Neuron Data Elements Environment

Version 2.1

Getting Started

Contents

Chapter

1 The Elements Environment

About the Elements

The *Elements Environment* (EE) is a suite of cross-platform tools for quickly building complex, object-based applications for today's rapidly changing organizations.

This product overview gives you general information about the product and points you to other information you need for developing your Elements Environment applications. The chapters that follow provide overviews of these items and pointers to the documentation for them.

These are the Neuron Data Elements and supporting services:

- The *Open Interface Element* (OIE) provides a window-based environment for building graphical user interfaces (GUIs). This Element allows developers to build portable user interfaces that are independent of the windowing environment and operating system.
- The *Data Access Element* (DAE) provides the front-end to independent data sources including flat files, relational databases, and data in spreadsheets and other applications.
- The *Intelligent Rules Element* (IRE) lets you capture data models and business logic as visually accessible entities that dynamically reflect changes in the organizational environment.
- *OOScript* is a simple, high-performance, object-aware scripting language that lets you create robust applications using all of the Elements with seamless interoperability.
- The *Web Element* (WE) provides an embeddable World Wide Web browser and Web-link navigation for building business-critical applications for Intranets.
- The *Distributed Messaging Element* (DME) provides a mechanism that allows you to partition and to dynamically configure and reconfigure your application components across various platforms, operating systems, and networking environments.
- The *Elements Application Services* (EAS) are the common set of core services that ensure a unified development environment. Elements Application Services include low-level services such as file input/output (I/O) and internationalization, plus higher-level application-development services, such as datasource/views.

- Support for true object interoperability in the Elements Environment through the OOScript language.
- Support for *C/C++* interfaces that let you automatically generate C and C++ for application code and class definitions.

New Features and Enhancements in the Elements Environment 2.0

The Elements Environment 2.0 incorporates former Neuron Data development tools into a cohesive suite of integrated elements, including:

- The Open Interface Element (OIE)
- The Data Access Element (DAE)
- The Intelligent Rules Element (IRE)

The 2.0 release adds new features and enhancements of existing features to these Elements.

In addition, two new Elements have been added to the development suite:

- The Web Element (WE)
- The Distributed Messaging Element (DME)

Upgrading existing applications developed with older Neuron Data products to the Elements Environment 2.0 enables you to take advantage of the new features and enhancements. In addition, migrating to the Elements Environment 2.0 now will ease the transition to future releases of the Elements Environment.

New Features and Enhancements to the Open Interface Element

- Support for the native environment on Windows 3.11, Windows 95, and Windows NT, including:
  - native drag and drop
  - native menu manager
  - underlying native window widget
  - native Clipboard

  Note: The Open Interface 3.*x* used emulation of these GUI elements.

  These improvements result in:
  - better performance
  - less memory overhead
  - better desktop integration
- New widgets for all supported platforms, including:
  - treeview widget

  - notebook widget
  - toolbar widget
  - tooltips (context-sensitive Help) for all widgets
- Support for the native environment on the Macintosh, including:
  - native Clipboard
  - native balloon help
- Long filenames (255 characters)
- UNC pathnames
- Setup guidelines
- Registered icons
- Compliance with Windows 95

New Features for the Intelligent Rules Element (NEXPERT)

- Enhancements to tools that allow you to:
  - build knowledge bases / expert systems and logic rules within a graphically object-oriented environment
  - have “smart” access to data through an inference-driven engine
  - encapsulate data models with business logic
- Support for a context-switch API
- A new Edit API

New Features for the Data Access Element

- Concurrent support for multiple databases within an application
- An array fetch for the Oracle driver
- Linking of multiple views to a single datasource
- Dynamic updating of editing changes to a single view with other linked views

Unicode and Multibyte Support

- All of the Elements Environment products now support Unicode and native multibyte for Japanese and Korean. (Each locale is licensed separately.)

Improvements to the OOScript Language

- Quick access to object classes and scripts through improved browsing and navigation
- Support for object interoperability, which allows you to integrate OOScript modules with other modules through such industry standards as CORBA and OLE

- Support for callback procedures, functions, and arrays

Improvements to the Development Environment

- You can now regenerate C++ code from the Resource Browser.
- You can customize code generation and regeneration.
- Documentation is now available online through Adobe Acrobat with an indexing search engine.

Improvements to the C++ API

- Full C++ classes for all Elements (except the Data Messaging Element)
- Support for the Intelligent Rules Element C++ classes
- Support for C++ exception handling (`try/catch/throw`)

New Features and Enhancements in the Elements Environment 2.1

The Elements Environment 2.1 release adds additional components and improvements to existing components, including:

- Improvements to OOScript
- Improvements to datasource/views in the Data Access Element
- Integration with third-party application-development software
- Support for character-based interfaces

Improvements to OOScript

- Support in the Distributed Messaging Element for a publish-and-subscribe interface
- Support in the Distributed Messaging Element for routing of serialized string messages to complex data, including to all Open Interface datasources
- Custom routing protocols in the Distributed Messaging Element that let you optimize broadcast objects to support two-tier, three-tier, and *n*-tier client-server application architectures
- Object support for datasource/views in the diagrammer, browser, and treeview widgets
- New code-template editor that lets you create and save custom templates for application-specific scripts

Improvements to Datasource/Views in the Data Access Element

- A tree datasource for hierarchically arranged node objects such as directories and subdirectories

- A graph datasource for creating complex diagrams and graphs
- Shared view dependencies with the treeview, browser, browser overview, and diagrammer widgets

Integration with Third-Party Application-Development Software

The Elements Environment now supports integration with Intersolv PVCS Version Manager.

Note: This product does **not** come with the Elements Environment 2.1. You must purchase it directly from the manufacturer.

PVCS integration is licensed separately. If you are licensed to integrate PVCS, the **PVCS** option on the **File** menu is enabled. If you are not licensed to integrate PVCS, this option is disabled.

Installation

With the Elements Environment suite of tools, you receive two CD-ROMs, one for the software and one for the documentation. You will need to install the software and the Adobe Acrobat Reader, which allows you to read and print the documentation for each Element.

Your software kit includes an *Installation Guide* for your specific platform. This booklet contains all the information you need to install:

- The Elements Environment on your platform
- The platform-specific Adobe Acrobat Reader

For More Information about Installation

- For information regarding installation of the Sentinel software and hardware, refer to the readme.txt file in the sentinel directory of the Development and Deployment Kit CD-ROM.
- For platform-specific installation information, refer to the *Installation Guide* that came with your software and documentation CD-ROMs. This booklet also contains information about using the online documentation.

- For platform-specfic information regarding installing online documentation, refer to the files in the read_me directory on the Documentation Suite CD-ROM:
  - rdme_mac.txt
  - rdme_unx.txt
  - rdme_win.txt

Technical Support

Neuron Data Corporate Headquarters USA (country code 1):

Tel: 415-943-2700 or 800-876-4900

Fax: 415-943-2756

E-mail support: @neurondata.com

For information or technical support for the Adobe Acrobat Reader, call the following Adobe Acrobat Technical Support telephone number:

206-628-2757

The Adobe Acrobat platform-specific ReadMe files contain more information about Adobe Technical Support. If you have any questions about the Neuron Data Online Documentation, call Neuron Data Technical Support.

Note: Neuron Data does **not** provide technical support for the Adobe Acrobat Readers.

The NDDN Technical Support Web Page

NDDN, the Neuron Data Technical Support page on the World Wide Web, gives you the following benefits:

- A wide range of Technical Support code examples
- Listings of supported products and platforms
- Numerous technical application notes (TANs) that are application- and platform-specific
- Information on the latest product releases
- Status information about reported Technical Support cases

This is the URL for the NDDN Technical Support Web page:

```
https://nddn.neurondata.com
```

Chapter

2 The Open Interface Element

The *Open Interface Element* (OIE) lets you develop cross-platform applications with native graphical user interfaces (GUIs). Open Interface comprises:

- A Resource Editor that helps you manage your resources—such as windows, widgets, fonts, and other graphic elements
- A Window Editor that helps you design windows for your application
- A set of libraries containing standard GUI routines

About the Software-Development Process

You use the Open Interface features to create your interface, which consists of resources such as windows. You then save your resources in these binary and text files:

- .dat file
- .rc file

The resource database is stored in a binary file (.dat). In addition, this database is stored in a text file (.rc), which you can edit directly.

The development environment also produces a C or C++ code template for your application. You use this template to link the application code to the GUI.

After adding your own code to the template, you use one of the supported compilers to compile the application into object files. You then link these object files with the Open Interface libraries to create an executable file.

Open Interface generates a makefile to assist you in compiling and linking your application. When you execute your program, the resource-database (.dat) file is opened. Your application then has access to the resources this database contains.

About Resources

Open Interface uses the concept of *resources* to modularize many parts of the application code. For example, items such as windows, buttons, menus, fonts, colors, cursors, and icons are stored as different types of resources. The specific information about an item—such as its location, size, or label—is retrieved and then handled using a set of generic tools. Almost anything can be stored as a resource.

In object-oriented terms, resources store all the persistent information for a particular item. This information does not change during the execution of an application. For example, a text-edit widget has information associated with it, such as size, location, attributes, and initial values. This persistent information is stored in the resource .dat file. However, the information that a user enters into the text-edit field of the widget is not persistent and is not stored as a resource.

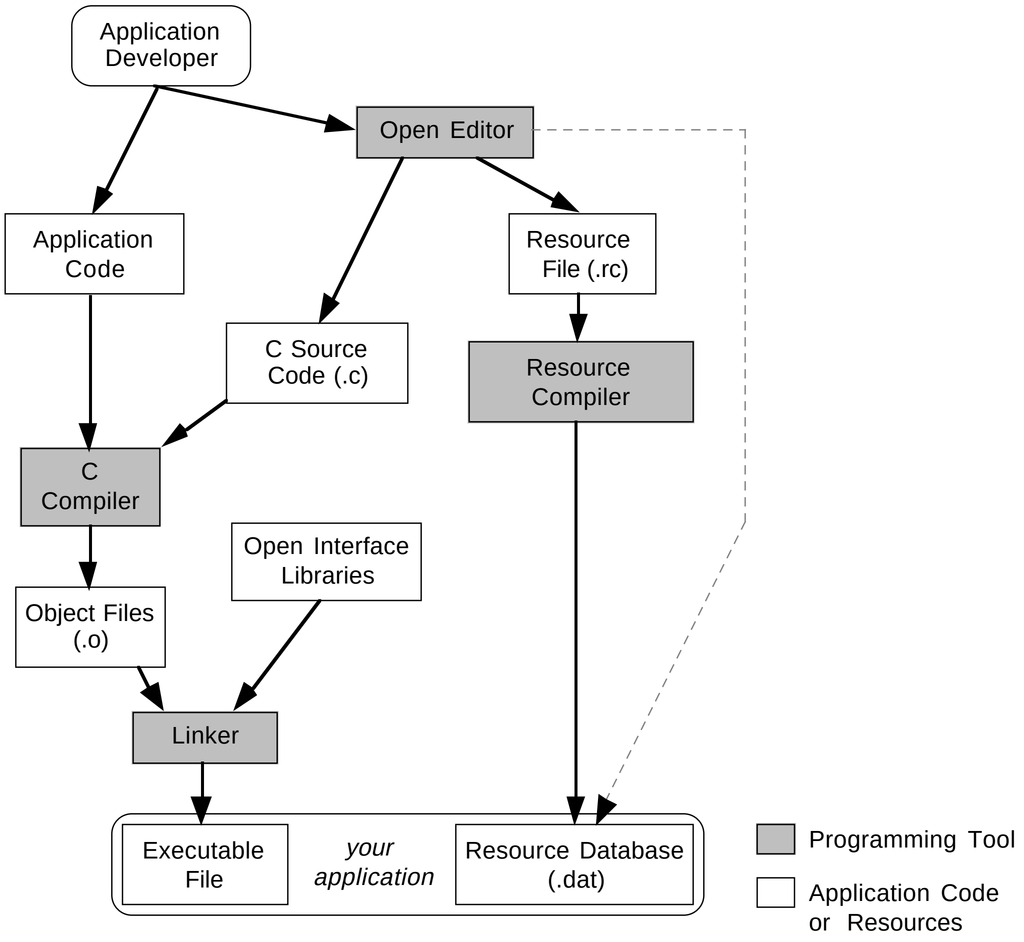


Figure 2-1 Open Interface Development Environment

Separating resources from the application code has several advantages:

- Using string resources allows you to prepare your products for various international markets without rewriting your code.
- Using a set of routines that are well defined, efficient, and generic greatly helps you in building resources.
- Your code assets can be more easily shared between different parts of the same application and between different applications.

Quick Tour

The flexibility of the Open Interface Element gives you a number of ways to begin creating your application. However, the introductory exercise that follows presents only a few of the possible approaches.

To complete the Quick Tour, you have to:

1. Create a user interface.
2. Create an application-startup module.
3. Code the application logic using OOScript.
4. Test and run the application.
5. Deploy and port the application.

This section describes each of these tasks. The instructions show you how to create an application with two buttons that display two different dialog messages.

Task1: Creating a User Interface

The Elements Environment's visual editors provide a graphical environment for creating and manipulating:

- Graphical user interface (GUI)
- Application logic
- Data access
- Business rules

These visual editors allow you to create GUIs with simple point-and-click operations. For detailed information about how to start and use the editors, see the *Open Interface Element User's Guide*.

Note: The illustrations that follow are based on Windows NT screens. However, the general tasks apply to all supported platforms.

To Design a New Window

1. Start the Elements Environment:

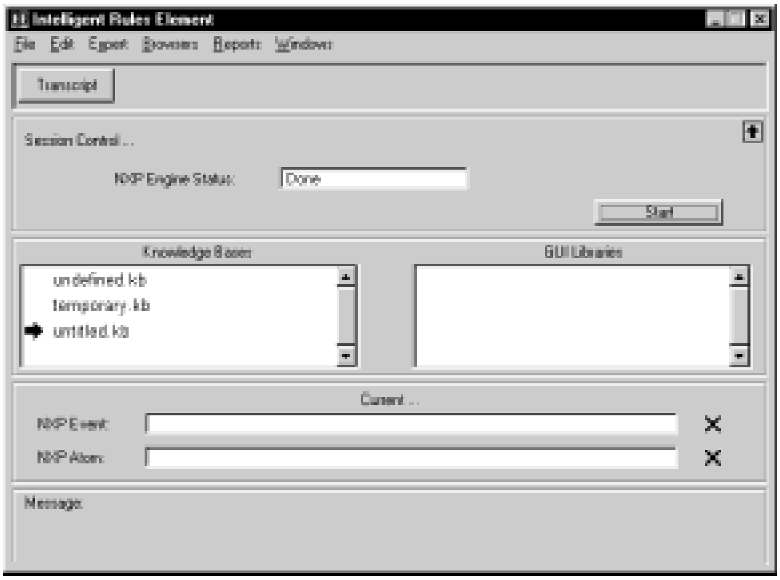
   – Click either the EE C or EE C++ icon.

   or:

   – Type `ee` at the command line prompt

   Note: Make sure that you are running the version of the Elements Environment for which you are registered.
   If you have installed both the C++ and C versions, the installer automatically registered you for C++ builds and interpretation. If you use the C version of the Elements Environment, but you are registered for C++, the C++ servers will not load properly. You will receive a “`Class Already Exists Error`” in your tracebacks.

   Refer to the *Elements Environment Installation Guide* for more information on how to register or unregister the servers.



NXP Engine Status: Done

Knowledge Bases:
undefined.kb
temporary.kb
➡ untitled.kb

Figure 2-2 Main Window for the Intelligent Rules Element (IRE)

Note: If your application is not licensed for the Intelligent Rules Element, you will see the Resource Browser instead of the Intelligent Rules Element main window and should proceed to step 3.

2. Choose **Browsers - Resource.**

   The Resource Browser window is blank because no library has been selected.

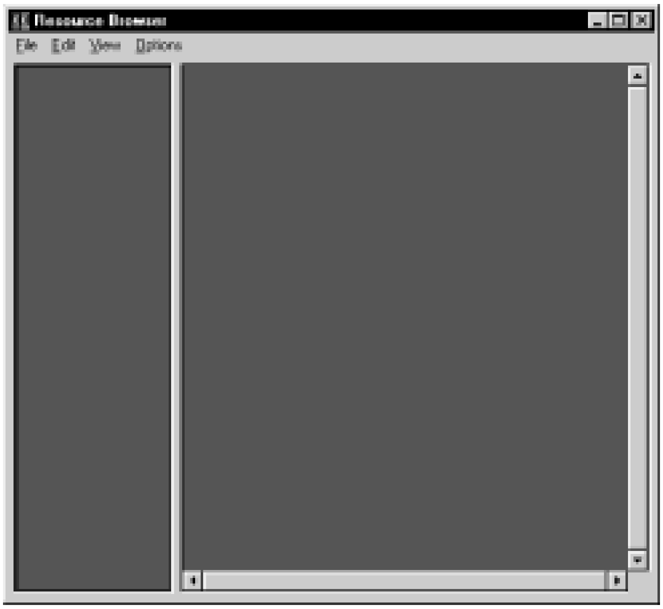


Figure 2-3 Resource Browser Window

3. Choose **File - New Application**.
4. Select the appropriate directory and name the application `Hello`.
5. Click **OK** to display the Window Editor.

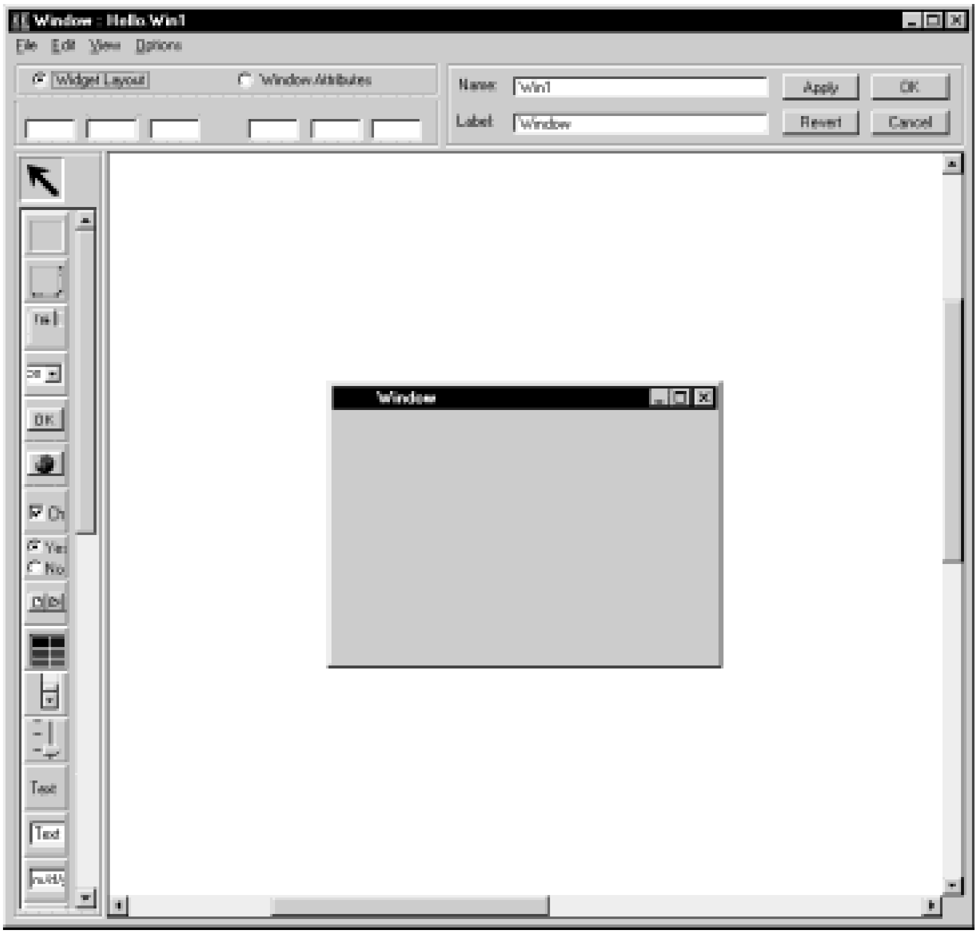


Figure 2-4 Window Editor in Small Tools Mode

About the Window Editor

The Window Editor opens in Small Tools mode by default when:

- the monitor has a small screen
- the monitor has low resolution

The Small Tools mode gives you a larger viewing area, but displays only minimal information and options. To switch to Large Tools mode, choose **Options - Large**.

In the Large Tools mode, the type of widget you select and put in a window appears in a box above the widget-icon column. This is helpful information when you are first using the product.

Note: This exercise uses Small Tools mode.

