid 2-13
opening 2-12
printing 24-2
refreshing the layout 2-49
Object Palette
buttons of user-defined objects 19-11
hiding and showing 2-13
selecting objects from 2-15
Objects
aligning 2-36–2-37
bring to front 2-25
centering 2-38–2-39
class and object icons 2-16
classes 2-8, 2-15
coloring 2-44
copying 2-30
cutting 2-29–2-30
default name 2-21
defined 2-2
deleting 2-32
deselecting 2-25
deselecting object class 2-16
deselecting object type 2-16
disabling 13-2
distributing 2-39–2-40
duplicating 2-32
editing in the Object Editor 2-47
finding in Logic Palette 12-11
finding references in logic 14-1
generated by user-defined objects 19-25
grouping 2-33
hiding on layout 2-42
hot spots 20-3, 21-8, 21-10
illegal names 2-21
inspecting contents during runtime 15-8
library 8-4–8-5
modifying parameters 2-20
moving in logic 13-4–13-6
moving on layout 2-27–2-28
moving to back 2-41
moving to front 2-41
name clashes 2-30
names 2-11, 2-21
obscured 2-25
overlapping 13-2–13-3
pasting 2-31
placing and resizing simultaneously 2-18
placing on the layout 2-18
replacing one with another 2-47–2-48
resizing 2-28–2-29
rules for naming 2-21
saving to library 8-5
selecting 2-24–2-26
selecting all active 2-26
selecting object and its children 2-26
selecting to add to layout 2-15

Objects *(cont.)*
showing on layout 2-42
sorting in the Logic Palette 12-8
testing 9-29
undoing changes 2-33
visibility of, in user-defined objects 19-25
Objects, active
SEE Active objects
Objects, drag 'n drop
SEE Drag 'n drop objects
Objects, dynamic
SEE Dynamic objects
Objects, multimedia
SEE Multimedia objects
Objects, nongraphic
SEE Nongraphic objects
Objects, primitive
SEE Primitive objects
Off [Grid] command 9-12
Open Application Object command 9-5
Open command 1-3
Open Library Object command 9-7
Opening
Debugger 15-2
Find and Replace 14-2
Logic Editor 12-2
Logic Palette 12-8
Object Editor 9-2
Object Layout 2-12
Prototyper 20-2
Rapid 1-2
Recorder 20-7
Reviewer 21-2
State Chart 17-2
Optimizing applications 16-17
Options menu
Application Manager 1-6–1-7, 1-9–1-10
or logical operator 12-23
Orphans 2-30, 2-32
Overlapping objects 13-2–13-3
Overwrite (text entry) mode 4-43

Packager 22-1–22-15
adding external files 22-7
adding fonts 22-9
adding readme file 22-8
editing a package 22-12
end-user instructions 22-14
installing 22-2
launching 22-3
overview 22-2
SEE ALSO Plug-in Packager
preparing applications 22-3
Reviewer file list 22-14
selecting an icon 22-6
selecting application 22-5
setup.ini file 22-13
specifying build directory 22-11
specifying default directory 22-10
specifying folder name 22-10
padSpacesLeftTo: function 13-9
padSpacesRightTo: function 13-9
.PAK report files 22-12
Palette
256-color 2-45, 2-51, 9-15, 21-10
basic 2-44, 2-51
basic, Object Editor 9-15
problems in Prototyper 20-4
replacing Rapid's palette 2-51, A-4
Palette Update command 20-4
Parameter pane 2-20
Parameters, DLL 7-17
Parent application
definition 19-2
Parent object 2-11
to label objects 3-5
Parent parameter 2-11, 2-22
parity 7-26
Pass by reference 7-18
Pass by value 7-18
Paste Bitmap command 2-50
Paste buffer 2-29
Paste command
Mode Tree 11-8
Object Editor 9-21
Object Layout 2-29, 2-31
Pasting
bitmaps 2-50
graphic objects 2-31
lines of logic 12-32
modes and subtrees 11-8
sections of a bitmap 9-21
Pattern objects 4-54
pattern property 13-61
Pause command 20-5
Pausing
applications 20-5
effect on applications 12-28
in Debugger 15-3