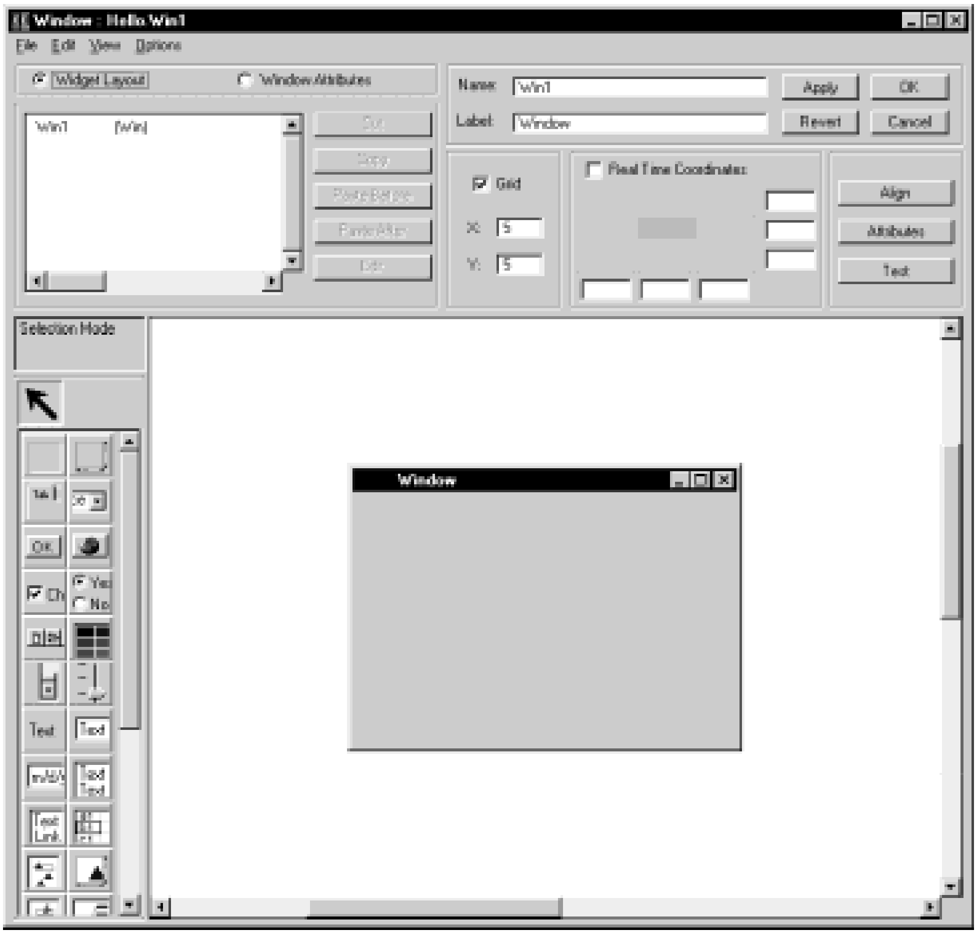
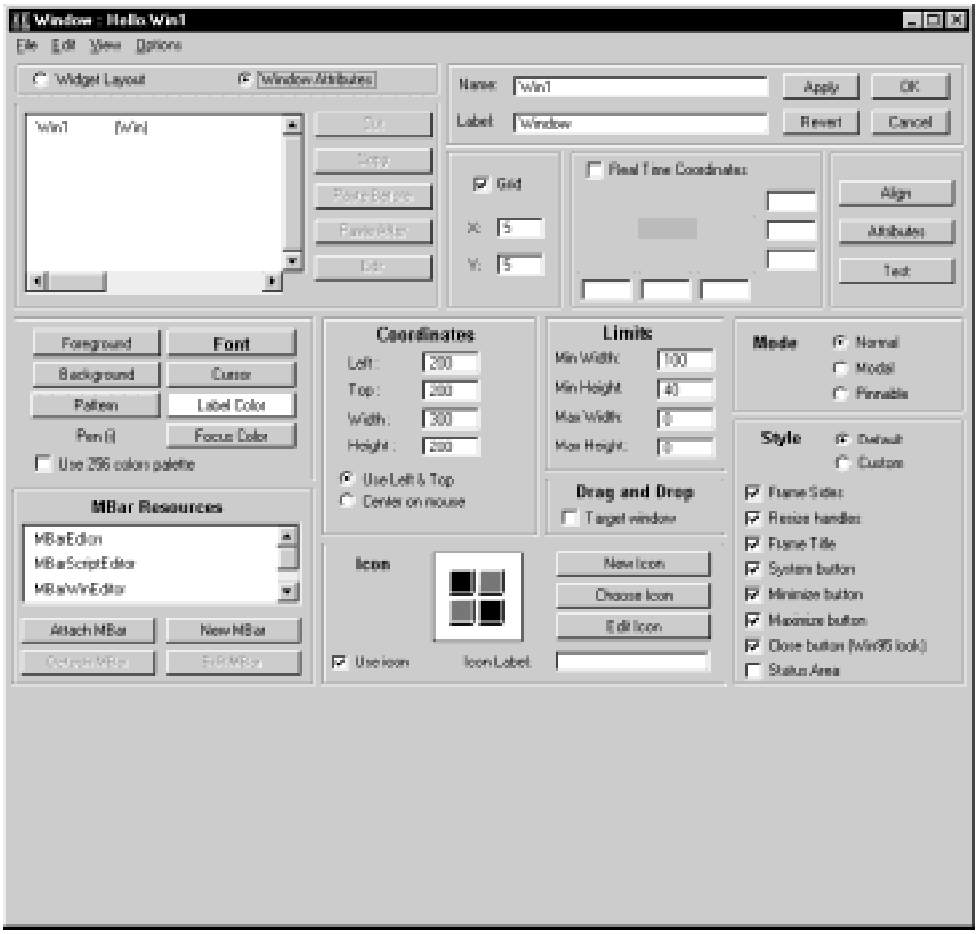


Figure 2-5 Window Editor in Large Tools Mode

The window you are designing is in the middle of the screen. You can resize this window by clicking inside the window and then dragging the corner handles. You can add widgets such as buttons to this window.

The vertical toolbar to the left of the window area contains icons for various types of widgets. You can scroll down and see other widgets to select from the toolbar. The large arrow at the top of this column is the Selection Tool icon. You click this icon to obtain the arrow cursor and select widgets. The two radio buttons in the upper left of the screen control the editing mode of the Window Editor. The default is **Widget Layout**. If you click **Window Attributes**, you can define the default attributes for your new window.



( ) Widget Layout [✓ Window Attributes]

Win1 [Win]

Name: Win1
Label: Window

[✓] Grid
X: 5
Y: 5

[ ] Real Time Coordinates

[ ] Use 256 colors palette

MBarEdition
MBarScriptEditor
MBarWinEditor

Coordinates
Left: 200
Top: 200
Width: 300
Height: 200
[✓ Use Left & Top] ( ) Center on mouse

Limits
Min Width: 100
Min Height: 40
Max Width: 0
Max Height: 0

Drag and Drop
[ ] Target window

Mode [✓ Normal] ( ) Modal ( ) Pinnable

Style [✓ Default] ( ) Custom
[✓] Frame Sides
[✓] Resize handles
[✓] Frame Title
[✓] System button
[✓] Minimize button
[✓] Maximize button
[✓] Close button (Win95 look)
[ ] Status Area

[✓] Use icon
Icon Label:

Figure 2-6 Window Attributes Dialog

In the Window Attributes dialog, the four text fields in the Coordinates panel show the window's placement on the screen (the *xy* origin and the *xy* extent).

In the two text-edit fields that appear in the upper right part of your window, you enter the name of the window module and the label.

These buttons are in the right corner of the screen:

- **Apply** implements changes you have made to the new window.
- **Revert** undoes any changes you have made with the last **Apply** and displays the previously saved state.
- **OK** saves your changes.
- **Cancel** undoes your changes.

To Place Widgets in Your Window

6. Select the **OK** pushbutton icon from the vertical toolbar, and draw the buttons on the window.
7. Arrange the buttons and resize the window as you wish.
8. Double-click the left pushbutton to open its property window.
9. Change the name **PBut1** to `PButGreeting`.
10. Change the label of **PBut1** to `Greeting.`
11. Click **Apply.**
12. Click **OK**.
13. Double-click the right pushbutton to open its property window.
14. Change the name of **PBut2** to `PButFarewell`.
15. Change the label of **PBut2** to `Farewell`.
16. Click **Apply**.
17. Click **OK**.

Note: Do not confuse **name** with **label**. The *name* is used to reference the widget in scripts and code. The *label* is what appears on the widget in the window.

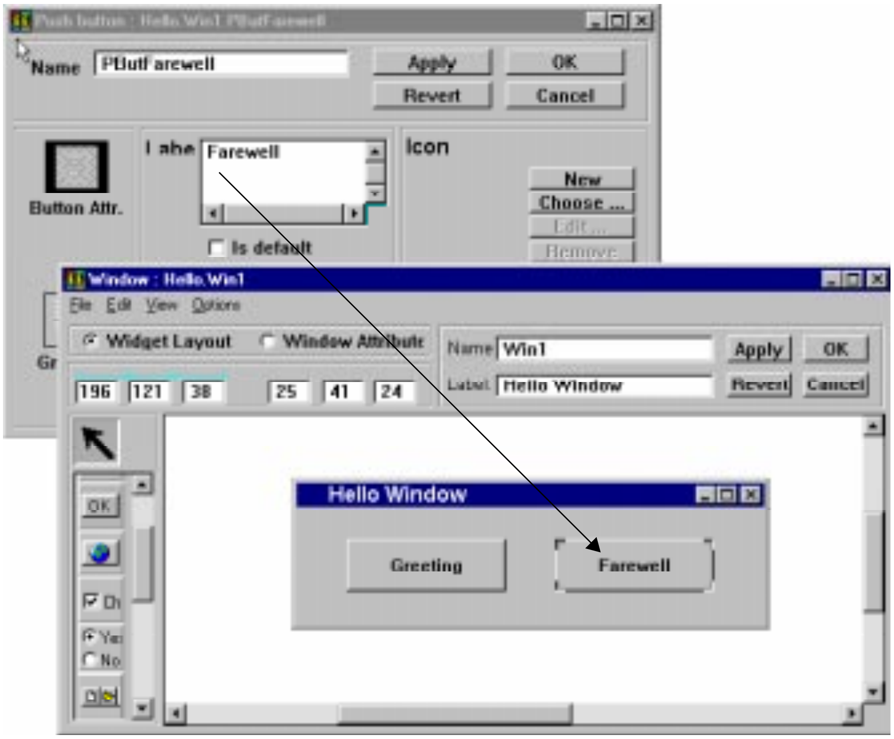


Figure 2-7 OK Pushbutton, Window, and Property Dialog

18. Choose **View - Test Mode**.

    This lets you test your design and layout.

19. Choose **File - Save**.
20. Choose **File - Close**.

About the Resource Browser Window

The Resource Browser window has two parts:

- Browser overview
- Browser

The panel on the left side of the window is the *browser overview*. This area allows you to rapidly scan through the modules and resources in the application. In this area are small iconic representations of the libraries, modules, and resources in your project. There is also a transparent rectangle. To move to another view in the *browser* (the panel to the right of the browser overview), you click and hold the mouse button within the rectangle, and then drag the rectangle to another position. The view within the browser shifts to the corresponding position.

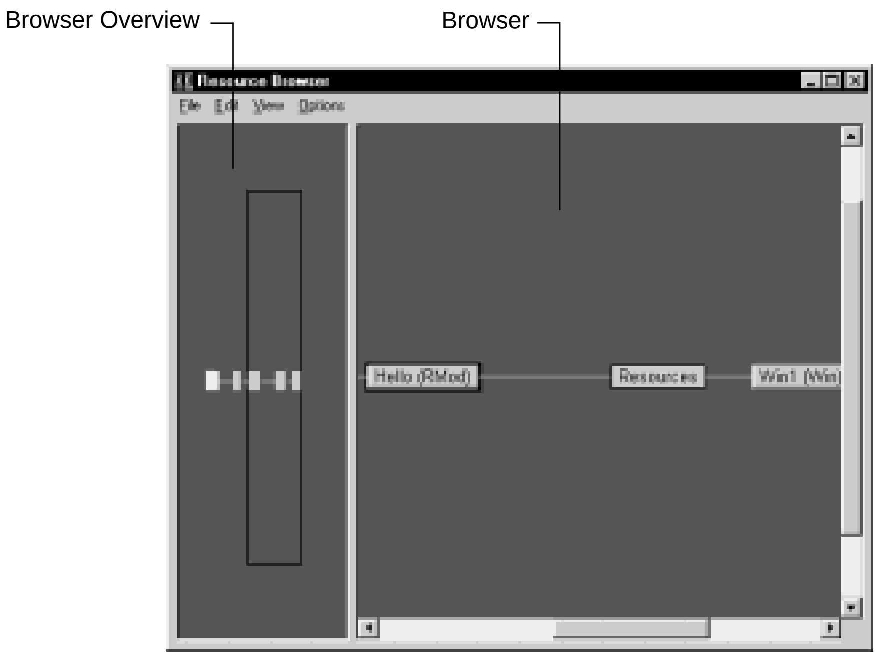


Figure 2-8 Resource Browser Window

The browser shows libraries, modules, and resources. (.dat files are also called *libraries*, as they are actually collections of the objects that make up your application.)

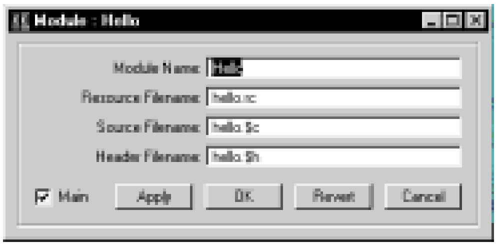
You edit the filenames of a module by double-clicking the module node icon. You can expand the module to view the window and associated widgets by right-clicking the module icon and holding the button, then choosing **Extend - Resources**. You can choose **Collapse** to retract the view. Other options are also available from the dropdown menu.

Task 2: Creating an Application-Startup Module

Producing an application-startup script is an essential step in creating OOScript applications. To do this, you need a module designated as "main," and you need to insert some startup OOScript statements in the script attached to the Hello (Rmod) module:

To create an application-startup module:

1. Double-click the **Hello (RMod)** node in the Resource Browser.
2. If **Main** is not checked in the Module Attributes dialog box, select it.
3. Click **OK**.



Module : Hello

Module Name: Hello

Resource Filename: hello.rc

Source Filename: hello.$c

Header Filename: hello.$h

☑ Main

Figure 2-9 Hello Module Attribute Dialog

4. Right-click the **Hello (RMod)** node in the Resource Browser and choose **New Application Script**.

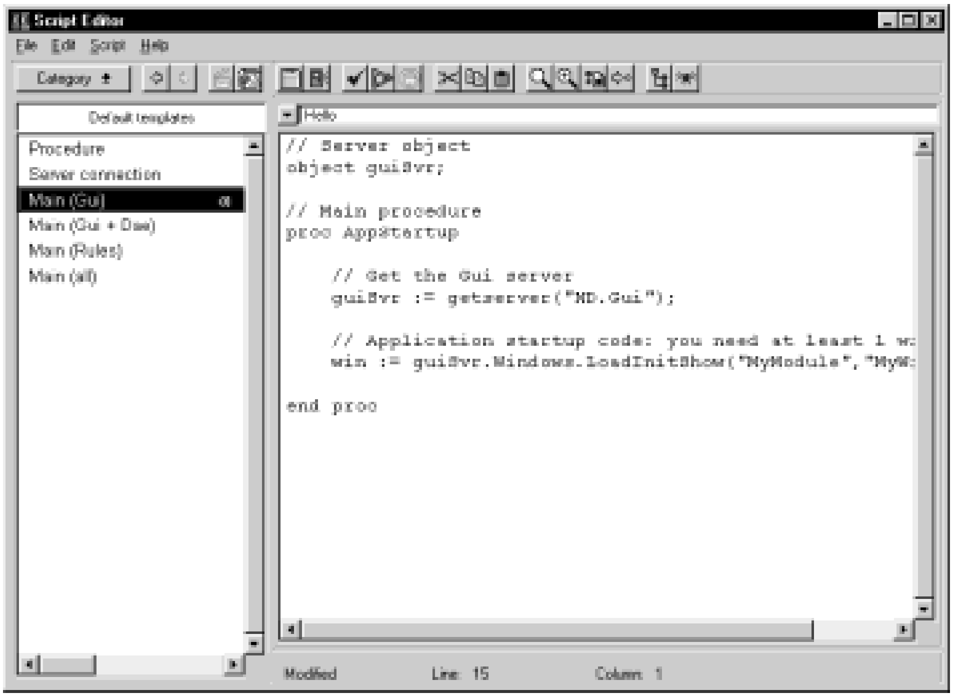
   When you use OOScript, all your applications must have an **AppStartup** procedure. The *AppStartup* procedure is the script that is executed when the application launches. It provides the essential startup information for your application. You can either insert the **AppStartup** code using Script Editor code templates, or you can write it yourself.

   Note: If you are coding in C or C++, you do not need an **AppStartup** script. Instead, you define the startup procedure in the function **main()**.

5. Choose **Category - Code Templates - Default Templates - Main(Gui)**.
6. Drag and drop the **Main(Gui)** startup script into the script-editing area.

Tip: To drag and drop the script, click the Main(Gui) listing once to highlight it. Then move the cursor to the right of the highlighted area. When the arrow cursor changes to a ⇥ , drag and drop the text into the script-editing area.

Figure 2-10 shows the **AppStartup** script for the Hello module.



```
// Server object
object guiSvr;

// Main procedure
proc AppStartup

    // Get the Gui server
    guiSvr := getserver("MD.Gui");

    // Application startup code: you need at least 1 w
    win := guiSvr.Windows.LoadInitShow("MyModule","MyW

end proc
```

Figure 2-10 **AppStartup** Script

7. In the script, replace `ModuleName` and `MainWindowName` with the names you gave your module and window. (These should be **Hello** and **Win1**.)

Note: In your script statements, when you want to assign an object reference to a variable (or another object reference), you need to use the “object-initialization and assignment” operator (:=). The := operator means that the object variable on the left side of the operator will contain a reference to the object resulting from the expression on the right side of the operator.

8. If you wish, test the syntax of your **AppStartup** script statements by choosing **File - Check Script Syntax**.

   If you find errors, go over the steps carefully to see if you have left out or mistyped anything.

Task 3: Writing OOScript Application Logic

Now that you have your startup module, you can add behavior to the pushbutton widgets. You have two options for writing your code:

- *Callbacks* allow your widgets to share event-handler procedures. You can group the callbacks in modules and reuse them for new widgets. Callbacks permit a more modular and efficient use of code by avoiding duplication of similar code in different widgets.
- *Event-handler scripts* allow you to write event-handler procedures for individual widgets. You should use this technique when behavior is unique to a specific widget**.**

The next two sections describe these two programming techniques.

Using Callbacks

To use callbacks, you add the code as procedures to the application script. You can either add keywords that appear in the code using the options in the Category menu, or you can type the code yourself.

For this introductory application, you will add code to the existing **AppStartUp** procedure and create two new procedures.

1. To add the code using the Category menu options, choose **Category - Resources**.
2. Double-click the name of the module **Hello**.

   This displays the resources for the module window.
3. Click **PButGreeting**.
4. Move the cursor to the right of the highlight until it appears as a +I .
5. Insert **PButGreeting** in the main procedure beneath this line of code:

   ```
   win := guiSvr.Windows.LoadInitShow("Hello","Win1");
   ```
6. Repeat steps 3–5 for **PButFarewell**.

7. Modify the two lines where the resource names have been copied as follows:

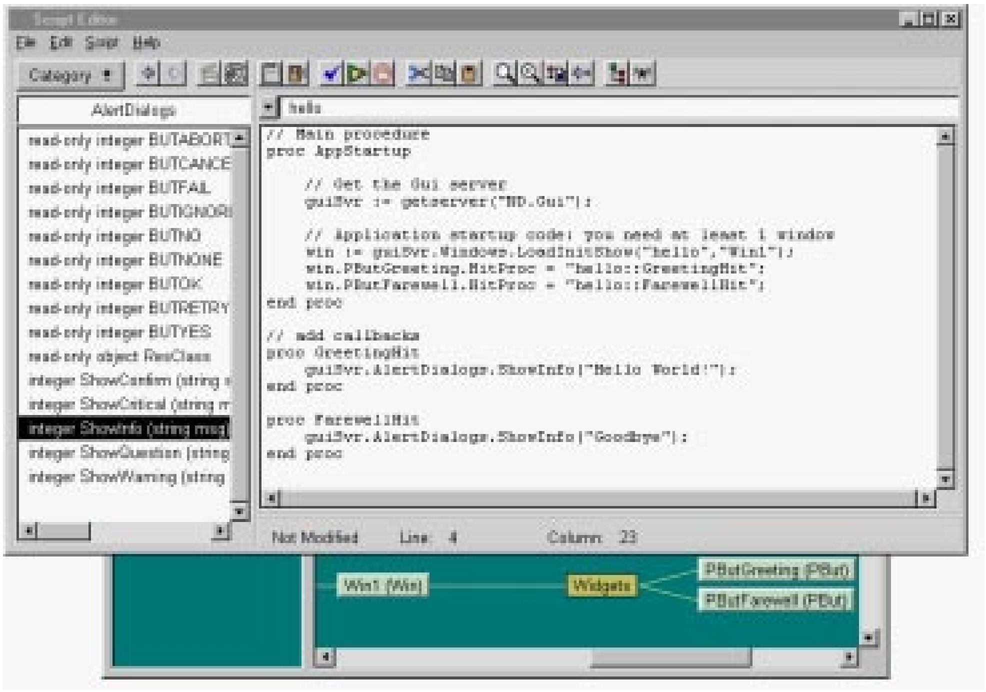
```
win.PButGreeting.HitProc="Hello::GreetingHit";
win.PButFarewell.HitProc="Hello::FarewellHit";
```

Note: You could have typed the names of the pushbuttons. However, in a complex application with many widgets and modules, using the Category menu options is convenient.

These two lines of callback code reference the procedures **GreetingHit** and **FarewellHit** for the **HIT** event. The callbacks instruct the application to invoke the appropriate procedure when the user generates a “hit” event by clicking one of the buttons.

8. Write the two procedures referenced by the callbacks beneath the **AppStartup** procedure as follows:

```
proc GreetingHit
    guiSvr.AlertDialogs.ShowInfo("Hello World");
end proc
proc FarewellHit
    guiSvr.AlertDialogs.ShowInfo("Goodbye");
end proc
```


Script Editor
File
Edit
Script
Help
Category
AlertDialogs
read-only integer BUTABORT
read-only integer BUTCANCEL
read-only integer BUTFAIL
read-only integer BUTIGNORE
read-only integer BUTNO
read-only integer BUTNONE
read-only integer BUTOK
read-only integer BUTRETRY
read-only integer BUTYES
read-only object ResClass
integer ShowConfirm (string
integer ShowCritical (string
integer ShowInfo (string msg)
integer ShowQuestion (string
integer ShowWarning (string
hello
Not Modified
Line: 4
Column: 23
Win1 (Win)
Widgets
PButGreeting (PBut)
PButFarewell (PBut)

```
// Main procedure
proc AppStartup

    // Get the Gui server
    guiSvr := getserver("ND.Gui");

    // Application startup code: you need at least 1 window
    win := guiSvr.Windows.LoadInitShow("hello","Win1");
    win.PButGreeting.HitProc = "hello::GreetingHit";
    win.PButFarewell.HitProc = "hello::FarewellHit";
end proc

// add callbacks
proc GreetingHit
    guiSvr.AlertDialogs.ShowInfo("Hello World!");
end proc

proc FarewellHit
    guiSvr.AlertDialogs.ShowInfo("Goodbye");
end proc
```

Figure 2-11 Using Callbacks in a Script

Tip: You can find the names of script methods, their return values, and their parameters online. To do so, choose **Category - Repositories - Neuron Data Gui Server**. Select the appropriate category. Select the class you want to view.

Using Event-Handler Scripts

You can also write event handlers to respond to the widget events.

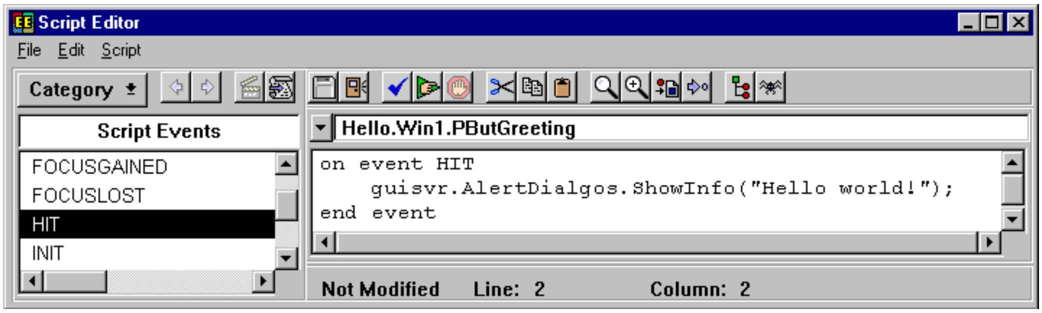
To create an event handler:

9. Delete the two lines of callback code you added to the AppStartup procedure and the two procedures you added.
10. Choose **Category - Resources.**
11. Double-click **Hello**.