Pipe object 4-52–4-55
adding to layout 4-52–4-53
creating a T joint 4-53
flow speed 4-54
flow style, changing 13-61
logic 13-60–13-61
pattern 13-61
speed of flow 13-61
Pixel painter tool 9-18
Placing graphic object on layout 2-18
Plug-in Packager 23-1–23-8
adding external files 23-3
adding fonts 23-3
launching 23-2
overview 23-2
preparing applications 23-2
selecting application 23-2
specifying name and location 23-3
.zrp files 23-1
Plug-in, Netscape 23-1–23-8
application window modes 23-4
control buttons 23-6–23-7
embedding an application in standalone mode 23-5
embedding an application in the HTML page 23-4
overview 23-8
providing pointer to plug-in 23-5
putting application on Web server 23-8
PLUGINSPAGE 23-5
Point object 5-20
as a property 3-9
Pointer object 4-17–4-18
Pointer objects 4-20–4-21
offset 4-20
style 4-18
width 4-19
Pointers
editing 9-10, 9-29
editing mode 9-28
linear indicator objects 4-7
Polygon tool 9-19
Port, serial 7-26
Position parameter 2-22
position property 13-4
Potentiometer objects 4-25–4-29
editing pointer of 9-28
jogdial 4-27–4-29, 4-31
recording changes in value 20-13
rotary 4-25–4-26
sliders 4-26
Primitive objects 3-1–3-11
adding to layout 2-14–2-15, 2-18
circles 3-2
defined 2-2
dynamic 9-7
ellipses 3-3
frames 3-2
imported bitmaps 2-50
lines 3-3
making active 3-8–3-9, 3-11
making drag ‘n drop 3-11
making dynamic 3-11
naming 3-8
placing in front of other objects 2-41
position parameter 2-22
selecting 2-24
size parameter 2-23
Primitive objects, line width
SEE Line width
Primitive objects, properties
SEE Properties, adding to primitive objects
Printing
mode data 24-10
Mode Tree 24-8
object data 24-4
Object Layout 24-2
object tree 24-3
Prototyper layout 24-6
State Chart 24-7
Propagate Changes command 9-14, 9-21
used when restoring bitmap/active areas 9-17
Properties 2-8, 10-13
adding to label object 3-5
adding to primitive objects 3-8–3-10
adding to user-defined objects 19-7
Prototyper 20-1–20-21
brief definition 1-6
CBT channel 20-15
inspecting object contents 15-8
pausing the application 20-5
printing 24-6
recorder 20-6–20-13
running the application 20-2
state 20-17, 20-19, 20-21
stopping the application 20-6
student mode 20-16
Trace command 20-4
pullFrom:to: function 13-24
Push (text entry) mode 4-43

Pushbutton objects 4-31
active area 9-26
recording events 20-13

Radians and degrees 13-15
Radius parameter 2-23
radius property 13-4
Random number object 5-9
formatting as a string 13-13
logic 13-15
RAPIDxx.INI file 1-9–1-10
.RCD files
SEE Recorder
Record
SEE Data Store object, record
Record index 13-17
Record name 13-17
Recorder 20-6–20-13
cue 20-8
editing files 20-10
in Reviewer 21-11
playing recorder files 20-9
playing the recording 20-7
recording 20-7
saving the recording 20-8
step 20-8
recorderStatus property 7-44
Redrawing
current bitmap in the Object Editor 9-22
the layout 2-49
Reducing file size 1-4
Refresh Display command 2-49, 9-22
Registering
user-defined objects 19-16
Regular expressions 13-11
Remarks
adding to logic 12-36
Remove command 11-11
Remove Default command 11-6
Remove Members [from Group] command 2-35
Removing
modes 11-11
objects from a group 2-35
records from a data store 13-24
Renaming
graphic objects 2-21
nongraphic objects 5-4
user-defined functions 18-10, 18-19
Replace Object command 2-48
Replacing
graphic objects 2-47–2-48
nongraphic objects 5-5
Rapid's 256-color palette 2-51
strings in logic 14-6
Reports 24-1–24-13
destinations 24-1
editing 24-12–24-13
file extensions 24-12
mode 24-7, 24-9–24-11
mode data 24-10
Mode Tree 24-8
object 24-2–24-6
object data 24-4
object layout 24-2
object tree 24-3
Prototyper layout 24-6
State Chart plot 24-7
requestData function 7-5
Reroute mode 7-43
Reserved names 2-22, 18-17
Resizing
graphic objects 2-28
layout frame 2-13
multiple graphic objects 2-29
text display object 4-42
text window object 4-46–4-47
Restore Bitmap/Active Areas/Pointer command 9-16
Resume command 15-4, 20-5
Returned (data) type 7-16
Reviewer 21-1–21-15
brief description 21-1
closing 21-3
command line options 21-12
control menu 21-4, 21-6
hot spots 21-8, 21-10
move command 21-7
opening 21-2
opening with specific application 21-3
palette 21-10
recording 21-11
state 21-11
stopping the application 21-11
views 21-4
RGB color 2-46
changing in runtime 13-7
Rocker switch object 4-33
Root mode 10-3, 11-2
editing name of 11-5

Root object 2-11
adding properties to 19-7
Rotary potentiometer object 4-25–4-26
Rotary switch object 4-34
RPX objects 19-16
.RST files
SEE State files
Runtime application version 1-5

Save As command 1-4
Save command 1-4
Save Settings on Exit command 1-9–1-10, 3-5
Save to Application command 9-8
Save to Library command 9-5
saveToFile: function 13-21
Saving
a recorded sequence 20-8
application under a new name 1-4
applications 1-4
data store to a file 13-21
edited objects 9-8
objects to libraries 9-5
system settings 1-10
Scales
Default scale 1-7
dial objects 4-3
linear indicator objects 4-8
moving 4-5, 4-9
rotary potentiometer object 4-26
rotation step 4-5
searchFor functions 13-28
Sector dial object 4-22, 4-24
creating circular dial 13-51
defining colors 4-24
logic 13-50–13-51
simulating radar 13-50
Select All Active command 2-26
Select Children command 2-26
Select from List command 2-26
Selecting
lines in Logic Editor 12-12
Selecting all active objects on layout 2-26
Selecting objects 2-15, 2-24–2-26
automatically selecting children 2-26
from the object list 2-26
multiple objects 2-24
obscured object 2-25
Selector button 2-16, 2-19
Selector routine 16-6
self property 12-19
sendCommand: function 7-5
Sequence of Rapid operations 16-7–16-17
Set Size command 9-11
when used with Restore Bitmaps command 9-16
setPosition function 7-50
setRed:green:blue: function 13-7
Setup command 20-15
Shading color element 2-43
menu object 4-48
Show command
Mode Tree 11-7
Show Mouse Coordinates command 2-14
Show Object Palette command 2-13
Show Objects command 2-42
Show Pointer command 9-25, 9-29
Showing
modes in Mode Tree 11-6
mouse coordinates 2-14
Object Palette 2-13
objects on the layout 2-42
pointer 9-25
Sibling object 2-11
Signal objects 5-24–5-28
event 5-24
message 5-26, 5-28
sound 5-24
Simulation
controlling response of 7-47
controlling window of 7-45
getting information about 7-46
sending events to 7-48
window size and position 7-50
Simulation Manager object 7-38–7-51
checking for specific events 7-49
controlling simulation's response 7-47
events, receiving from simulation 7-47
logic 7-44–7-50
sending messages to simulation 7-50
setting simulation window size 7-50
simulationStatus property 7-44
size property 13-4
Slider potentiometer object 4-26
Slider switch objects 4-35
Sound object 5-24
speed property 13-61
Start command 20-2
Start parameter 2-22
Starting application
and execution of entry activities 10-13

State 20-17–20-21
compressed 20-21
default 20-18
saving to a file 20-17
State Chart 17-1–17-19
brief definition 1-6
center 17-10
climb 17-12–17-13
dive 17-11
fit view 17-17
frame 17-16
highlighting transitions 17-9
levels 17-7–17-8
Logic Information pane 17-5
magnifying a section 17-16
mode names in 17-6–17-7
modes 17-3
normal view 17-17
printing 17-9, 24-7
selecting modes 17-18
selecting transitions 17-18
top 17-14
transitions 17-4
transitions, displaying 17-8
transitions, numbering 17-8
zooming in 17-16
zooming out 17-16
State files
creating 20-17
editing 20-21
format 20-18, 20-20
loading 20-18
SEE ALSO State
Status box
library name, Object Editor 9-2
mouse coordinates, Object Editor 2-14, 9-2
Status messages in Debugger 15-5
step into 15-3
step over 15-3
Step, in Recorder playback 20-8
Stepper switch object 4-36
active areas 9-26–9-27
recording events 20-13
stepping mode 15-3
Stop bits 7-26
Stop command 20-6
Stopping application
and execution of exit activities 10-13
Stopwatch object 5-21
String object 5-6
as a data store field 5-15
as a property 3-9
characters, addressing individual 13-10
logic 13-9–13-12
Structures 7-18
Student mode 7-43, 20-15–20-16
accept external events 7-42
monitor mode 7-42
normal mode 7-41
reroute mode 7-43
running the Reviewer in 21-12
Student Mode command 20-16
Student name 20-16
studentName property 7-44
Subtrees
copying 11-8
cutting 11-7
duplicating 11-8
moving 11-9
pasting 11-8
Switch objects 4-32–4-39
active areas 9-26
changing position name 4-33
constant position 4-32
cross 4-35
deleting switch positions 4-38
events and conditions 13-54–13-56
joystick 4-37
logic 13-54–13-56
momentary position 4-32
recording events 20-13
rocker 4-33
rotary 4-34
slider 4-35
stepper 4-36
toggle 2-8, 4-34
Syntax
arithmetic functions 16-4
language D-1–D-9
logic, summary 16-4
Syntax, logic
activities 10-13
arrays 13-26
conditions 12-21
data store elements 13-18
events 12-18
mode triggers 12-25
negative conditions 12-23
summary 16-2–16-3
System objects 5-28–5-29