    This displays the resources for the module window.
12. Select **PButGreeting**.
13. Click **Open Script**.
14. Choose **Category - Script Events.**
15. Select the **HIT** event, and drag and drop it into the script-editing area to obtain the following script template:

    ```
    on event HIT
    end event
    ```
16. Within the event-handler script, type this line:

    ```
    guisvr.AlertDialogs.ShowInfo("Hello world!");
    ```



```
on event HIT
    guisvr.AlertDialgos.ShowInfo("Hello world!");
end event
```

Figure 2-12 Event-Handler Script

Note: You can also use the Script Browser to insert verbs, objects, methods, properties, globals, and constants into your script. For more information, see the *OOScript Language Programmer's Guide*.

Task 4: Test and Run the Script-based Application

1. In the Script Editor, choose **File - Check Script Syntax** to check your script.
2. Click **Script Tracer** to start the OOScript trace facility.

   The Trace window displays a line-by-line analysis of the currently running script.
3. To run the application, choose **File - Run Script**. This displays the `Hello` window.

   Note: By default, the Run Script command looks in your currently active library to compile and run a script. If you have multiple applications loaded in the Resource Browser, the **File - Set Current Library...** command allows you to determine the current library and to switch to other libraries if necessary.
4. If you have runtime errors, go back and review the previous steps.
5. If you click **Farewell** in the application window, the results should look like this:

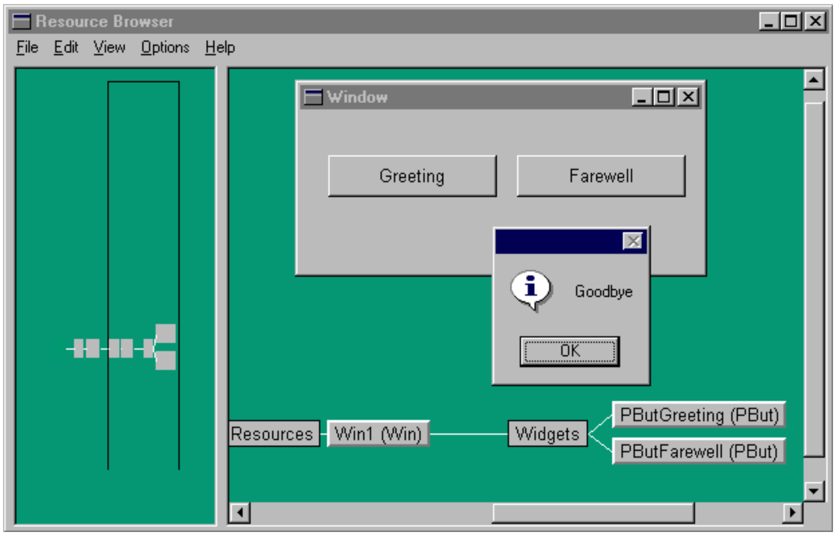


Figure 2-13 Run Script Results

Alternate Ways of Running a Script

You can also run your script from the command line:

```
ee -ND_DYNCONFIG=runscrpt <LIBRARY_NAME>
```

- **UNIX** users can run the Elements Environment runscrpt option in a C shell and create an alias to the command.
- **PC** (Windows 3.1/95/NT) users can type the command line in the MS-DOS window or just click the RunScript icon to execute the application in runtime mode.

To Edit Existing Applications

1. From the Resource Browser, choose **File - Open Library**.
2. Open the application library file.
3. Right-click and hold on the library icon.
4. Choose **Full Extend - Modules, Resources**.
5. To edit windows, double-click the appropriate resource icon.
6. To edit a script, right-click and hold on the appropriate module icon, then choose **Edit Application Script**.

Task 5: Deploying and Porting Applications

Note: These instructions assume that you are using MSVC Microsoft Developer Studio. However, the steps would be similar for other development environments.

1. Launch Microsoft Developer Studio.
2. Close any workspaces you may have open by choosing **File - Close Workspace**.
3. Choose **File - Open Workspace**.
4. Move to your working directory and give your workspace a name.
5. Choose **File - Open**.
6. Set **File of Type** to **All Files(*.*)**.

7. Select the appropriate makefile:
   - makefile.pc for Window 95 or Windows NT platforms
8. Click **OK**.
9. If necessary, choose **File - Save As** and change the filename to the name of the application.

   This creates the appropriate project workspace.
10. If necessary, open the .c or .cpp file and implement changes.
11. Choose **Build - Build <myapp>.exe**.
12. Make appropriate changes to your runtime environment.

Application Components

Open Interface generates these components for each application:

- **.rc files** contain (in modules) definitions of window objects, and if desired, script procedures and script application logic.
- **.dat files** are the binary format of the information defined in the .rc files.
- **Source files** for the C or C++ languages containing the application logic.
- **Image files** used in icons:
  - .nxp (flat files containing business objects and rules containing simple data)
  - .gif, .tiff, .bmp icon files containing the definition of icons used in the GUI
  - html files (on a server) for use with the Web Element with .dat files for applets

Note: The .dat files are portable across platforms. You also need to use an Elements Environment runtime configuration and the Neuron Data libraries and resources to build and run the application.

For More Information about the Open Interface Element

- See the *Open Interface Element C/C++ Programmer’s Guide.*
- See the *Open Interface Element C/C++ API Reference, Vol. 1, Widget Classes.*
- See the *Open Interface Element C/C++ API Reference, Vol. 2, GUI Helper Classes.*
- See the *Open Interface Element User’s Guide.*

For more information about Elements Environment executables and the ee runscrpt option, refer to Chapter 10, “Building Applications in the Elements Environment,” in this manual.

Chapter

3 The Data Access Element

The *Data Access Element* (DAE) is a component of the Elements Environment. The Data Access Element is designed to:

- Give you access to multiple data sources, such as relational databases, flat-file systems, object-oriented databases, or transactional databases
- Insulate you from the underlying complexities of data access, such as database connections
- Let you take advantage of database-specific functionality such as stored procedures, triggers, and referential integrity

With the Data Access Element, you can give a single application simultaneous, full read-and-write access to heterogeneous data sources in a host-based or networked client-server environment. The Data Access Element supports these standard datatypes:

- numeric
- interval
- ANSI
- char
- real
- variant
- date
- time
- "blobs" such as text and images

In addition to transparently retrieving data, the Data Access Element gives you an *abstraction layer*, which separates the logical view of the data from the specific data source. This makes it much easier and more intuitive to manipulate data.

The Data Access Element gives you an *application programming interface* (API). This allows your applications to access any relational, flat-file, or hierarchical database on the network, including legacy data sources. The Data Access Element API is open and extensible. This lets you extend its functionality to meet any special requirements that your applications may have.

Each data source has a corresponding Data Access Element *driver*, which handles communication between the Data Access Element API and the data source. The Data Access Element fully supports all server-specific functionality related to the underlying data source.

The Data Access Element and OOScript

You can use the Data Access Element objects in Neuron Data's OOScript language for seamless integration with:

- Other Elements Environment objects
- OLE-automation server applications

Using the OOScript language, the Data Access Element objects are available:

- On all platforms as in-process servers
- For Microsoft Windows 3.1/95/NT as out-of-process, local servers

Quick Tour

The Data Access Element gives you several key components for manipulating data:

- Data-access objects
- DBVu resource
- An object API

Generic Data-Access Objects

The Data Access Element includes a set of generic *data-access objects*. These are application resources that let you separate application code from data-access information, which remains unchanged after each application execution. Data-access objects give your application a more modular structure, which makes it more efficient and maintainable. Custom editors in the Elements Environment let you create these data-access objects:

- Connection object
- Query object
- Virtual-table or RecordSet object

DBVu Resource

You can use custom editors in the Elements Environment to combine data-access objects into a *DBVu resource*. The DBVu resource accesses data-dictionary information, and it lets you view and edit its data.

You can use the Elements Application Services (EAS) *datasource/views* mechanism to link GUI objects with a DBVu resource. This lets you establish one or more views of the same data.

Once registered with a datasource object, the GUI objects are synchronized with the data accessed through the DBVu resource. By separating the data from the views, the datasource / views mechanism automatically maintains consistency between the data and the user interface.

You can register these Open Interface Element GUI objects with DBVu resources:

- Text-edit areas (input fields)
- Check boxes
- Choice boxes
- List boxes (forms and tables)

Object API

Each Data Access Element resource described in this manual has a corresponding full-function API accessible from the C and C++ languages. The purpose of these resource API modules is to give you runtime control of the data-access objects:

- NDCnx (connection) class
- NDDbVu (view) class
- NDQry (query) class
- NDVTab (virtual table) class

For More Information about the Data Access Element

- See the *Data Access Element C/C++ Programmer's Guide.*

Related Subjects

- OOScript language
- Datasource / views mechanism in Elements Application Services and OOScript

Chapter

4 The Intelligent Rules Element

The *Intelligent Rules Element* (IRE) lets you develop knowledge-based applications using a rich and flexible graphical user interface (GUI). You do not have to be an expert in programming to use the Intelligent Rules Element.

The Intelligent Rules Element is a hybrid system that supports:

- A reasoning system
- A powerful object-oriented representation

Reasoning System

To represent reasoning, the Intelligent Rules Element uses rules. *Rules* are knowledge structures that let the system perform actions based on data, such as proving a goal or deducing a conclusion. A rule is also a chunk of knowledge representing a situation and its immediate consequences.

This is the format of a rule:

`if` *conditions* `then` *hypothesis* `then do` *actions* `or else do` *alternative actions*

- The *hypothesis*, or goal, becomes true when the conditions are met.

  Conditions define expressions that test the values of slots.

  A *slot* is a data value represented by an object and its property.
- *Actions* are undertaken when the conditions are positively evaluated.
- *Alternative actions* are undertaken when any one of the conditions is negatively evaluated.

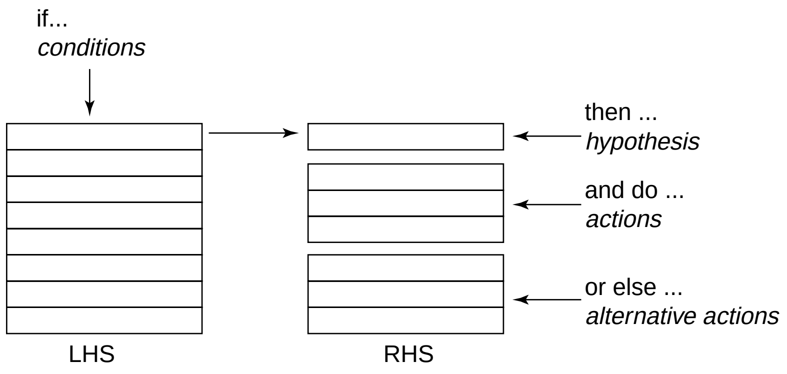


Figure 4-1 The Components of a Rule

The block to the left of the arrow represents the *Left-Hand Side* (LHS) of the rule. This is where the conditions are expressed, with individual conditions represented by horizontal lines.

The blocks to the right of the arrow form the *Right-Hand Side* (RHS) of the rule:

- The first block is the *hypothesis*, or goal, to be proven.
- The middle block represents *actions* you specify for a positive evaluation of the rule's LHS (THEN actions).
- The lower block represents *alternative actions* you specify for a negative evaluation of the rule's LHS (ELSE actions).

Rule Dynamics

This section briefly examines how the Intelligent Rules Element processes rules.

Rule Evaluation

The building block of the reasoning path is the single rule. The Intelligent Rules Element processes one rule at a time. For example, assume that the value of a slot involved in one of a rule's conditions is known. Since this slot's value is known, the rule is "relevant." The system can use this rule to try to make inferences. Each condition of a rule's LHS can have one of these values:

- "true"
- "false"
- "not known"

Assume now that all of the conditions (each line on the LHS) are verified and are true. The rule can now be *triggered*.

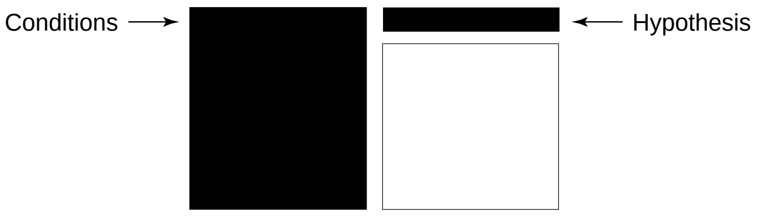


Figure 4-2 The Evaluation of a Single Rule

In Figure 4-2, areas that appear in black were found by the system to be true.

Actions

Rules usually have more complex structures than the simple case of an RHS made up of just an hypothesis. Such structures let you add *actions*, which induce some change in the overall system or its environment. Figure 4-3 shows the kinds of actions that can be *initiated* when a rule is triggered:

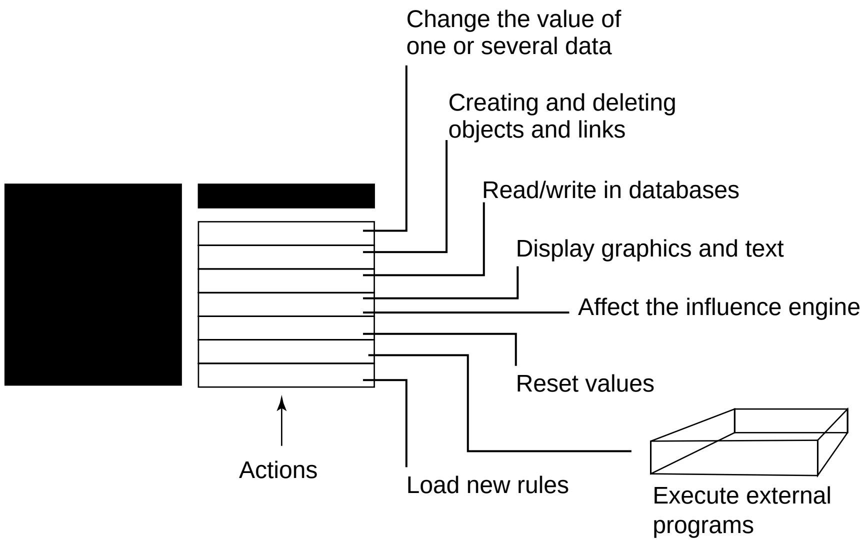

Change the value of one or several data
Creating and deleting objects and links
Read/write in databases
Display graphics and text
Affect the influence engine
Reset values
Actions
Load new rules
Execute external programs

Figure 4-3 Possible Actions Initiated by the Evaluation of a Rule

Backward Chaining

A hypothesis has a value; therefore, it can appear in the LHS of a rule as a condition that the system will verify. Consider the situation that Figure 4-4 depicts:

- The hypothesis, *hypoA*, of the rule has an unevaluated condition (shown in gray).
- The unevaluated condition is itself another hypothesis, *hypoB*.
- To evaluate *hypoA* to be true, the system must verify that *hypoB* is true.

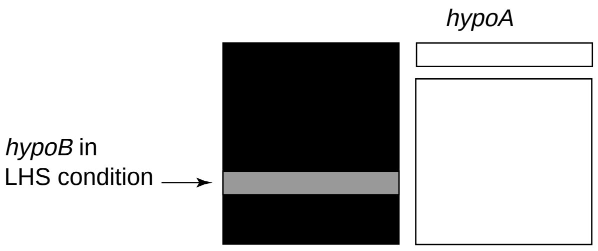


Figure 4-4 A Hypothesis as a Condition

To verify whether *hypoB* is true, the system must find one or more rules with *hypoB* as the hypothesis. Once the system finds a rule with *hypoB* as its hypothesis, it evaluates the conditions in the LHS of the rule leading to *hypoB*, as shown in Figure 4-5:

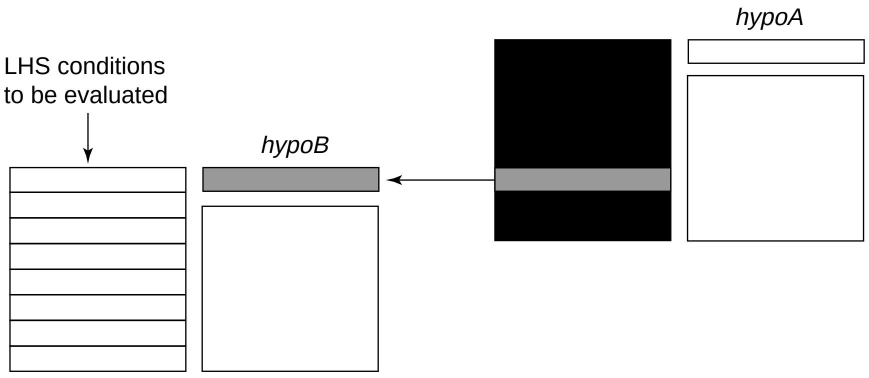


Figure 4-5 The Evaluation of a Hypothesis as a Condition

This inference evaluation that the Intelligent Rules Element conducts is a deductive process, or *backward chaining*. It can be made at many levels and can involve many rules, as shown in Figure 4-6:

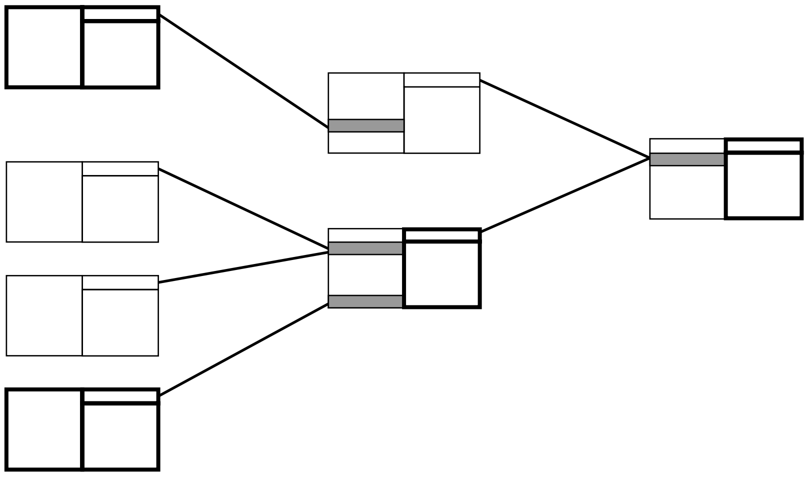


Figure 4-6 Backward Chaining with Multiple Hypotheses

Note: In multilevel backward chaining, there might be conflicts between rules because more than one rule may lead to a single hypothesis. The Intelligent Rules Element has special mechanisms to deal efficiently with such conflicts in a way you can define.

Forward Chaining

When the RHS actions of a rule change the values of data in other rules, this can trigger the evaluation of those rules, as shown in Figure 4-7. The triggered rule (in black) brings three other rules on the *agenda* for evaluation. This is an evocative progression, or *forward chaining*. The upper left rule has been evaluated but not activated because of a nonverified condition (indicated by a cross). Thus, its hypothesis is found to be *false*, and its RHS actions fail to occur.

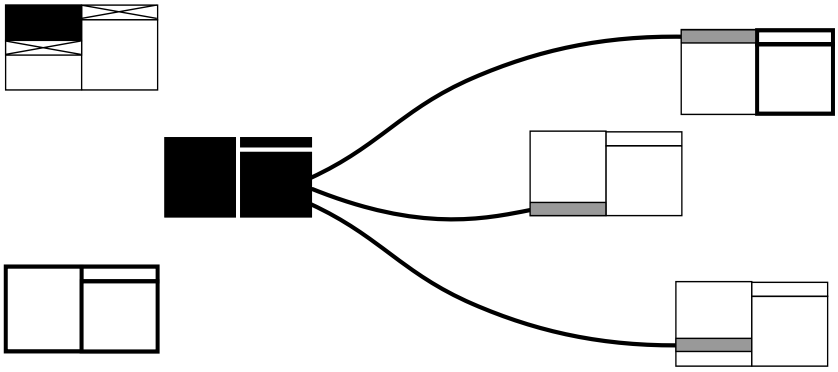


Figure 4-7 Forward Chaining Triggered by RHS Actions

Revisions

Rules can also trigger *revisions* of other rules. Consider rule 1 in Figure 4-8. The value of a variable caused the condition in rule 2 to fail. However, later an action of rule 1 modifies the state of the variable in rule 2.

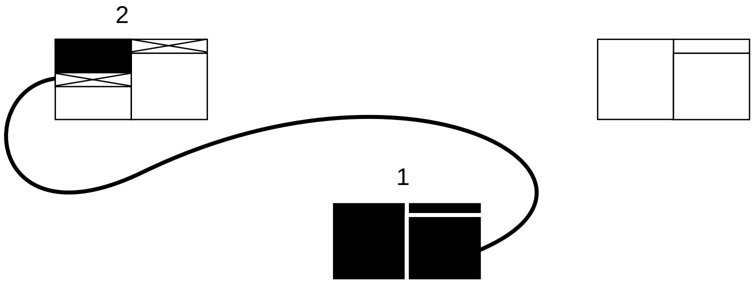


Figure 4-8 Reevaluation of a Previously False Rule Triggered by Actions

If the inference engine permits it at a particular time, the rule will be revised, and then perhaps activated. The activation of rule 2 might place rule 3 on the *agenda* for evaluation through forward chaining, as shown in Figure 4-9:

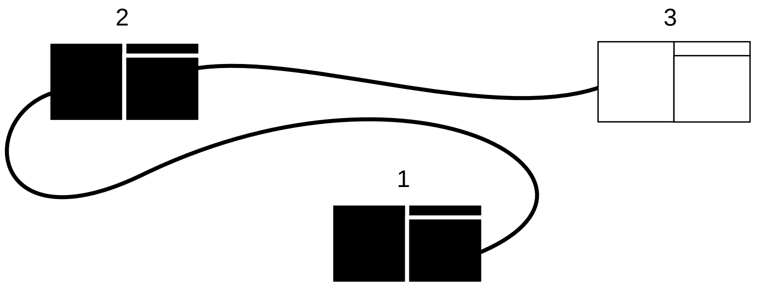


Figure 4-9 Forward Chaining Triggered by Rule Revisions

Integration

From the perspective of knowledge processing, rules in the Intelligent Rules Element are *symmetric*. This means that they can be used in both backward and forward chaining. Consequently, it is not necessary to define a rule as “forward chained” or “backward chained.” How the system processes rules at a given time depends on:

- The events that occur because of an action or external information
- The current hypothesis that the inference engine is trying to evaluate

Figure 4-10 shows an example of a reasoning path that integrates various mechanisms in a knowledge base with several rules. The reasoning path follows this order:

1. After Rule 1 is triggered, rule 2 is placed on the agenda for evaluation.
2. The system’s focus is on rule 2 (circled in Figure 4-10).
3. Rule 2 triggers backward chaining and brings rule 3 under evaluation.
4. Rule 3, in turn, triggers backward chaining, which brings rule 4 and rule 5 under the system’s evaluation
5. During this backward-chaining process, an action of rule 5 brings rule 6 to the system’s attention.
6. The final evaluation of rule 2, as a result of the backward chaining, triggers rule 7.