ASCII 5-28
keyboard 5-28
mouse 5-29
SystemCursor 5-29
SystemDate 5-29
SystemTime 5-29
SystemCursor object 5-29
SystemDate object 5-29
SystemTime object 5-29

Tank objects 4-56
active area 9-26
logic 13-62–13-63
Temporary files 1-3
Test mode 9-10, 9-29
Testing
applications in the Prototyper 20-3
objects in the Object Editor 9-29
Text display object 4-39–4-44
cursor styles 4-44
making it transparent 4-42
text entry direction 4-43
text entry mode 4-43
Text window object 4-44–4-46
position in the Prototyper 2-41
window size 4-46
tick event 5-21
Time object 5-23
logic 13-31
Time objects 5-21–5-23
date 5-22
stopwatch 5-21
time 5-23
timer 5-21
Time stamp 20-12
Timeout 7-26, 7-33
Timer object 5-21
timeStamp property 7-45, 7-47
TMP files 1-3
Toggle pushbuttons 4-32
Toggle switch objects 2-8, 4-34
Tool tips 1-9
Tools menu 1-5
Top command 17-14
Topic parameter
CBT channel 20-16
DDE 7-3
Simulation Manager object 7-39
Transition type 10-7, 11-14
changing 12-16
defining 12-15
Transitions 10-6–10-10
conflicting 16-11
creating in Logic Editor 12-14
creating in the Mode Tree 11-13
deep history 10-9
default 10-7
definition 10-6
destinations 11-13–11-14
editing destinations 11-14
from a subtree of exclusive modes 16-15
highlighting in State Chart 17-9
history 10-8
in data reports 24-11
in State Chart 17-4, 17-8, 17-18
involving concurrent modes 16-16
logic warnings 12-42
numbering in State Chart 17-8
parent to child 16-6
to a subtree of exclusive modes 16-13–16-14
to concurrent modes 10-8, 16-16
to exclusive modes 16-9
type of 10-7
within a subtree 16-14
Transitions command 11-13
Transitions, internal
SEE Internal transitions
Transparent color element 2-45
.TRD report files 24-6
Triggers 10-11–10-12
building 12-17–12-27
column in Logic Editor 12-5
compound 10-12, 12-26
condition 12-23
conditions compared to events 13-54–13-55
defined 10-6
event 12-17
in data reports 24-11
multiple 10-12
with modes 12-24
Trigonometric functions, example 13-8
.TRL report files 24-12
.TRM report files 24-10
.TRO report files 24-4
True color bitmaps A-1
Typing logic 12-35, 12-37

Undo command
Find and Replace 14-8
Object Editor 9-11, 9-17
Object Layout 2-33
Ungrouping graphic objects 2-34
User-defined functions 18-1–18-20
arguments in 18-13
assigning Help ID numbers to 18-12
building 18-4
building in Logic Editor 18-8
definition 18-2
deleting 18-10
deleting objects involved in 18-6
editing 18-9
exporting 18-10
generating arguments 18-14
naming 18-17
renaming 18-10, 18-16, 18-19
User-defined objects 19-1–19-40
adding properties to 19-7
adding to an application 19-14
building 19-4
buttons in Object Palette 19-11
changing, after adding to applications 19-21
definition 19-2
drag ‘n drop 19-17
drag ‘n drop objects in 19-4
dynamic 19-17
dynamic objects in 19-4
events of, recording 19-29
events of, recording 20-14
events, adding to 19-9
including logic of, in parent application 19-19
keyboard and 19-27
nongraphic 19-6
opening 19-5
passing an object from 19-27
passing an object to 19-26
position of objects in 19-4, 19-23
properties, adding to 19-7
recording events of 19-29, 20-14
registering 19-16
removing logic of, from parent applications 19-19
runtime 19-6
saving 19-5
scale of 19-24
search path for 19-17
size of 19-4
tutorial 19-30
tutorial (end) 19-40
visibility of objects in 19-25
when to use 19-2

Valve objects 4-56
logic 13-62
Verification Test 12-38–12-41
logic warnings 12-41–12-42

Warm link 7-5
Wild cards 14-4
windowPosition property 7-45, 7-50
windowSize property 7-45, 7-50

Zoom In command
Object Editor 9-14
State Chart 17-16
Zoom Out command
Object Editor 9-14
State Chart 17-16