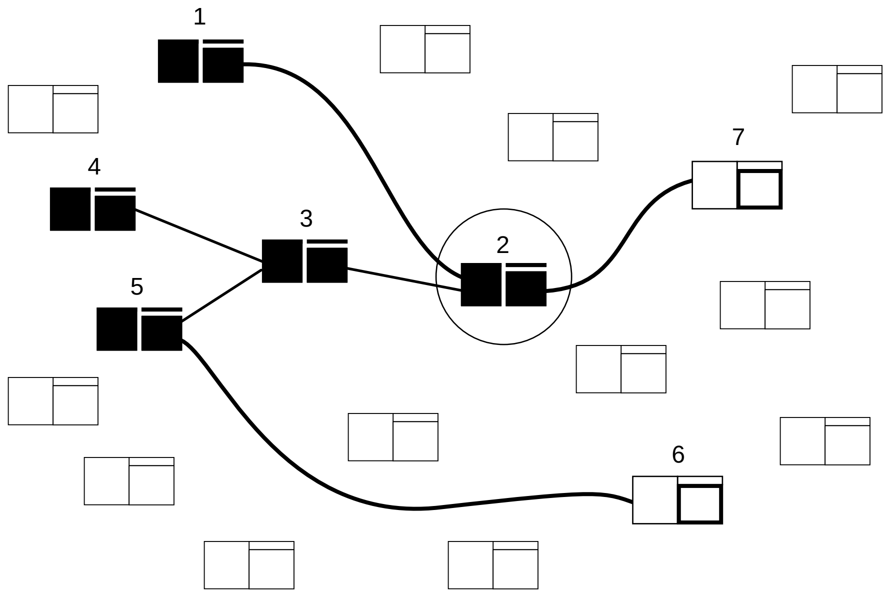


Figure 4-10 Integrated Forward and Backward Chaining

As shown in Figure 4-10, a rule is evaluated for one of these reasons:

- It solves the present subtask.
- It is the most relevant to the reasoning process.

This *opportunistic* character of the Intelligent Rules Element architecture lets the system follow the best line of reasoning. This is crucial for building systems that need to adapt to changing environments.

Open Architecture

The Intelligent Rules Element architecture is *event-driven*:

- It can integrate messages from external programs, which include those triggered by the Intelligent Rules Element rules or objects.
- A rule or a hypothesis can become relevant when an external event justifies its evaluation, even if the system is currently evaluating another area of the knowledge base.
- You can even make the external events have priority over the current focus (this priority can be set by the external mechanism).

You can control the effect the integration with external events has on knowledge processing. The Intelligent Rules Element provides *inference-control mechanisms* that can either be set globally or incorporated into the rules themselves. These mechanisms can affect the backward-chaining paths (their exhaustivity, for example) or the forwarding of RHS actions. They allow you to prevent specific actions from having any effect on the system's agenda and focus. The set of functions controlling these mechanisms is called the *strategies*.

Object-oriented System

Rules use reasoning on a *data representation* of the problem. This representation is made up of interrelated objects. As shown in Figure 4-11, the representation dimension intersects the reasoning system at the data level:

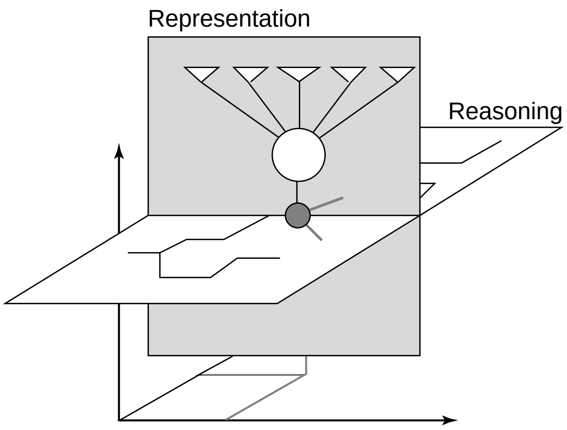


Figure 4-11 Intersection of the Reasoning and Representation

Object Structure

The object structure includes:

- Objects
- Properties
- Classes
- Methods

Object

An *object* is a basic unit of description. Objects represent the knowledge on which the rules reason. Objects also describe variables in the knowledge base. You can define hierarchical relationships between objects to give rules greater flexibility in reasoning.

Property

A *property* is a characteristic that you can associate with an object or a class. The combination of an individual object and a specific property is called a *slot*. Slots store the actual values.

As an example, consider a condition of an Intelligent Rules Element rule:

```
Is object1.property1 "blue"?
```

This condition is part of the LHS of a rule, which is the current focus of attention. This is a translation of this syntax:

“Is the value of the slot (property ‘property1’ of the object ‘object1’) blue?”

Class

A *class* is a collection of objects that usually share properties.

Consider this condition:

```
Is <CLASS1>.property1 blue
```

This condition translates to:

“Is there any object in the class ‘CLASS1’ whose slot ‘object*x*.property1’ has the value blue?”

Classes let you represent objects hierarchically:

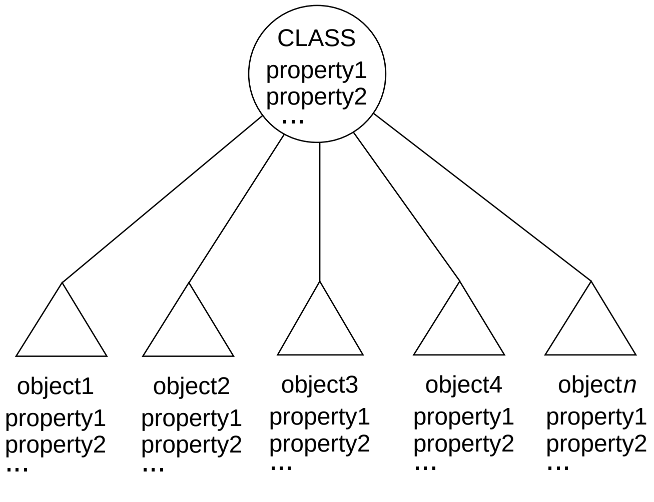


Figure 4-12 The Hierarchical Structure of Classes and Objects

Inheritance

Classes can store information relevant to all their objects. The objects, when necessary, **inherit** this information. This mechanism is called *inheritance*.

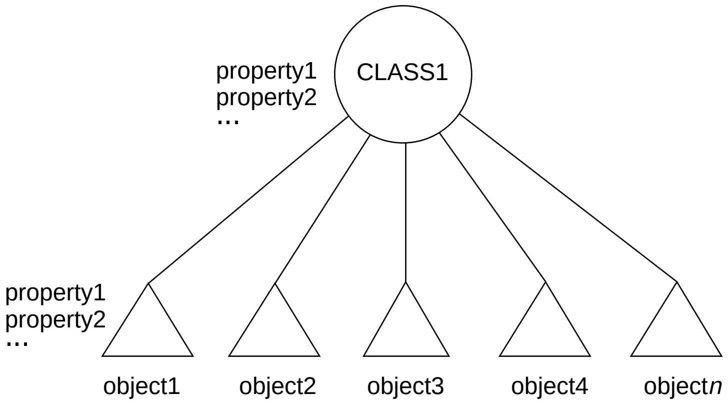


Figure 4-13 Inheritance of Properties by Objects of a Class

Method

A *method* is a sequence of actions associated with an object, class, property, or slot that executes under certain circumstances.

As an example, consider the following condition of an Intelligent Rules Element rule that is the current focus of attention:

```
SendMessage   "ComputeArea" To:object1
```

This condition translates to:

“Trigger the method ‘ComputeArea’ attached to the object ‘object1.’”

Encapsulation

Methods let you hide procedures related to an object’s unique behavior within the object itself. That way, you do not have to write them elsewhere in the rules of the knowledge base.

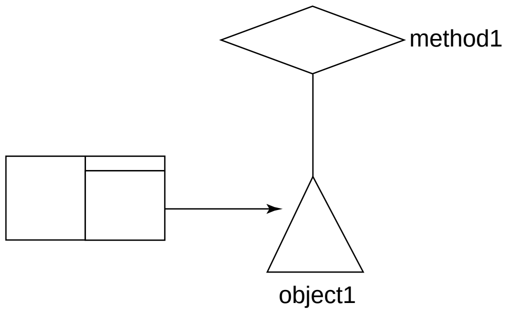


Figure 4-14 Method-Object Relationship

This idea that an object can be a self-contained unit that includes both the data and the procedures to process those data is known as *encapsulation*.

Polymorphism

Like properties, methods can be inherited by the children from their parent object.

Methods are best used to represent related knowledge about a set of objects. When each object has its own way of accomplishing the same task, the attached methods typically use the same name for each object, such as "Determine_Area" or "Calculate_Cost." Then when a rule sends a message to trigger the method, the message and the method bind with the list of objects that receive the message.

Figure 4-15 depicts one message that binds with several objects that have a method of the same name attached. The ability to use methods with the same or similar names to represent a type of task for more than one object is known as *polymorphism*.

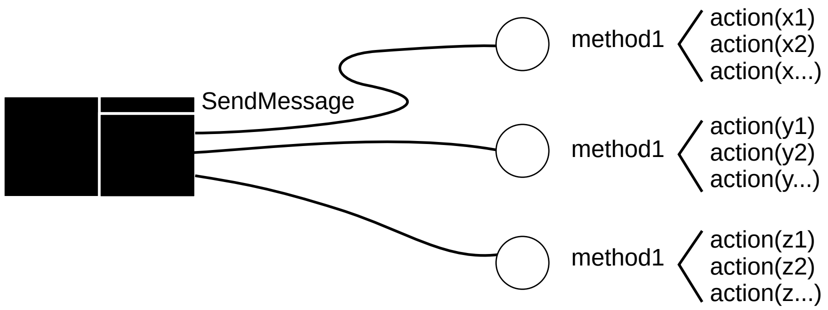


Figure 4-15 Same Message Triggering Different Method Actions

Pattern Matching

The class structure itself acts as a pointer to a set of objects, with data held by the structure formed by associating a property with each object. This structure is represented by the form `class.property` and is known as a *slot*.

Classes thereby provide a way to search through lists of objects in order to identify which objects meet a specific condition. This is called *pattern matching*. Consider the following pattern-matching condition of a rule:

```
=   <CARS>.color   blue
```

This means:

"Is there any object in the class 'CARS' whose slot 'object*x*.color' has the value blue?"

Objects in the class "CARS" all have the property "color." The system:

1. Evaluates the condition and identifies all the slots with the value "blue."
2. Automatically creates a list of objects for use by the rule's RHS actions.

Figure 4-16 depicts a pattern-matching situation where all objects with "slot1 = blue" will have their slot "slot2" set to 0 by the RHS actions:

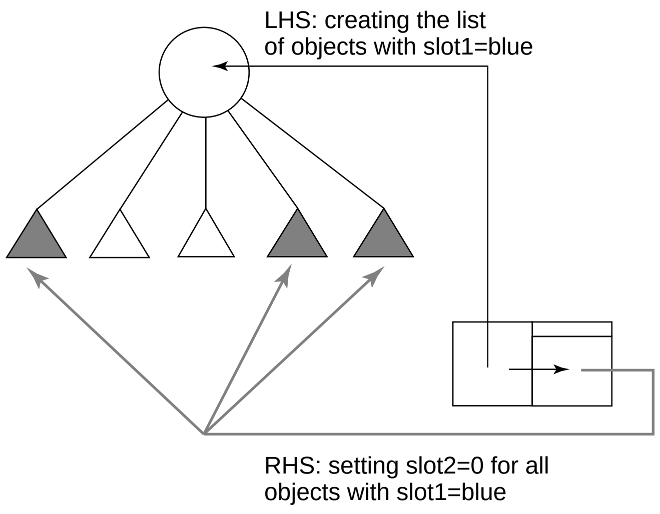


Figure 4-16 Pattern Matching on Class Creating List of Objects

While the list is created on the basis of a condition that incorporates "slot1," the RHS action can modify slot1 or any other slot of the same objects.

System Methods

Methods that the system automatically triggers under circumstances that you define are called *system methods*. You can define two types of system methods:

- Order of Sources
- If Change

Order of Sources

Order of Sources lets you define and prioritize the sources that the system can use during a session to obtain the value of a slot that is not known.

To evaluate a rule, the Intelligent Rules Element must have the appropriate information on which to base its conclusions. If the values of slots in the LHS conditions are unknown, the system must first obtain the values to complete the evaluation.

For example, consider the following condition, and assume that there is no current value for the slot “car.color”:

```
=    car.color    blue
```

Assume that the value of slot1 of the class “car” is unknown. The system will not be able to find the value of slot1 for the object, which it needs to evaluate the current rule. However, the system can switch to a different *source* that you have defined to obtain the value.

As shown in Figure 4-17, inheritance fails, and an external routine is computed that may provide the value:

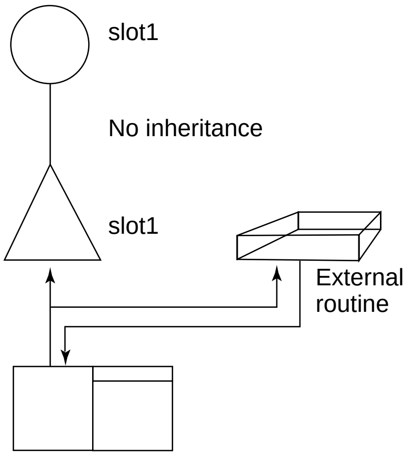


Figure 4-17 Value Computed Externally after Inheritance Fails

For any object slot, you can declare an Order of Sources. This object-oriented functionality adds robustness to the system. This is because a data value can be

- Inherited from the parent class
- Fetched from an external source
- Directly assigned

If Change Methods

If Change methods let you specify the actions that the system initiates whenever the value of a slot changes during the evaluation of a rule.

Assume that an RHS action of a rule changes the slot value:

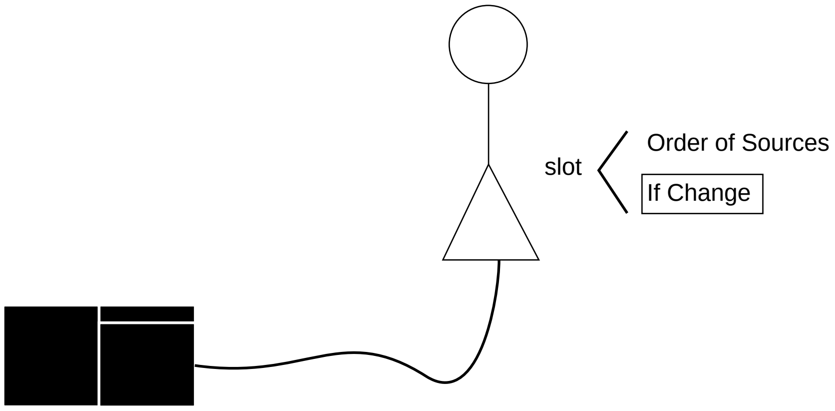


Figure 4-18 If Change Attached to an Object Slot

Assume also that there is a value for the slot at the level of the class to which the object belongs. When the value of the slot is changed, the system immediately executes the If Change methods.

User-defined Methods

You can specify when to trigger *user-defined methods* in an application. Another rule or method triggers these methods during the evaluation of a rule. A rule or a method always uses the *SendMessage* operator to trigger a user-defined method. Figure 4-19 depicts a rule that uses the SendMessage operator to trigger a method, which in turn uses its own SendMessage operator to trigger another method.

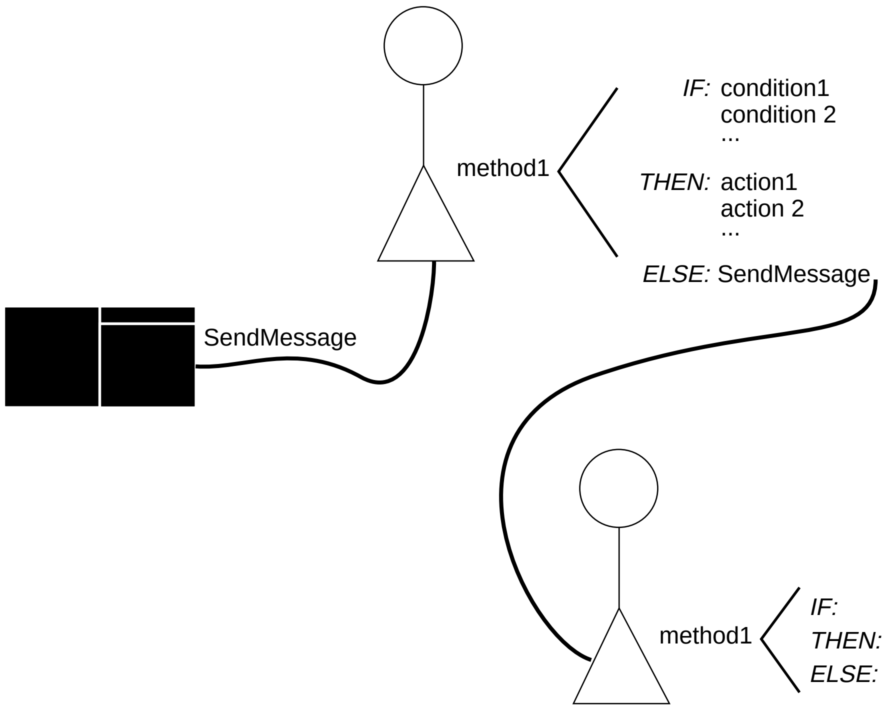


Figure 4-19 User-defined Methods Triggered by Message Passing

Graphical User Interface Dynamics

Interaction from the Intelligent Rules Element to the GUI

The Intelligent Rules Element inference engine generates several types of events that provide a variety of *graphical user interface* (GUI) interactions:

- Asking questions
- Displaying conclusions
- Displaying alert messages
- Updating displayed windows with new information such as:
  - a change in the value of a slot
  - a change in the structure of an object
  - unloaded knowledge-base file

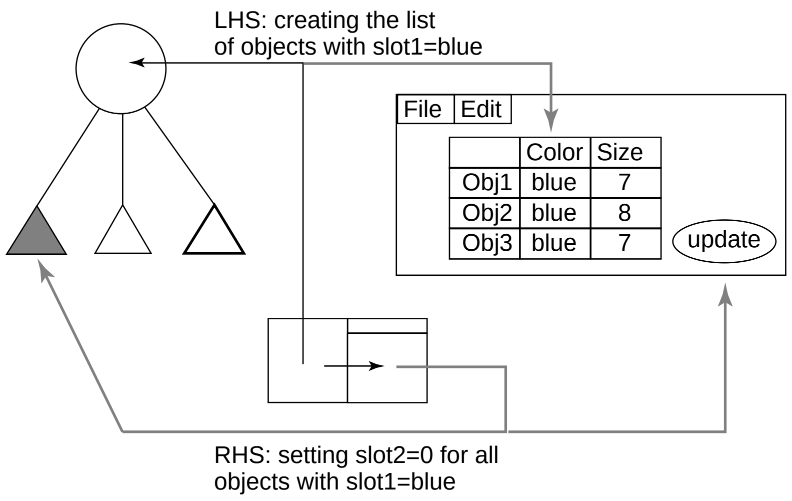

LHS: creating the list of objects with slot1=blue
File
Edit
Color
Size
Obj1
blue
7
Obj2
blue
8
Obj3
blue
7
update
RHS: setting slot2=0 for all objects with slot1=blue

Figure 4-20 Events in the Intelligent Rules Element

Figure 4-20 shows a create-object event generated by the Intelligent Rules Element inference engine. The GUI receives the event. The window then displays the data in a table element with:

- Rows that correspond to each object
- Columns that correspond to the properties of each object

To make this type of communication with the GUI possible, the Intelligent Rules Element uses the OOScript language. You can define scripts at these levels:

- The application and its modules to provide startup control
- A window
- The individual graphical element

The LHS action in Figure 4-20 is an example of *immediate updating*. The script engine handles the create-object event from the Intelligent Rules Element by updating the table.

Interaction from the GUI to the Intelligent Rules Element

The GUI has its own set of events generated by user actions. Typically, the end-user makes a selection from a menu or enters a value from the keyboard in response to an IntelligentRules Element event. The GUI engine monitors these user actions in the background. If one of these actions occurs within the area of a particular graphical element, such as a menu item or an input field, the system interprets it as a GUI event. Like their Intelligent Rules Element counterparts, GUI events allow you to initiate appropriate responses such as:

- Sending information to the Intelligent Rules Element
- Triggering inferences in the Intelligent Rules Element
- Requesting more information from the Intelligent Rules Element

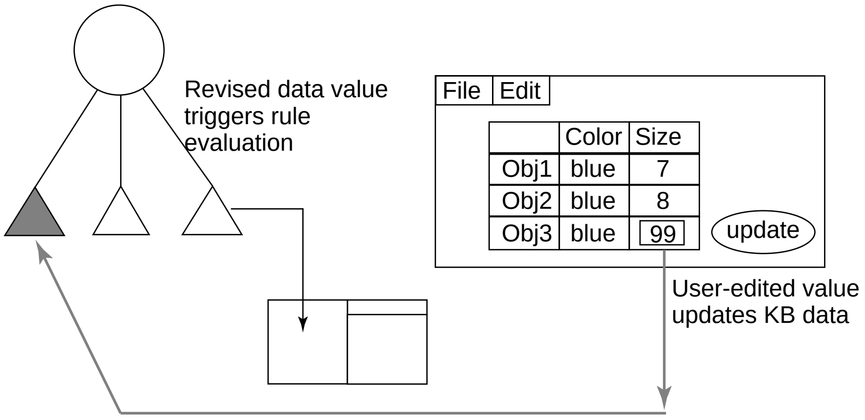

Revised data value triggers rule evaluation
File
Edit
Color
Size
Obj1
blue
7
Obj2
blue
8
Obj3
blue
99
update
User-edited value updates KB data

Figure 4-21 GUI Events and a Data Source

Figure 4-21 shows an update event that the GUI engine generates after the end-user modifies a value in a table and presses Enter. The same script engine that handles the Intelligent Rules Element events that the inference engine generates also handles GUI events that an end-user initiates. Thus, you can edit the scripts of your knowledge-based application while you construct the windows and GUI objects.

Building Applications

The Intelligent Rules Element lets you build knowledge-based applications for a wide range of tasks by using specialized knowledge-design facilities.

Starting the Intelligent Rules Element

How you start the Intelligent Rules Element depends on the type of operating system your computer uses:

IBM-compatible PC

1. Set the path to include Windows, the Intelligent Rules Element, and the knowledge-base directories.
2. At the prompt, enter `WIN EE20`.

UNIX Workstations

1. Start X Windows.
2. Change the directory to `$ND_HOME/bin`.
3. At the prompt, enter `ee20`.

Macintosh

Double-click the Intelligent Rules Element application icon.

The Main Window

When you start the Intelligent Rules Element, the main window appears. This is the window that controls the application. The main window displays important options for interacting with the Intelligent Rules Element:

- A list of the knowledge bases that you load into memory
- A Session Control panel to monitor events
- A customizeable control panel that lets you add button equivalents for commonly used menu items
- An inference engine status field
- A GUI engine status field

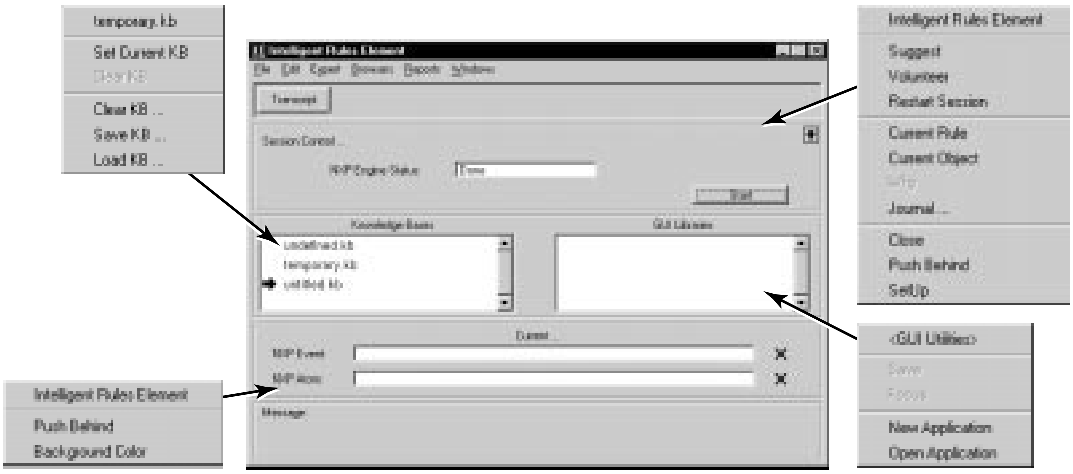


Figure 4-22 The Main Window of the Intelligent Rules Element

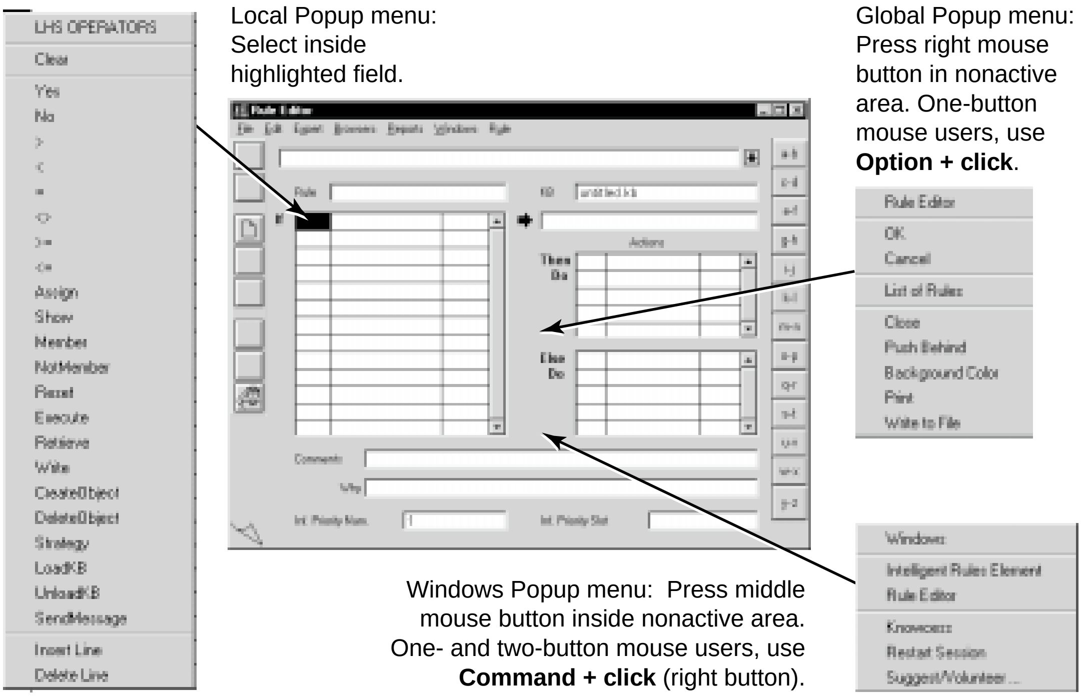
Note: See the *Intelligent Rules Element User's Guide* for a complete description of the main window and of the rules development environment.

Displaying Popup Menus

Popup menus provide additional functions for the various fields in the main window. Popup menus also group related functions or display only relevant functions. These are the three types of popup menus that you will use with the Intelligent Rules Element facilities:

Popup Type	Description	Selection Method
Local popup	Displays options for highlighted fields of the editor windows or for specific items displayed in the network windows. **Note:** Local popup menus that you display for individual fields are context-sensitive; thus, their options vary with the field selected.	Click the highlighted field. If you are using a one-button mouse, move the cursor slightly to the right to display the popup menu.

Global popup	Displays general options related to the active window outside of its highlighted field.	Click inside the active window but away from a field or button. If you are using a one-button mouse, use **Option + click**.
Windows popup	Displays the windows that are open and gets system options outside of a highlighted field.	Click anywhere in the active window using the middle mouse button. If you are using a one- or two-button mouse, use **Command + click**.



Local Popup menu: Select inside highlighted field.

Global Popup menu: Press right mouse button in nonactive area. One-button mouse users, use **Option + click**.

Windows Popup menu: Press middle mouse button inside nonactive area. One- and two-button mouse users, use **Command + click** (right button).

Figure 4-23 Sample Popup Menus

Entering Text

For building knowledge structures, the Intelligent Rules Element provides a set of six specialized *editor windows*. Each window has a set of *toolbar buttons* that you select to begin editing:

Button	Text Equivalent	Description
🗋	NEW	Clears the editor window and highlights the first field to accept your entry for a new application structure.
	EDIT	Places the currently displayed application structure in edit mode and lets you make changes.
	COPY	Makes a duplicate of the currently displayed application structure and lets you make changes.
🗑	DELETE	Deletes the currently displayed structure from the application. **Note**: You **cannot** reverse this operation.
OK	ACCEPT	Verifies the syntax of entries of the current application structure, accepts the structure into the application, and prepares the editor window for further additions.
✋	CANCEL	Returns the currently displayed application structure to its original unmodified state.
✓	CHECK	Verifies the syntax of the current application structure.
	FIND	Searches the application to locate the specified structure.

The following sections outline how to use the editors to build:

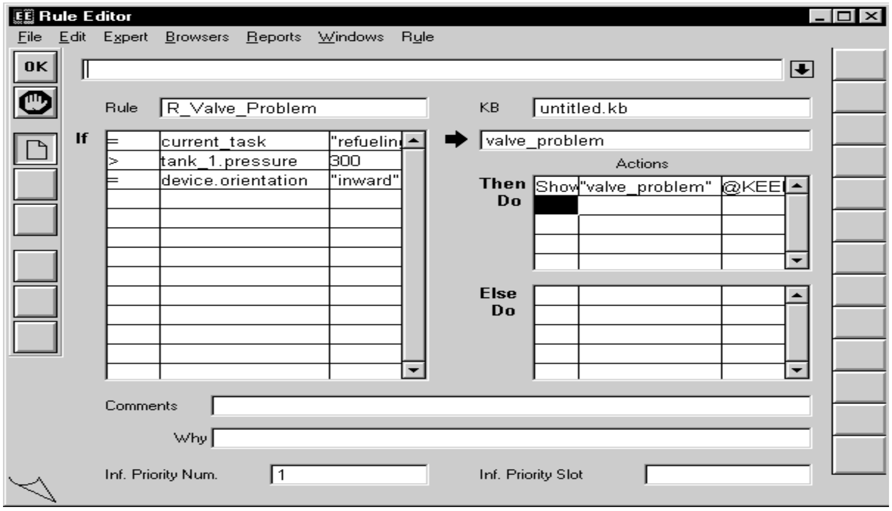
- Rules
- Objects
- Method structures

Rule Editor

You add rules in the Intelligent Rules Element by using the Rule Editor window. Like other editor windows, the Rule Editor aids the application-development process by acting as a template for entering data.

You can use the Rule Editor option to add, edit, or delete any rule in the knowledge base.

Figure 4-24 shows a rule with the conditions, hypothesis, and then-actions fields in the Rule Editor window.



Rule: R_Valve_Problem

KB: untitled.kb

=	current_task	"refuelin
>	tank_1.pressure	300
=	device.orientation	"inward"

→ valve_problem

Show	"valve_problem"	@KEE

Comments

Why

Inf. Priority Num. 1

Inf. Priority Slot

Figure 4-24 Rule Editor Window

This is an example of a simple rule with three conditions, a hypothesis, and a single action to recreate. The exact data for the rule shown in Figure 4-24 is as follows:

```
IF     =    current_task         "refueling"
       >     tank_1.pressure       300
       =    device.orientation   "inward"
THEN   valve_problem
       Show "valve_problem"
```

Building Rules in the Rule Editor

Note: Always use the text-edit line to enter or edit values for the highlighted fields in the Rule Editor.

1. Start the Intelligent Rules Element.

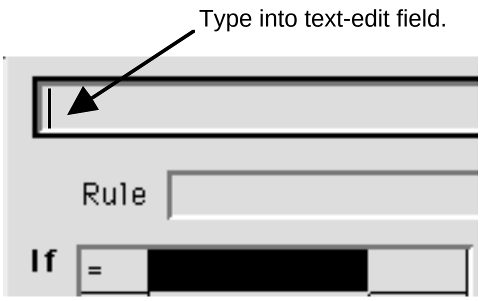
   See “Starting the Intelligent Rules Element” on page 49.

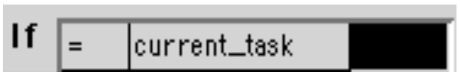
2. Choose **Editor - Rule**.
3. Click **New**.
4. Click the highlighted field of the conditions columns and display the local popup menu for this field.

Note: If you are using a one-button mouse, make sure the cursor looks like a small menu. Move the cursor slightly to the right to display this menu.

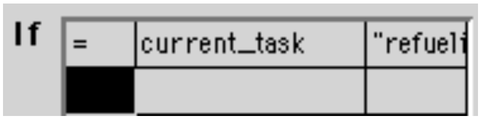
5. Select **=** (equals).

6. Enter the slot name `current_task`:



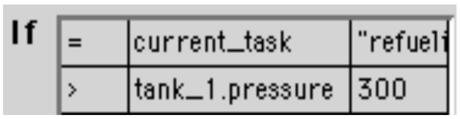


7. Enter `"refueling"` in the third field:

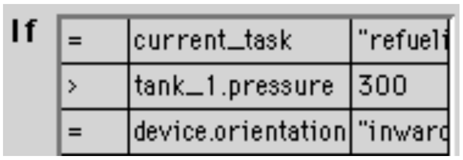


8. Follow steps 4–7 to complete the second condition as shown below:

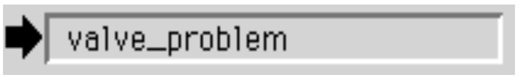
Note: Be sure to type an underscore in the name `tank_1.pressure` and to select **>** instead of **=**.



9. Follow steps 4–7 to complete the third condition as shown below:

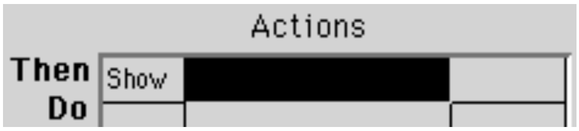


10. Click the hypothesis field, and enter `valve_problem`:

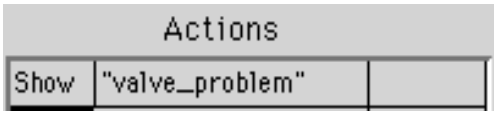


valve_problem

11. Display the popup menu for the Actions operator column, and select `Show`:



12. Enter the filename `"valve_problem"`:



13. In the Show File dialog, click **OK**.
14. Click **OK** to verify and compile the rule.
15. Leave the Rule Editor window open, and go to “Editing Object Structures in the Object Editor” on page 56.

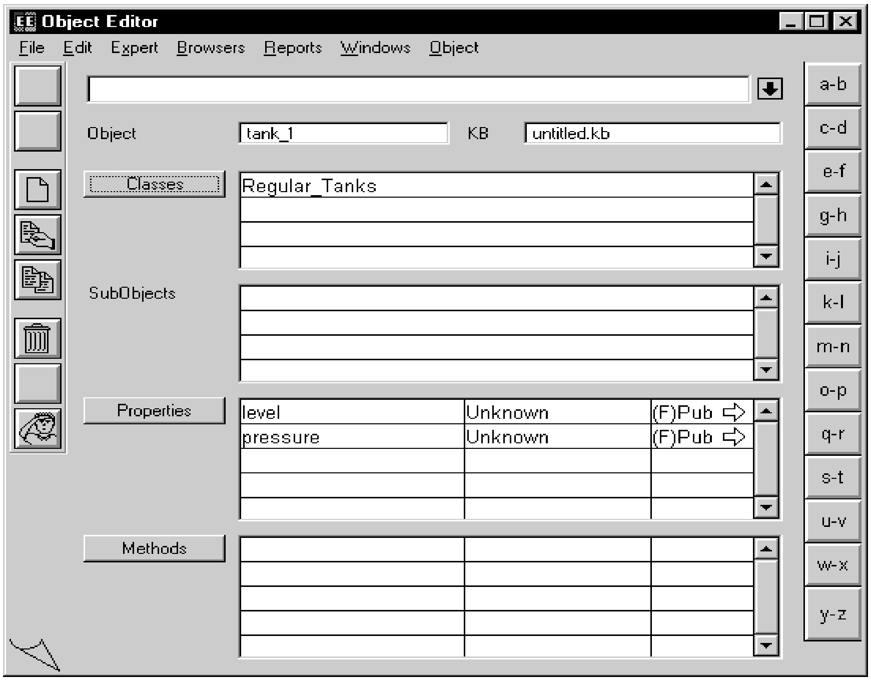
Object Editor

You can add classes and objects in the Intelligent Rules Element using the Object Editor. Like the Rule Editor, the Object Editor also serves as a template. You can shift from creating rules to editing objects or the reverse at any point while developing your application.

Figure 4-25 identifies the object name `tank_1`, which has two properties. Therefore, the Object Editor in this case defines two slots:

- `tank_1.level`
- `tank_1.pressure`

It also shows that these object structures belong to the class `Regular_Tanks`.



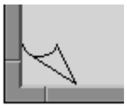
Object: tank_1 KB: untitled.kb

Classes: Regular_Tanks

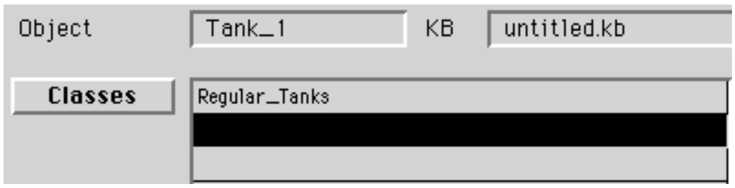
level	Unknown	(F)Pub
pressure	Unknown	(F)Pub

Figure 4-25 Object Editor

Editing Object Structures in the Object Editor



1. Choose **Edit - Object**.
2. In the Object Editor window, click the page-flip graphic on the bottom left-hand corner to display the object `tank_1` in the Name field.
3. Click **Edit**.
4. Enter the class name `Regular_Tanks`:

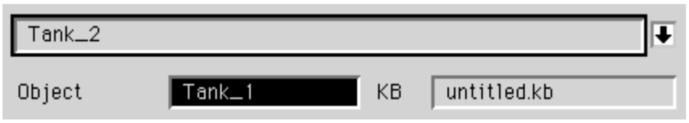


Object: Tank_1 KB: untitled.kb

Regular_Tanks

5. Click **OK** to verify and compile the class.

6. Click **Copy** to duplicate the object and its property.
7. Enter `Tank_2` to change the object name:



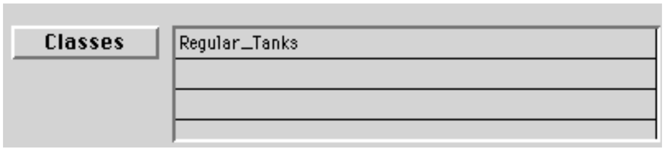
Tank_2

Object Tank_1 KB untitled.kb

8. Click **OK** to verify and compile the new object.

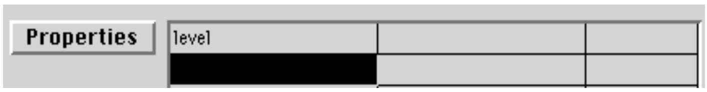
Editing Classes in the Class Editor

1. Click **Classes**.



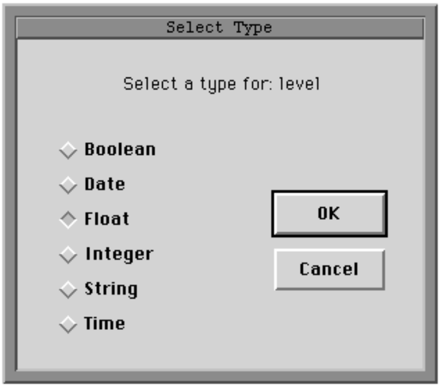
Regular_Tanks

2. Click **Edit**.
3. Click the first column of the Properties field, and enter `level` as the property name:



level

4. Click **OK** to verify and compile the class.
5. Select the datatype **Float** and click **OK**:

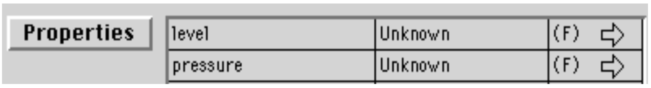


Select a type for: level

◇ Boolean
◇ Date
◇ Float
◇ Integer
◇ String
◇ Time

6. Click the Object Editor window.

   A second property level now appears. This is due to the creation of a property automatically inherited from the Class Editor.



Properties			
	level	Unknown	(F) ⇨
	pressure	Unknown	(F) ⇨

7. Leave all three editor windows open, and go to “Viewing Previously Created Rules” on page 59.

Viewing Rules and Objects

List Windows

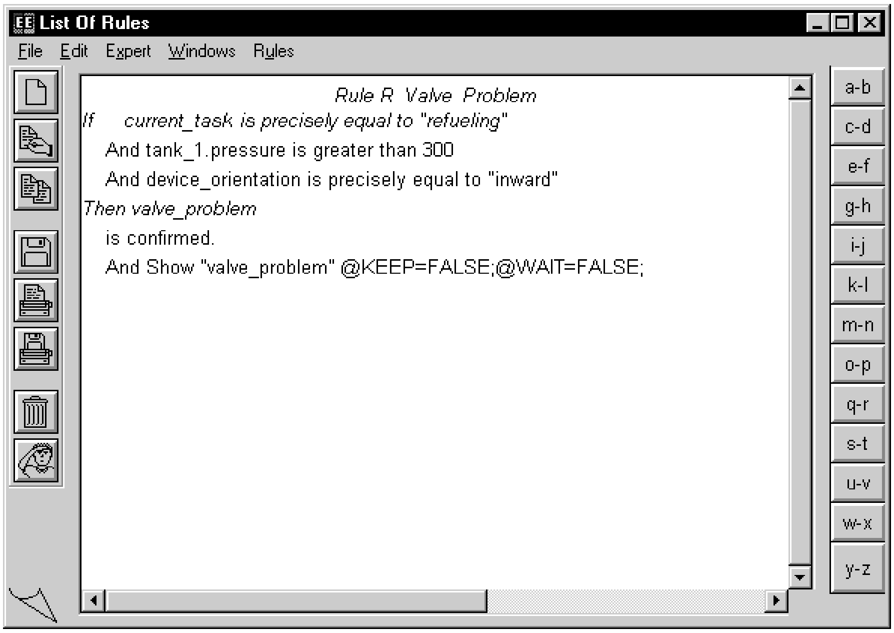
List windows in the Intelligent Rules Element provide a textual display of all the information in your application. Each of these has its own window:

- Rules
- Methods
- Hypotheses
- Data
- Classes
- Objects
- Properties

Each window contains a scrollable list of related structures arranged in alphabetical order. You can:

- Display List windows to examine the structures and their status
- Print them to obtain a record of the knowledge base.

Figure 4-26 shows the rule created in the previous section:



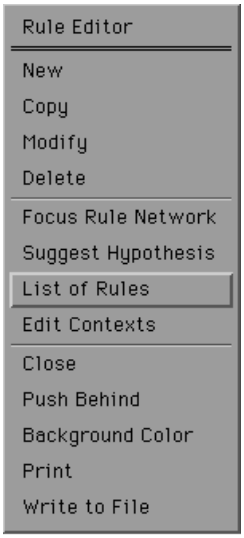
```
                    Rule R  Valve  Problem
If     current_task is precisely equal to "refueling"
   And tank_1.pressure is greater than 300
   And device_orientation is precisely equal to "inward"
Then valve_problem
   is confirmed.
   And Show "valve_problem" @KEEP=FALSE;@WAIT=FALSE;
```

Figure 4-26 List of Rules Window

Viewing Previously Created Rules

1. Click the still open Rule Editor to make it the frontmost window.
2. In the Rule Editor, click in an inactive area to display the global popup menu.

3. Choose **List of Rules**:



4. Click **Edit** to view the rule again in the Rule Editor.
5. Click **Cancel** to prevent the rule from changing.
6. Close the List of Rules.

Rule Network

The Rule Network window lets you view the relationships among the rules in your knowledge base. You can view:

- A single rule
- Specific groups of rules
- All the rules in the knowledge base.

Figure 4-27 shows the previously created rule in a Rule Network window. The rule graph shows each rule's:

- LHS conditions
- Hypothesis
- RHS actions (optional)

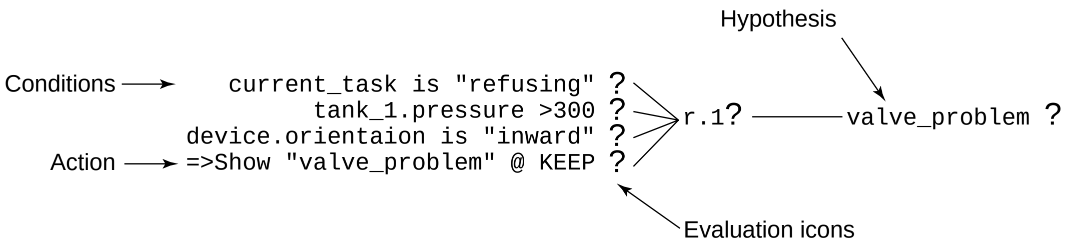

Hypothesis
Conditions
current_task is "refusing" ?
tank_1.pressure >300 ?
device.orientaion is "inward" ?
Action
=>Show "valve_problem" @ KEEP ?
r.1?
valve_problem ?
Evaluation icons

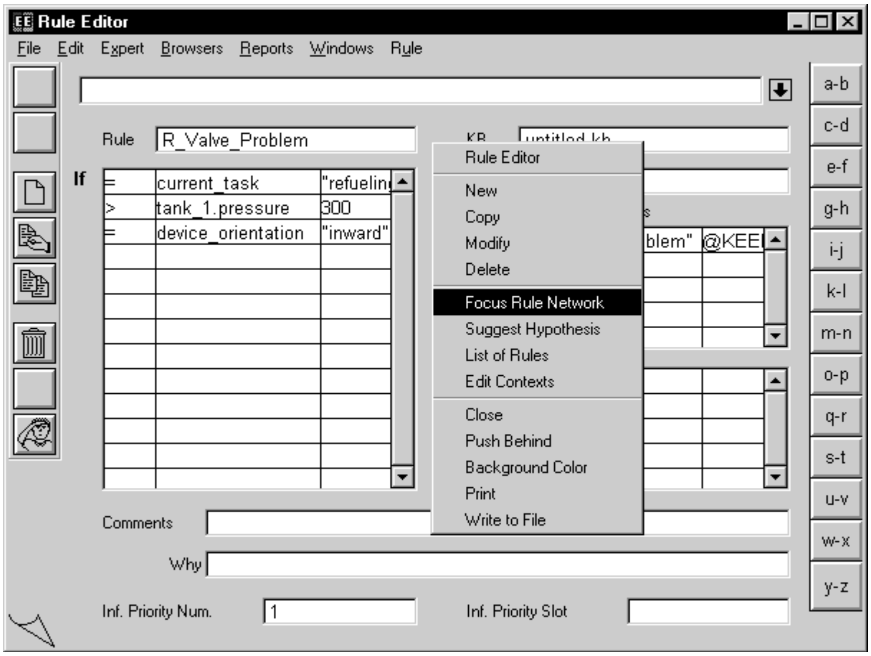
Figure 4-27 Rule Network Window Rule Graph

Note: The rule graph shows a question mark (?) to indicate the current evaluation status of each component. During knowledge processing, this symbol changes.

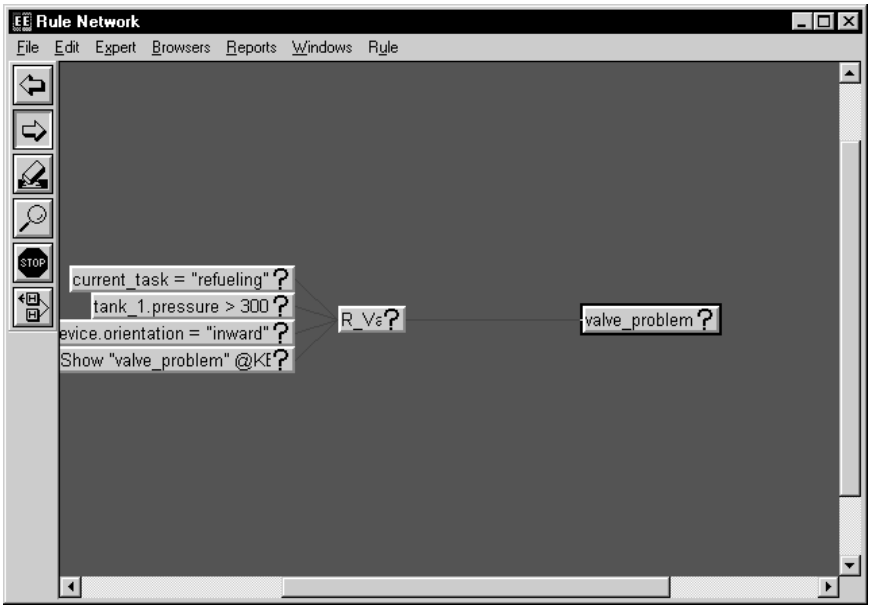
Displaying Rules in the Rule Network Window

1. Click the Rule Editor window, which is open.
2. In the Rule Editor, click in an inactive area to display the global popup menu.

3. Choose **Focus Rule Network**:



4. Leave the Rule Network window open, and go to “Displaying Class-Object Hierarchies in the Object Network” on page 63.


Rule Network
File Edit Expert Browsers Reports Windows Rule
STOP
current_task = "refueling"
tank_1.pressure > 300
evice.orientation = "inward"
Show "valve_problem" @KE
R_Va
valve_problem

Object Network

The Object Network window lets you view the class-object hierarchy in your knowledge base. You can view:

- An object
- Its properties
- The classes or other objects to which it belongs

Figure 4-28 shows how the Object Network window displays the class-object hierarchy for an object structure. The hierarchy includes:

- A class with its own property
- Two objects with properties attached

This simple hierarchy shows that the objects `tank_1` and `tank_2` actually inherited the property `level` from their parent class `Regular_Tanks`:

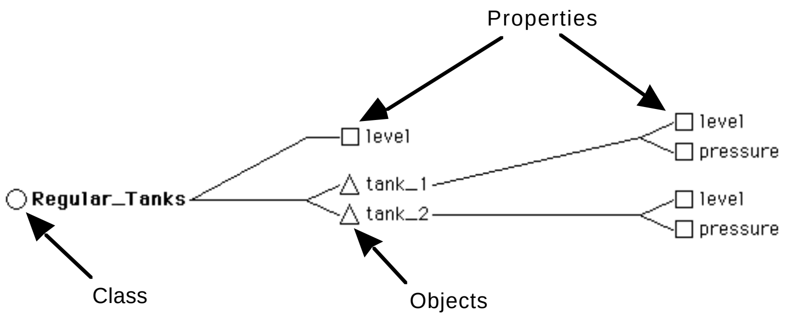
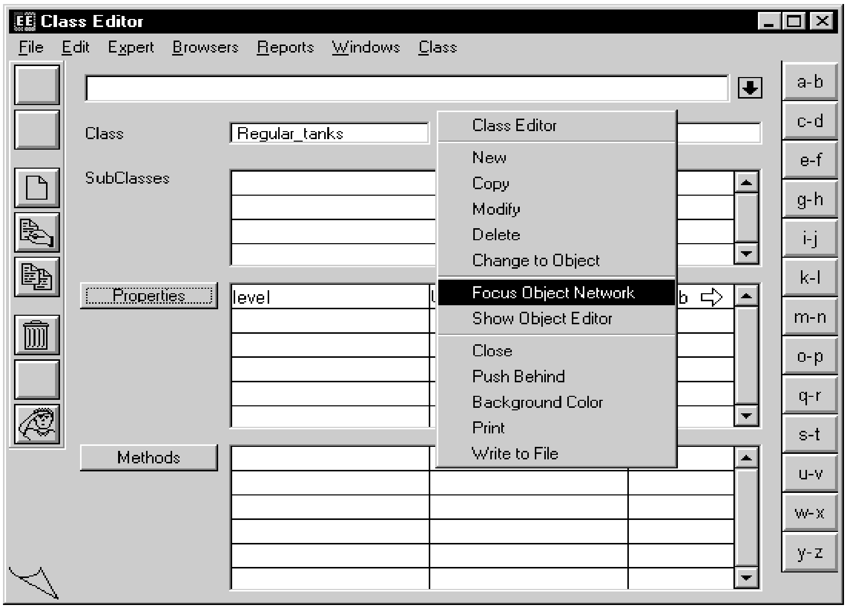


Figure 4-28 Network Window Class-Object Hierarchy

Displaying Class-Object Hierarchies in the Object Network

1. Choose **Windows - Class Editor**.
2. In the Class Editor window, click in an inactive area to display the global popup menu.

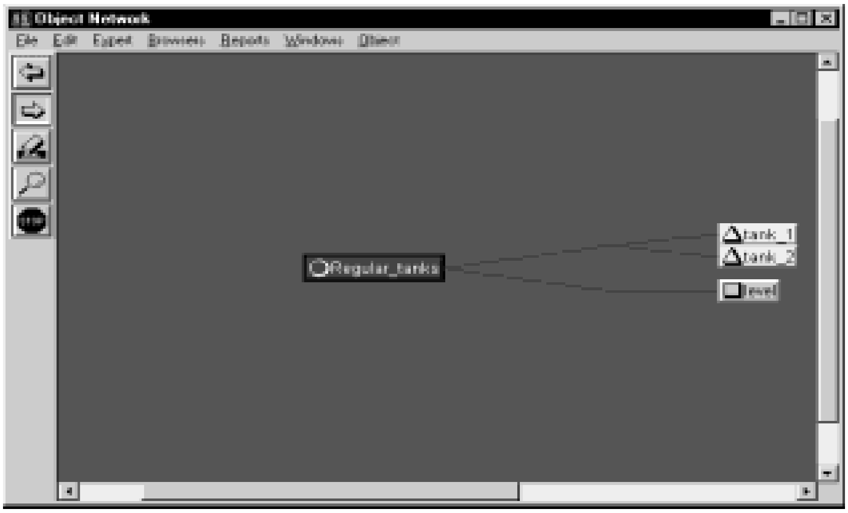
3. Choose **Focus Object Network**:



4. Position the cursor over the object `tank_1`, and click to expand the object-network diagram.

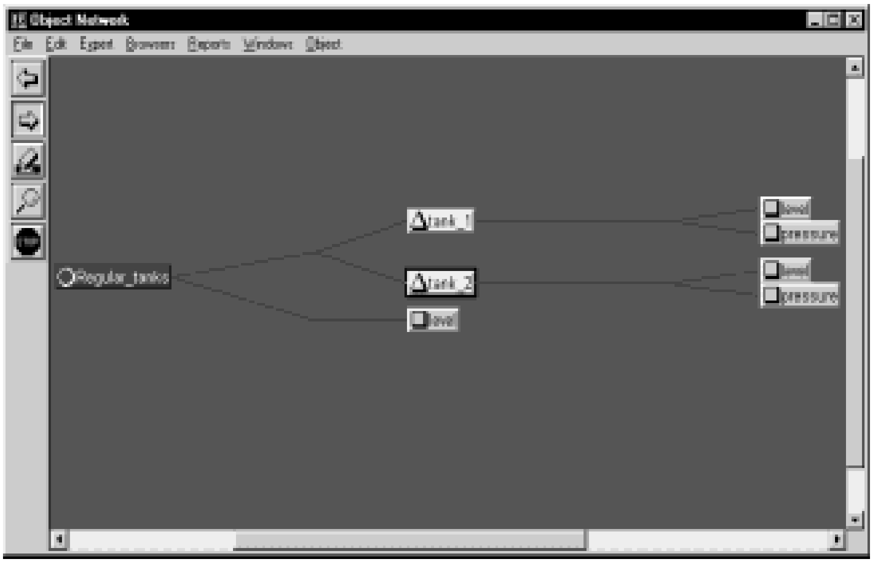
Note: The cursor changes to a right-pointing arrow when you place it on the object.

5. Repeat step 4 for the object `tank_2`:



6. Scroll the Object Network window to reposition the object-network diagram.

Note: The scroll arrows let you move the diagram left and right, as well as up and down.



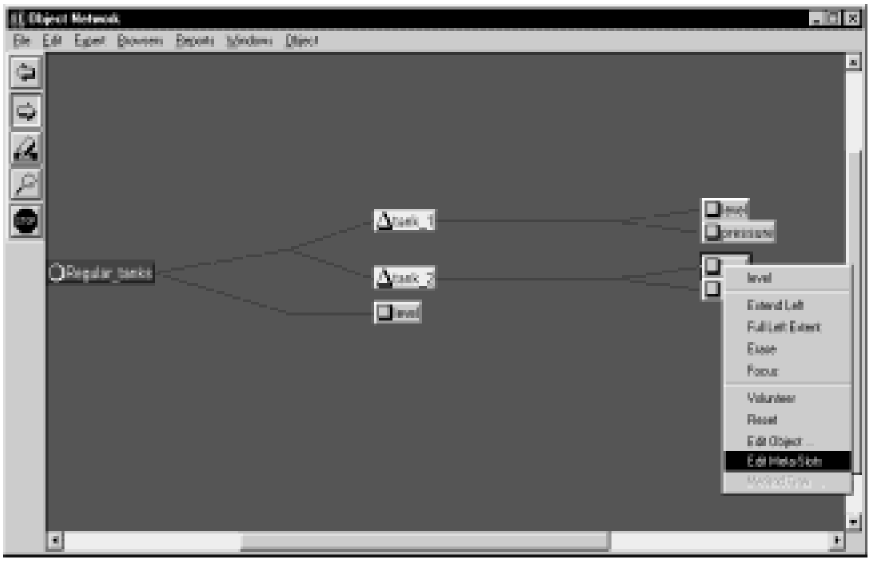
7. Click the property `level` for the object `tank_2` to display the local popup menu for this item.
8. Choose **Edit Meta-Slots**.

   The Meta-Slots Editor lets you control the attributes of specific slots in the knowledge base, including their:

   – inference priority

   – inheritance priority

– data-validation attributes



9. Close the Meta-Slot Editor.
10. Close the Object Network window, but leave the Rule Network window open.

Processing the Application

The Rule Network window is especially convenient for starting knowledge processing with a knowledge base loaded. Its special facilities give a graphic overview of the state of knowledge processing **while** inferencing is taking place. In the following demonstration, you will see that the graphic symbols of the rule graph change according to the status of the evaluation process. The system uses these graphic symbols—network icons—to represent the evaluation status of individual:

- Conditions
- Actions
- Rules

■ Hypotheses

Network Icon	Value/Meaning	Description
?	Unknown	In the initial state and not yet evaluated
✗	False	Evaluated by the system, but failed the test
✓	True	Evaluated by the system but passed the test
▭	Not Known	Evaluated by the system but found data to be insufficient for testing
●	Evoked hypothesis	Hypothesis currently under evaluation
⊙	Current condition or action	Condition or action under evaluation

In this chapter, the knowledge base consists of a single rule and the objects it reasons over.

These steps let you start knowledge processing with a knowledge base loaded:

1. Give the system a place to start (Suggest and / or Volunteer and / or Send a Message).
2. Run the session (Knowcess).
3. Set up the session to run again (Restart Session).

The following sections show a knowledge-processing session from the Rule Network window. Two separate procedures are given for the previously created rule:

1. You **suggest** the hypothesis `valve_problem`. This action places the hypothesis on the Intelligent Rules Element agenda for evaluation. The system proceeds by investigating the status of the rule's conditions.
2. You **volunteer** data that appears in a condition from the same rule. This action also places the hypothesis `valve_problem` on the agenda for evaluation.

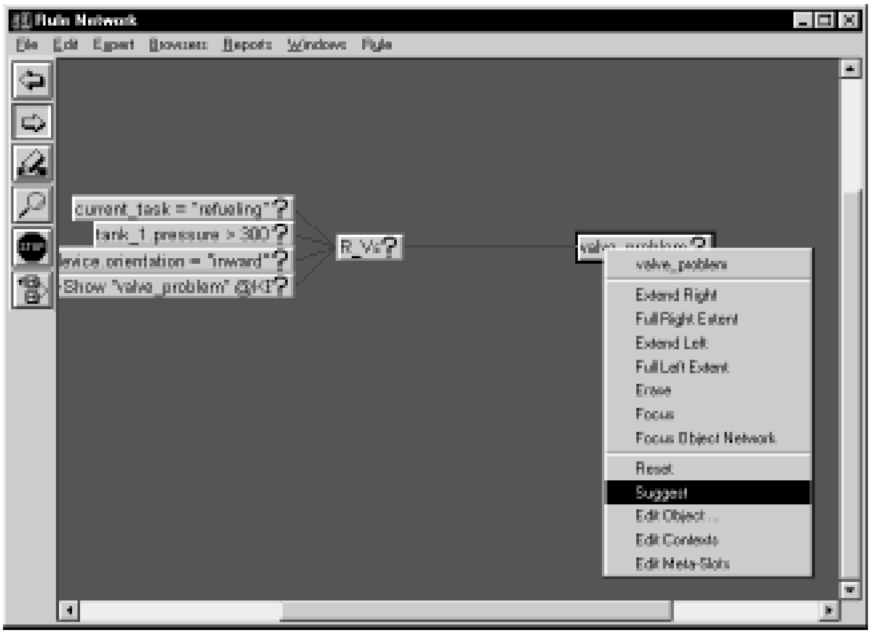
These actions—suggesting and volunteering the same rule—demonstrate the bidirectional nature of rules you build in the Intelligent Rules Element. It is not necessary to define the rule as one type or the other. Rule evaluation proceeds according to how knowledge processing was started.

Using Hypotheses

Hypotheses can be placed directly on the Intelligent Rules Element agenda for evaluation. This process is referred to as *suggesting an hypothesis*.

Suggesting an Hypothesis from the Rule Network Window

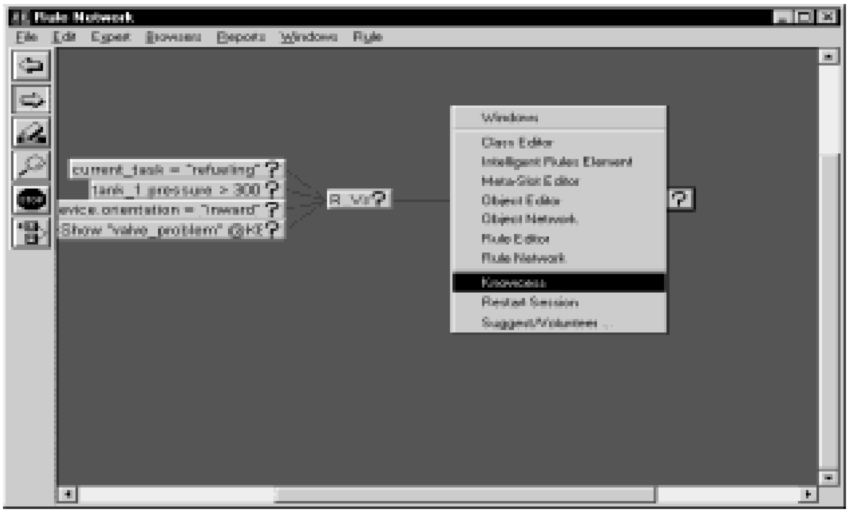
1. Click the Rule Network window, which is still open.
2. Click the hypothesis `valve_problem` to display the local popup menu for this item.
3. Choose **Suggest**:



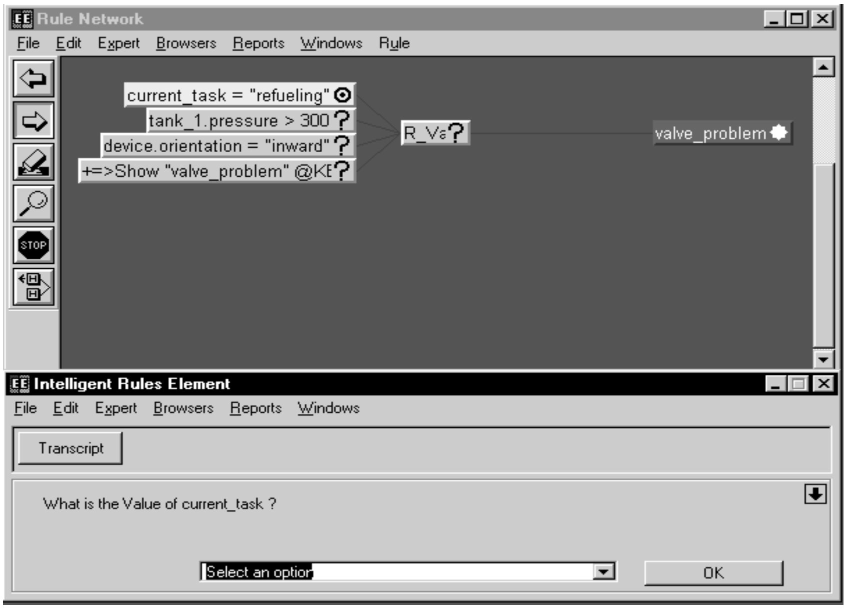
4. Click in an inactive area of the Rule Network window to display the window's popup menu.

   Note: If you are using a one-button mouse, use **Option + click**.

5. Choose **Knowcess**:

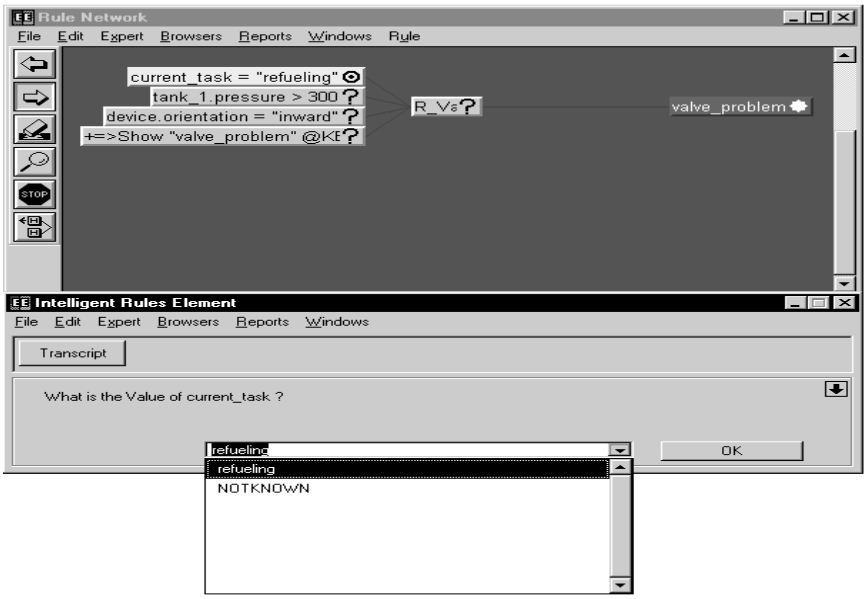


6. Click the up arrow on the top right of the main window to shrink it to the size of the Session Control panel.
7. Drag the main window below the graph of the Rule Network window, as shown here:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
STOP
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Select an option
OK

What is the Value of current_task ?

8. In the Session Control panel of the main window, click the edit-line arrow and select `"refueling"`:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
STOP
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
OK

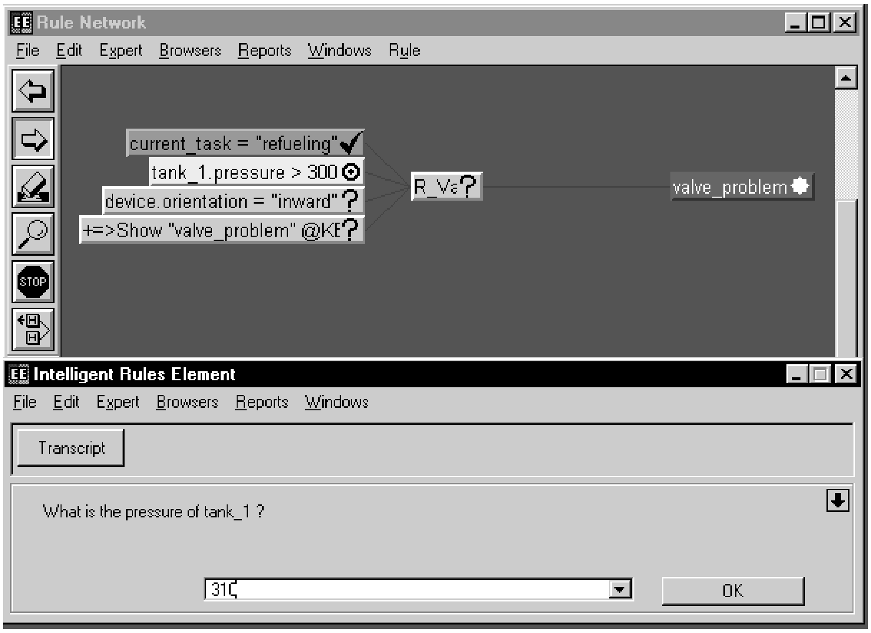
What is the Value of current_task ?

refueling
refueling
NOTKNOWN

9. Click **OK** to continue the inferencing session.

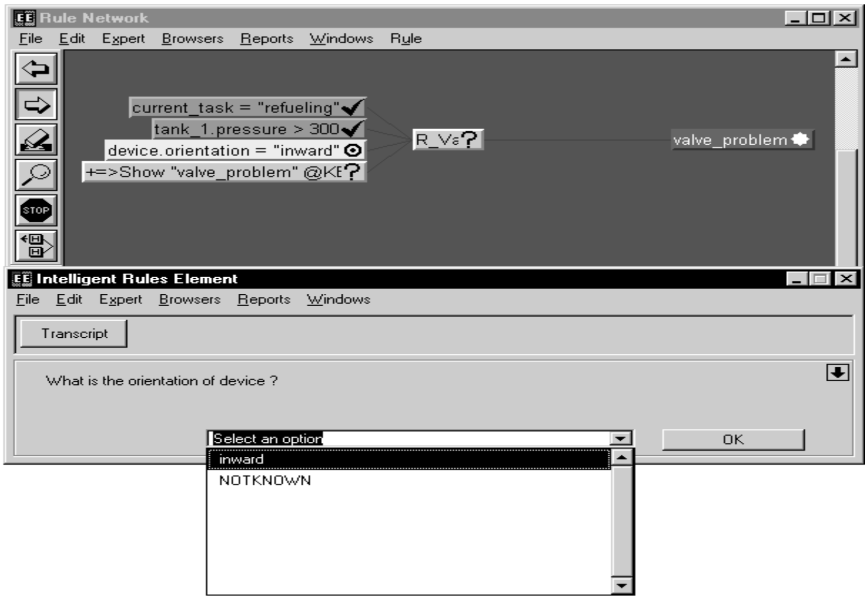
After evaluating the rule's first condition, the system displays the question in the Session Control panel for the second condition.

10. In the edit line of the Session Control panel in the main window, enter `"310"` and click **OK**.


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
What is the pressure of tank_1 ?
310
OK

After the evaluating the rule's second condition, the system displays the question for the third condition.

11. In the Session Control panel of the main window, click the edit-line arrow and select `"Inward"`:


Rule Network
File Edit Expert Browsers Reports Windows Rule
STOP
R_Va?
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Select an option
OK

current_task = "refueling"✓
tank_1.pressure > 300✓
device.orientation = "inward"
+=>Show "valve_problem" @KE?

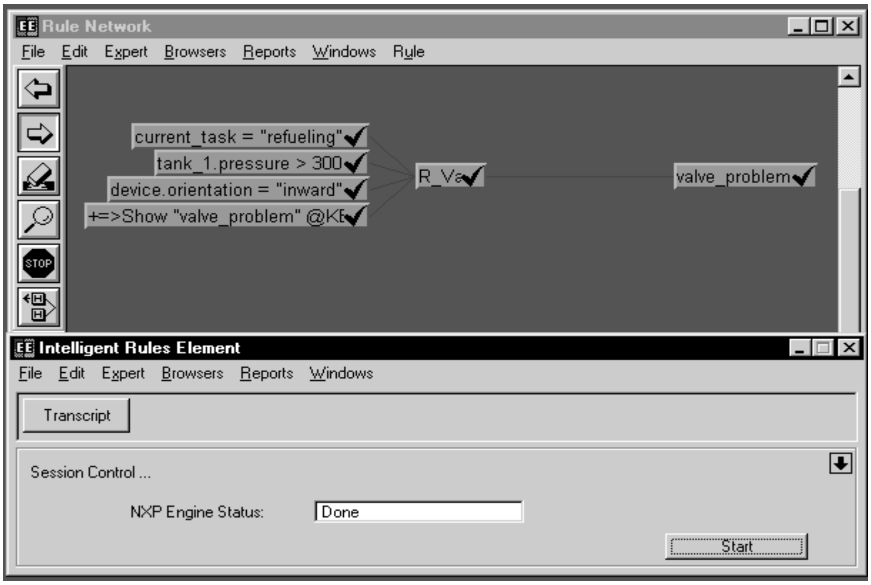
What is the orientation of device ?

inward
NOTKNOWN

12. Click **OK** to continue the inferencing session.

After the system finishes evaluating all three LHS conditions, it triggers the single action shown to the right of the arrow symbol (=>) in the rule graph. The newly displayed figure is a result of the rule’s action.

13. Close the Apropos window.

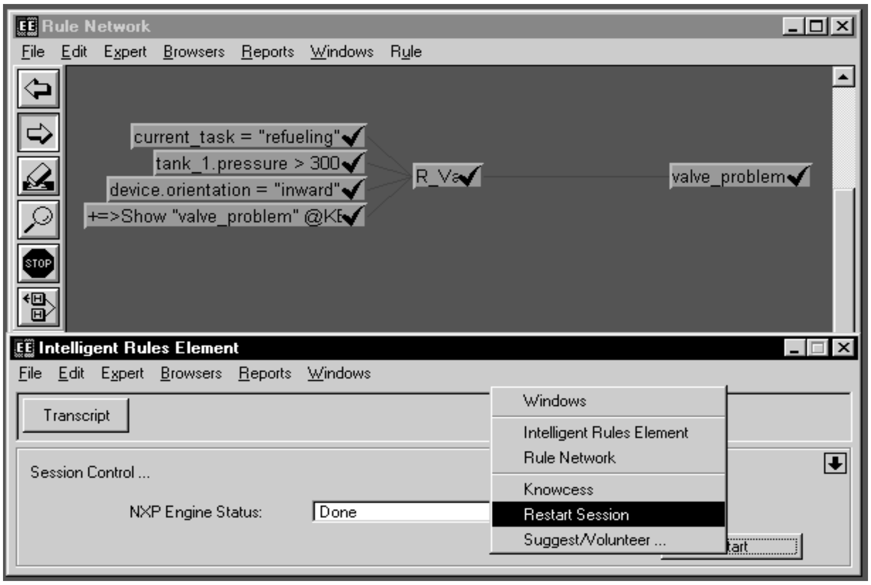

Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
Start

NXP Engine Status: Done

When processing is complete, the NXP Engine status shows “Done” in the Session Control panel of the main window. The check marks of the rule graph indicate the outcome of the evaluation process.

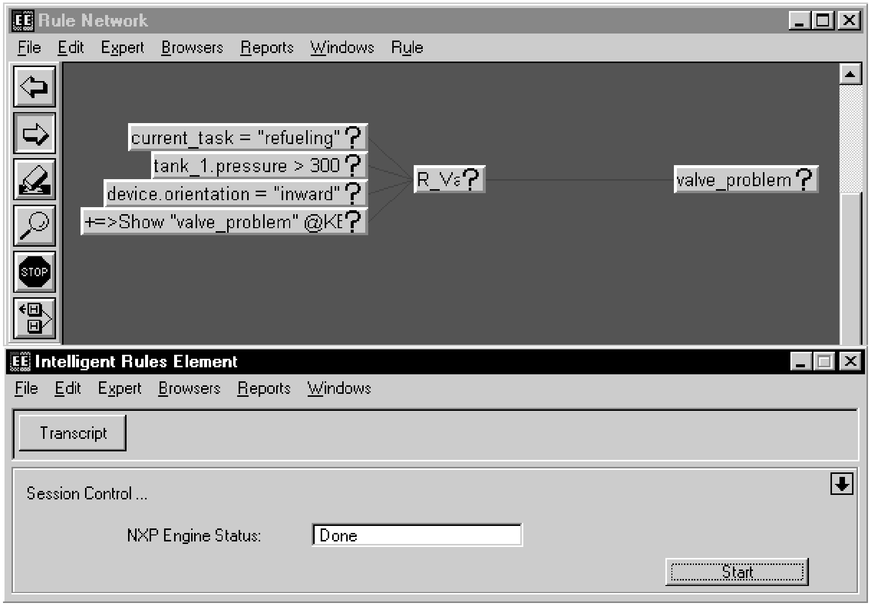
14. Click in an inactive area of the Session Control panel to display the global popup menu.

15. Choose **Restart Session**.


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
NXP Engine Status: Done
Windows
Intelligent Rules Element
Rule Network
Knowcess
Restart Session
Suggest/Volunteer ...
tart

The system returns the evaluation status of the rule graph to its original `Unknown` state, as indicated by the question marks.

16. Leave the Session Control panel and the Rule Network window open.

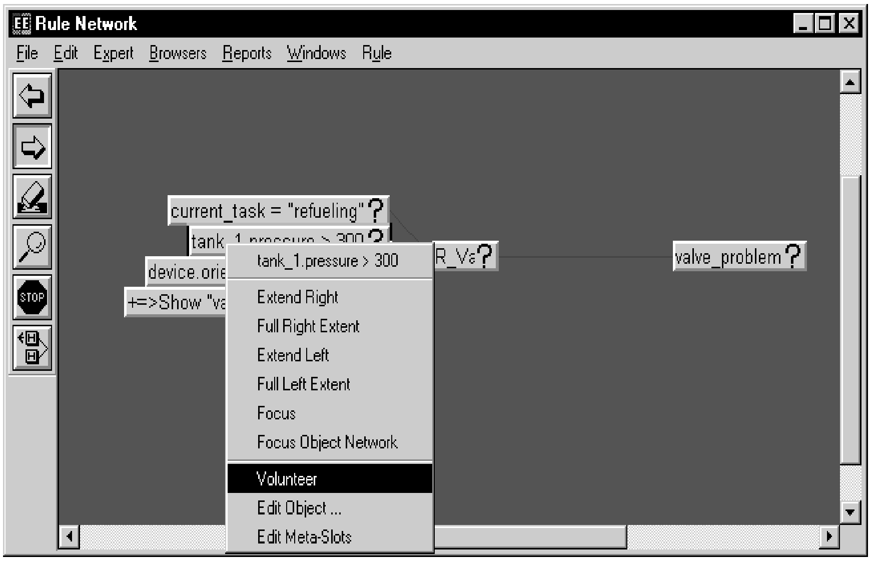

Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling" ?
tank_1.pressure > 300 ?
device.orientation = "inward" ?
+=>Show "valve_problem" @KE ?
R_Va ?
valve_problem ?
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
NXP Engine Status: Done
Start

Using Data

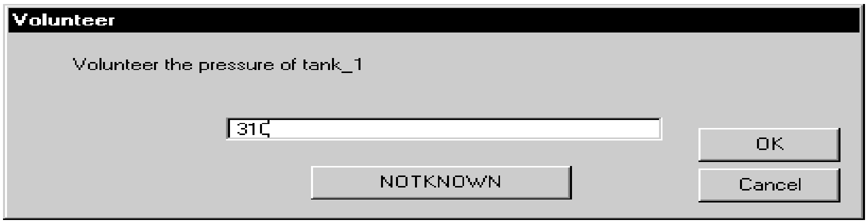
Data that causes the evaluation of one of the rule's LHS conditions can also place a hypotheses for evaluation on the Intelligent Rules Element's agenda. This action of starting knowledge processing with data is referred to as *volunteering data.*

Volunteering Data from the Rule Network Window

1. In the Rule Network window, click the slot `tank_1.pressure` in the rule's second condition todisplay the local popup menu for this item.
2. Choose **Volunteer**:

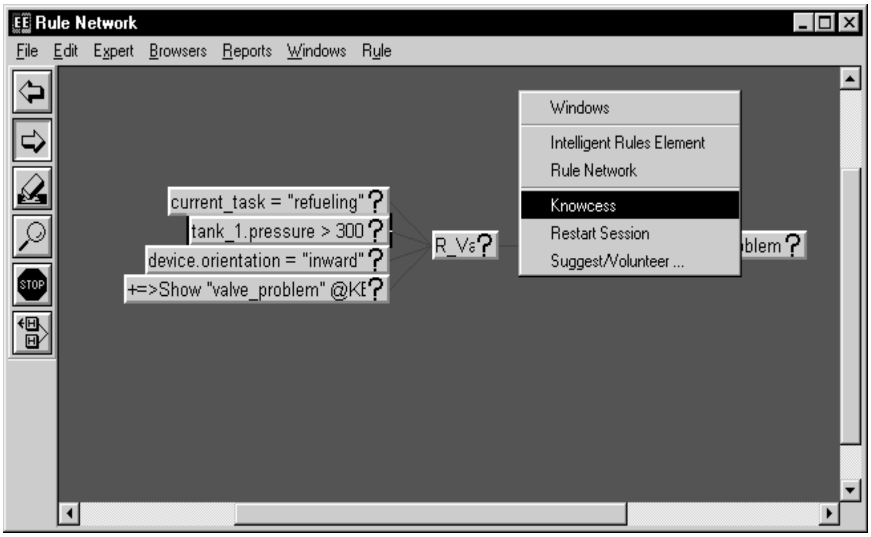


3. Enter `"310"` and click **OK**:

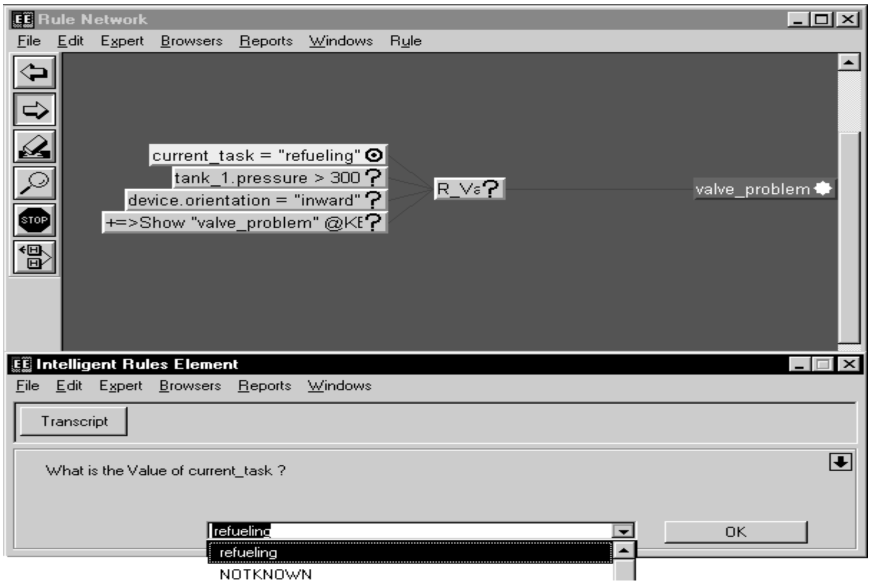


4. Click in an inactive area of the Rule Network window to display the window's popup menu.

5. Choose **Knowcess**:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
blem
Windows
Intelligent Rules Element
Rule Network
Knowcess
Restart Session
Suggest/Volunteer ...

6. In the Session Control panel, click the edit-line arrow and select `"refueling"`:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
OK
refueling
NOTKNOWN

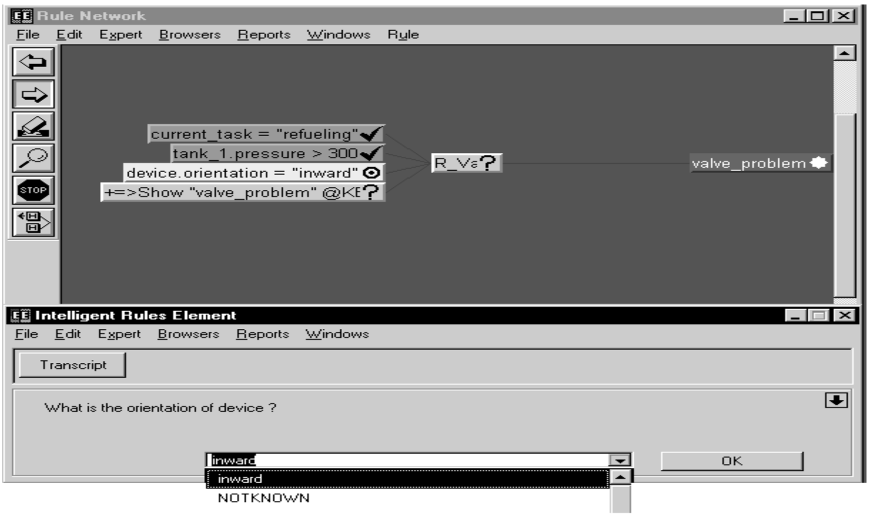
What is the Value of current_task ?

refueling

7. Click **OK**.

Since the value of the second condition is already known, the system proceeds to the next unknown condition.

8. In the Session Control panel, click the edit-line arrow and select `"Inward"`:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
inward
inward
NOTKNOWN
OK

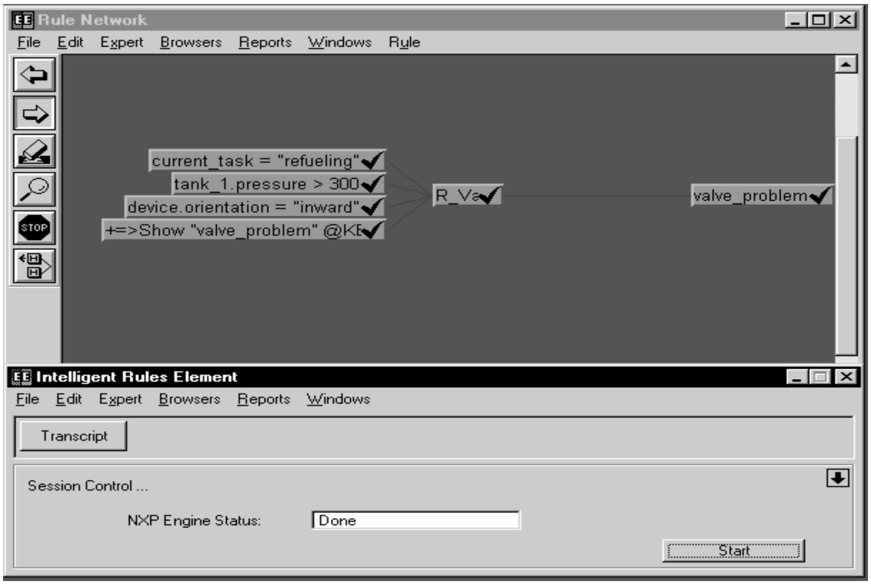
What is the orientation of device ?

9. Click **OK**.

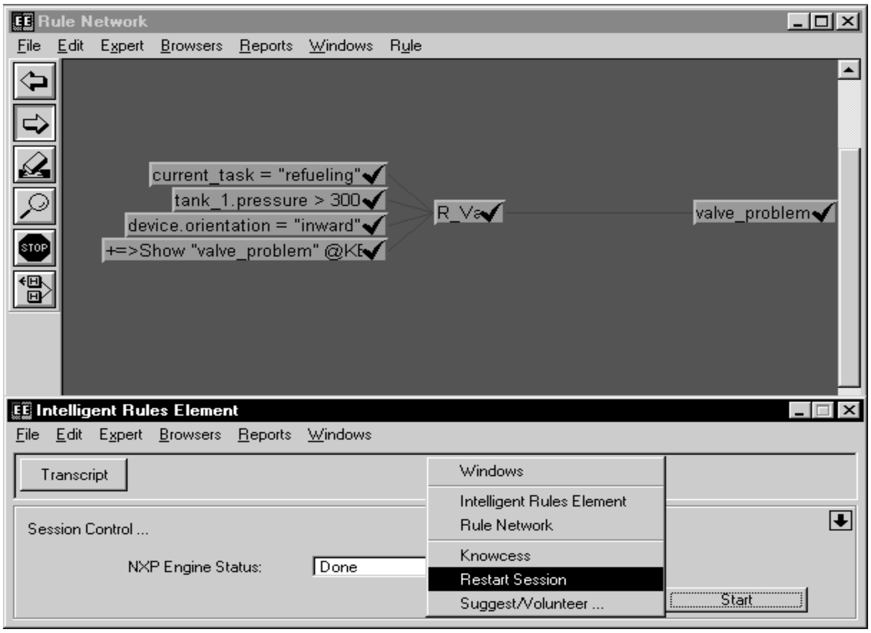
   After the system finishes evaluating the two unknown LHS conditions, it triggers the single action shown to the right of the arrow symbol (=>) in the rule graph. The newly displayed figure is the result of the rule's action.

10. Close the Apropos window.

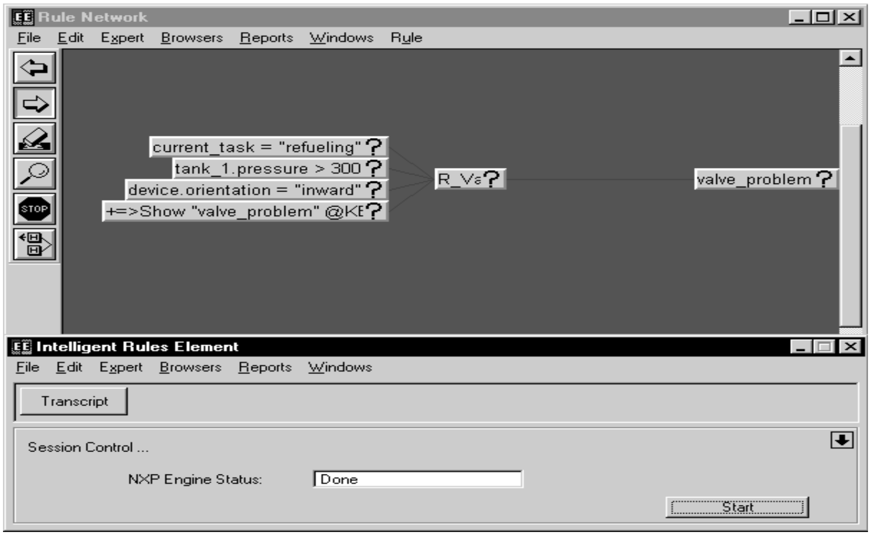
    When processing is complete, the Session Control Panel shows the status of the rules engine (NXP) as "Done." The check marks of the rule graph indicate the outcome of the evaluation process.


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
NXP Engine Status: Done
Start

11. Click in an inactive area of the Session Control panel to display the global popup menu.
12. Choose **Restart Session**:


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling"
tank_1.pressure > 300
device.orientation = "inward"
+=>Show "valve_problem" @KE
R_Va
valve_problem
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
NXP Engine Status: Done
Windows
Intelligent Rules Element
Rule Network
Knowcess
Restart Session
Suggest/Volunteer ...
Start

The system returns the evaluation status of the rule graph to its original `Unknown` state, as indicated by the question marks.


Rule Network
File Edit Expert Browsers Reports Windows Rule
current_task = "refueling" ?
tank_1.pressure > 300 ?
device.orientation = "inward" ?
+=>Show "valve_problem" @KE ?
R_Va ?
valve_problem ?
Intelligent Rules Element
File Edit Expert Browsers Reports Windows
Transcript
Session Control ...
Start

NXP Engine Status: Done

13. To conclude this session, close the Intelligent Rules Element windows.

For More Information about the Intelligent Rules Element

- See the *Intelligent Rules Element Language Programmer's Guide.*
- See the *Intelligent Rules Element Language Reference.*
- See the *Intelligent Rules Element User's Guide.*
- See the *Intelligent Rules Element C/C++ Programmer's Guide.*

Chapter

5 OLE Automation and OOScript

Object-Model Interoperability

The Elements Environment's built-in *interoperability layer* allows you to create enterprise-wide, business-critical applications by seamlessly integrating objects from different object models, such as the Elements Environment and Microsoft's Object Linking and Embedding (OLE) standard.

Note: The Elements Environment's interoperability layer currently supports the OLE object model for Microsoft Windows 3.1, Windows 95, and Windows NT. In addition, you can access CORBA servers using OLE-to-CORBA bridges such as Iona's Orbix.

The Elements Environment provides a set of *object servers* to access objects of the Neuron Data Elements through drivers to specific object models. The interoperability layer allows Neuron Data's OOScript language to access these types of objects through object servers:

- Objects internal to the Elements Environment, such as GUI, data-access, business-rule, and Web objects
- Objects external to the Elements Environment, such as Excel spreadsheet objects or other objects enabled for OLE automation

Note: You can also access the Neuron Data objects through these object servers by using other languages, such as Visual Basic, C, and C++.

OOScript

Neuron Data's *OOScript* is a powerful fourth-generation language (4GL) for scripting. OOScript allows you to quickly build business-critical applications. It can effectively integrate the Elements Environment objects and other external objects to create a comprehensive and interoperable development environment.

The object-aware, event-driven OOScript language combines the performance and functionality of a third-generation language (3GL) with the visual-editing techniques of a 4GL. Using the OOScript language, you can write scripts and then attach them to your application components (objects and modules). When system events affect your application objects, the attached scripts execute automatically in response.

Neuron Data provides these applications for building and editing scripts:

- A visual *Script Editor* that lets you build and edit scripts for different objects within the Elements Environment. You can start it from any graphical object or resource within the Elements Environment at any time.
- A visual *Script Browser* that allows you to locate and use external servers, classes, modules, script procedures, and global variables.
- A *Script Tracer* gives you a basic tool for analyzing an OOScript application as it executes.

For More Information about OLE Automation and OOScript

- See the *Interoperable Objects OLE Server Installation Note*.
- See the *OOScript Language Programmer's Guide* and the *OOScript Language Reference*.

Related Subjects

- The Elements Environment Application Services (EAS)

Chapter

6 *The Web Element*

The *Web Element* (WE) allows you to enable your applications for the World Wide Web. Besides being portable, the Web Element contains all the features that you can find in a Web browser.

The Web Element includes these components:

- Web Element browser
- Navigator Link and Navigator Overview
- Web Control
- Navigation API
- HTML Editor

The Web Element has these features:

- Is compatible with HTML 2.0
- Allows you to embed a World Wide Web browser window in an Elements Environment application
- Includes *Neuron Data Applets*, which allow you to create dynamic, distributed miniapplications that clients can download from anywhere in an enterprise-wide Intranet or from the Internet.

  Neuron Data Applets that use the OOScript servers have full access to external objects, such as those available in OLE automation servers.

The Web Element Pro contains C/C++ APIs that allow you to:

- Embed a Web Control in any window of an application
- Build widgets that are aware of Web Control

For More Information about the Web Element

- See the *Web Element Programming Guide*.
- For information on the Web Element OOScript classes, see the *OOScript Language Reference*.

Chapter 7

The Distributed Messaging Element

The *Distributed Messaging Element* (DME) lets you run multiple processes on several computers in a distributed system. It allows you to share information without having to worry about low-level communications. The Distributed Messaging Element provides these features for building distributed systems and applications:

- A comprehensive set of routines for building simple and complex distributed systems
- Tools to manage the distributed systems as if they were a single application running on a single processor

Applications can be

- Self-contained programs running on one or more processors
- Multiple threads of a single program

Communication between the processes can be simple sequencing routines within a single program. It can involve more complex processes, such as “interrupt-driven” real-time control and transmission of messages between completely independent programs with long latencies.

The Distributed Messaging Element handles data transfers between:

- Threads of a single process
- Processes on the same computer
- Processes on separate computers

For intraprocess communication, the Distributed Messaging Element can use a variety of mechanisms, including UNIX sockets and shared memory. For interprocessor transfers, the Distributed Messaging Element can use:

- Transport Communications Protocol / Internet Protocol (TCP / IP) sockets
- Serial connections (including modems)
- Other standard computer networks and protocols

Components of the Distributed Messaging Element

- A communications protocol and interface-definition language (IDL) that describe the format for sending and receiving data objects on the physical network

- Router processes that are responsible for routing the data objects to the appropriate processes
- A library of routines that implement the application programming interface (API) to the Distributed Messaging Element

The Distributed Messaging Element and Other Distributed Systems

The Distributed Messaging Element and other distributed data systems, such as distributed databases, differ in the following ways:

- The Distributed Messaging Element is a data-transmission mechanism. It distributes information to all appropriate processes and then completely erases the data from its own temporary buffers.

Note: The only exception to this is the persistent delivery information, which is kept at the sender and recipient ends, not in the infrastructure itself.

- Any new processes that attach to the Distributed Messaging Element cannot access previously distributed information. This is unlike other databases, which constantly update their files to retain a current version of all the data that they receive.
- The Distributed Messaging Element does not need to maintain data internally. Therefore, it does not require the complicated and time-consuming data-locking and cache-coherency mechanisms inherent in other distributed systems.

The Distributed Messaging Element Applications

The Distributed Messaging Element provides a comprehensive foundation for distributed applications. It supports event-driven, peer-to-peer applications as well as client-server applications. The Distributed Messaging Element is distinctive in its real-time, dynamic control capabilities. Application areas for the Distributed Messaging Element include:

- Distributed manufacturing systems
- Real-time data distribution and distributed databases
- Collaborative workgroup and workflow software
- Distributed control systems
- Networked multimedia applications
- Accounting applications, such as order processing and inventory control
- Integrated logistics
- Distributed decision-support systems.
- Migration of applications to new distributed environments

Concepts of the Distributed Messaging Element

The following sections describe concepts underlying the Distributed Messaging Element, including:

- Data objects
- The data-distribution method used to send data objects
- The event-driven model of programming

Portability

The Distributed Messaging Element software is designed to be easily portable across different:

- Computer architectures
- Operating systems
- Programming languages

Different architectures and operating systems can connect to the same virtual network to send and receive information from any other process. The Distributed Messaging Element handles all data translation required for communication between processes.

Data-centered Processing

The Distributed Messaging Element uses a data-centered approach, in which processes that exchange data indicate the type of data they exchange, and not their addresses or process IDs. In a data-centered approach, processes can be replaced, run on different processors, or simulated without affecting any of the other processes. This makes modular systems possible. Since all processes can access all data, debugging communications becomes easier than in an address-based system.

An example of a data-centered system is a distributed database, in which processes exchange information by reading from, and writing to, database entries. Each process accesses the database through a local interface. The processes have no prior knowledge of the other processes accessing the database.

Data Object and Structure

Since the Distributed Messaging Element is a data-centered system, the organization of data in the Distributed Messaging Element is very important. The Distributed Messaging Element uses *structures* to define the format of each class of data. All processes use the same format for the structures to ensure a consistent data format.

Each piece of information exchanged between processes is referred to as a *data object*, which contains these fields:

- The data itself
- Information about the process that created the data
- The time the data was created
- Information to uniquely identify the structure of the data

The Distributed Messaging Element provides various routines for handling all this information as a unit.

The Distributed Messaging Element provides mechanisms for defining new classes of data objects in terms of the data structures they contain. Once a data-object class is defined, applications can:

- Create instances of this object
- Fill them with information
- Exchange them with other applications

Note: In this manual, the term “object” refers to a data object.

Routers and Fully Connected Networks

The Distributed Messaging Element implements a ‘‘virtual’’ fully connected network, where all processes are connected to all other processes. Any process can read information from, and write information to, any other process.

Most implementations of such a network are based on the point-to-point communication model. Each process in the network has an open file descriptor for every other process and keeps track of its connections with all other processes.

However, the Distributed Messaging Element implements a multipoint method by creating a process called the *router*, which is responsible for these tasks:

- Maintaining all the necessary connections
- Automatically routing information among various processes
- Exchanging information between processes on its processor as well as on different processors

The user program has a single connection with its local Distributed Messaging Element router. It connects to other processes through the Distributed Messaging Element router. This simplification not only reduces the complexity of the user code, but also improves performance by optimizing the data distribution.

This architecture is *natively multipoint,* which means that messages can have more than one sender and receiver. Therefore, the Distributed Messaging Element is more scalable and flexible than point-to-point communication models.

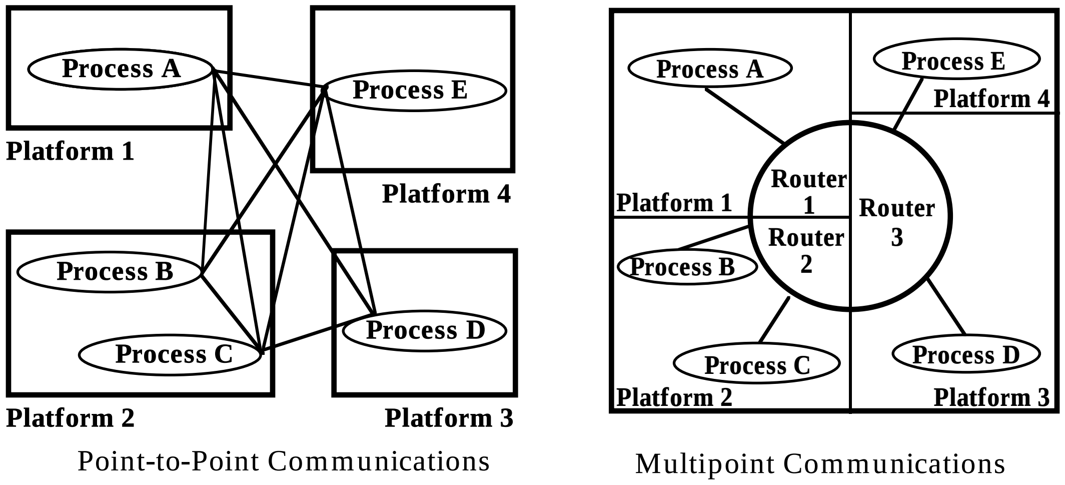


Figure 7-1 Multipoint Routing versus Point-to-Point Communications

Data Distribution

Using the data-centered approach and routers, the Distributed Messaging Element has these mechanisms for exchanging data of a particular class:

- A process indicates what classes of data it needs by “registering an interest’’ in the corresponding data format.

Note: A process registering interest in a particular class of data is equivalent to saying: ‘‘Whenever data of class X is available, no matter who created it, I want a copy.’’

- When the particular class of data is available, the Distributed Messaging Element router distributes the data to all processes that registered an interest in that class of data, but not to processes not interested in that class of data.

A process does not have to know who needs the data. The process just communicates to its router to distribute a copy of the data to any process that wants it. The Distributed Messaging Element router translates this command into a specific set of transmissions that depend on the interests of all processes.

Event-driven Programming

The Distributed Messaging Element API uses an *event-driven* model of programming, in which you define a set of conditions that may become true at some point in the execution of an application. An *event* occurs whenever a condition becomes true.

Examples of events in a Distributed Messaging Element application include:

- The receipt of data from another application
- Timers
- Keyboard input
- File input/output (I/O)

The Distributed Messaging Element provides mechanisms for indicating an "interest" in each kind of event. You have to specify an event handler to be invoked whenever the specified event occurs.

Interactions with Applications

An *application* is a piece of code that can be uniquely identified and uniquely addressed. An application usually is a separate executable, but it can also be

- A thread within an executable
- A dynamic link library (DLL)
- A separate virtual "environment" within an executable

The Distributed Messaging Element can interact with applications in the following ways:

- Seeing the applications running
- Starting and stopping applications
- Identifying the application that generated a piece of data
- Sending a piece of data to an application

The Distributed Messaging Element provides routines for these functions:

- Initializing a new application
- Exiting an application (not the whole executable)
- Controlling the behavior of an application, such as reloading persistent information

Resource Files

Resource files store, in a common format and directory, the following configuration information. The Distributed Messaging Element components require this information at various times during the execution of a distributed application:

- Fields contained in each kind of data object
- Default parameters for starting an application
- Information about which remote computers to connect to
- The current configuration of the applications
- Data that the applications have received or sent

Resource files provide a very simple file-based database. Each resource file contains one or more *records* of information. These records are instances of objects stored in a file.

The Distributed Messaging Element provides tools for reading, writing, and updating objects in these resource files. In addition, various Distributed Messaging Element tools use the resource files to store information such as:

- Object definitions
- Connection configuration
- Startup defaults

Standard Object Library

The *standard object library* contains a set of standard object definitions and routines that provide default capabilities for all the Distributed Messaging Element applications. This library includes several ‘‘utility packages’’ of objects or routines. Each package provides a service. It also contains one or more object definitions, and routines for interpreting and exchanging those objects.

Capabilities of these utility packages include:

- Monitoring the application states of all systems on the network
- Monitoring configuration information, including routers
- Starting and stopping remote applications
- Automatic maintenance of connections

For More Information about the Distributed Messaging Element

- See the *Distributed Messaging Element Programmer’s Guide.*
- See the C *API Reference Manual.*

Chapter 8

The Elements Application Services

To reduce development time, the *Elements Application Services* (EAS) provides the support layer for these low-level, platform-specific functions:

- Built-in memory and print management
- Graphic primitives
- Error handling
- File input and output
- Asychnronous event management
- String manipulation

The Elements Application Services makes the graphical-presentation layers and the integration layers of an application portable.

For high-level application development, the Elements Application Services provides complex building blocks, which you can use to assemble your application. With building-block mechanisms, such as datasource / views**,** you can avoid repetitive coding tasks related to these functions:

- Manipulation of data and sources of data
- Display of data for complex widgets such as tables and list boxes

Datasource/Views

The *datasource/views* mechanism provides the underlying bidirectional protocols linking datasources and views for scripts written in OOScript, C, and C++. By using this mechanism, you avoid having to write code to directly manipulate widgets.

Datasource / views separately addresses:

- Data management
- Application logic
- GUI management

The datasource / views mechanism allows you to concurrently access the same data, such as information from a database, and to present it in multiple views—for example:

- A spreadsheet-like table

- A choice box
- An input field

The data can then be modified at either end of the link. When the datasource is modified, it can automatically update all the views, depending on the options present in the registered view. The datasource also controls access to prevent conflicts among multiple views during simultaneous changes.

Internationalization

The Elements Application Services also provides the underlying support for internationalization. With the Elements Application Services, you can quickly port applications to several single-byte or double-byte languages, including Japanese. The internationalization features include:

- Character sets
- Porting support
- Text rendering
- In-place editing
- Standard or native in-text widgets
- String-manipulation services

See Chapter 12, “Localizing Applications in the Elements Environment,” for more information about internationalization.

For More Information about the Elements Application Services

- See the *Elements Application Services C/C++ Programmer’s Guide*.

Chapter

9 C++ Programming in the Elements Environment

The Elements Environment for C++ consists of:

- A set of C++ libraries
- A set of resource files
- A development environment

The Elements Environment libraries themselves have been compiled in C++. This implementation of the product achieves a high level of object orientation, including:

- Subclassing
- Virtuality
- Templates
- Exception handling
- Generic classes

The Elements Environment C++ development environment includes these features:

- The C++ classes implement the same set of API calls provided in the C libraries.
- Naming conventions in the C++ API are similar to those in the C API.
- Virtual member functions implement the notification mechanism used in C to achieve virtuality.
- There is a set of C++-specific constructors for each of the C++ classes. Many of these constructors are overloaded for specific processes. Some of the classes also have specific destructors.
- The classes inherit directly from each other; there are no intermediate implementation classes.

General Architecture

The Elements Environment C++ libraries include:

- Resource classes
- Utility classes

Resource Classes

In the Elements Environment, the **base class** for objects that can be persistent and that support instance customization is the Neuron Data Resource (**NDRes**) class. All subclasses of **NDRes** inherit directly from each other in a single inheritance graph. These *resource classes* have part of their interface defined as virtual member functions. You can reimplement these in subclasses to customize their behaviors.

Utility Classes

Utility classes are a group of C++ classes that do **not** derive from the **NDRes** class. These objects:

- Do not support persistence or instance customization
- From a C++ perspective, do not offer any virtual API functions

All these classes are defined as standard C++ classes. All the API functions are defined as:

- Member functions
- Static member functions

Utility classes provide this functionality:

- Memory management
- Memory pools
- Storage objects (such as arrays, hash tables, and AVL trees)
- Buffered input/output (I/O)
- File I/O
- File management
- Compression and decompression
- Encryption
- String management
- Time management

In the case of arrays, several generic classes are defined according to whether the data should be stored directly or by reference.

Utility classes provide much of the functionality of the standard C++ library functions. In addition, they ensure that the applications you develop are compatible across platforms.

Constructors and Destructors

Each class in the Elements Environment C++ library defines:

- Overloaded versions of the constructor `new`
- The C++ destructor `delete`

C++ Constructor new and Destructor delete

The **NDRes** class redefines the C++ `new` and `delete` operators as calls to Elements Environment memory management. This allows you to create and delete objects with the familiar C++ `new` and `delete` operators. When you use these constructors to subclass from the base **NDRes** class, they allow you to construct an object from information persistently stored in the resource manager.

For example, to construct a window `win` from the information stored in the module `mod` and resource `win` in a loaded resource library, you can use:

```
NDWinPtr win = new NDWin("mod", "win");
```

To create a window and a button in that window, you can use:

```
NDWinPtr win = new NDWin;
      win->Init();
NDPButPtr pbut = new NDPBut;
      win->AddWgt(pbut);
```

You can remove the button from the window with:

```
win->RemoveWgt(pbut);
delete pbut;
```

Note: Usually, you do not delete widgets one by one. Instead, when users exit a window, the window manager deletes all the widgets. You delete windows either by using the window manager of the interface or by this explicit call:

```
win->Terminate();
```

The Elements Environment Constructor new

The Elements Environment constructor `new` allows advanced Elements Environment applications to subclass an object at runtime. This constructor takes a third argument, `&array`. This lets you specify, at construction time, the instantiation class for each of the objects that gets created.

For example, this code will instantiate a button in the window for the subclass **MyTBut** of **NDTBut**:

```
ResRunTimeClassArray array;
ResRunTimeClassRec info;
info.ResName = "PButOk";
info.ResClass = MyTBut::Class();
array.AppendElt(&info);

NDWinPtr win = new NDWin("mod", "win", &array);
```

Note: When you use this runtime subclassing mechanism, make sure that the objects you want to instantiate are a subclass of the stored object.

Encapsulation

The Elements Environment widget classes use the C++ encapsulation mechanisms in a simple way:

- The class-specific fields are **protected** and therefore cannot be directly accessed; however, they still permit subclassing.
- The member functions are **public** and appear to be directly accessible.

A large number of field names, such as `TButPrivate1` and `TButPrivate2`, are encoded. Therefore, even if the corresponding fields are in the public section, they behave as if they were private because their true names are concealed.

You can access protected fields of classes only by making a call to the appropriate member function. In particular, you cannot use certain functions as freely as in C. Instead of passing pointers to the resource, you use member functions of the instantiated objects. For example, this function call does not compile in C++:

```
WgtPtr Wgt;
WGT_GetFgColor(wgt);
```

The correct implementation is

```
wgt->GetFgColor();
```

Customization

The Elements Environment allows you to **customize** the behavior of objects in classes inheriting from **NDRes** at two levels:

- Class
- Instance

Class-Level Customization

To customize the behavior of objects at the **class level**, you can reimplement virtual member functions in subclasses. This is the standard C++ mechanism for customization.

For example, if you want to replace the redraw method for your own subclass of **NDPBut**, you can redefine the **NfyRedraw** virtual member function for the subclass:

```
class MyPBut : public NDPBut {
	RCLAS_CPPFULL(MyPBut, NDPBut)
	void NfyRedraw(void);
};

void MyPBut::NfyRedraw(void)
{
	// default drawing
	NDPBut::NfyRedraw();
	// custom draw in addition
	...
}
```

Use this mechanism when you need to customize a significant number of instances.

Instance-Level Customization

You can customize the behavior of any **instance** of a given class without affecting the behavior of all the other instances.

Instance-level customization is most useful in specialized applications in which customization affects only a single instance. For example, the Elements Environment code generator uses this mechanism to ease compilation.

You can achieve instance-level customization by registering notification handlers to override the default behavior that the corresponding virtual member functions implement.

For each virtual member of any class, there is a notification that identifies the callback. For example, the **NfyHit** virtual member function of the **NDTBut** class is identified by `TBUT_NFYHIT`.

You can register a callback by using the macros provided in the public header file respub.h**.** These macros allow you to register member functions of any particular class as callbacks.

This example shows how to customize the class **Form** at the instance level:

1. Create the class:

```
class Form : public NDWin {
protected:
      NDPButPtrmPButOk;
      NDLBoxPtrmLBoxData;
public:
      Form();
      ~Form() { }
      RES_NFYVOIDHANDLER(Form, NfyHitOk)// declares callback
}
```

2. Set the handler in the constructor for the class:

```
Form::Form() : NDWin("mod", "win")
{
      mPButOk = (NDPButPtr)GetNamedWgt("OK");
      mLBoxData = (NDLBoxPtr)GetNamedWgt("Data");
      RES_SETNFYVOIDHANDLER(Form, mPButOk,
                             TBUT_NFYHIT,NfyHitOk);
}
```

3. Implement the handler:

```
void Form::NfyHitOk(void)
{
      NDWin::Terminate();
}
```

You can have the handler call the default action for the notification before or after it does its custom processing. You can do this by calling the **DefNfy** method if the **ND<Class>** pointer for the object is available:

```
void Form::NfyHit(void)
{
      mPButOk->DefNfy(TBUT_NFYHIT); // Call default action.
      Terminate();
}
```

Note: You have to call the **DefNfy** method. If you call the **NfyHit** method, it will call **Form::NfyHit** again and cause an infinite loop.

There is a **RES_SHAREDNFYVOIDHANDLER** version of the macro. This macro passes the `ResPtr` of the object that triggered the notification as a parameter to the handler. However, you need to know the actual type of the object so that you can cast it from `ResPtr` to `<Class>Ptr` to call the correct default action for **<Class>**.

Subclassing in C++

The Elements Environment exports its entire API as C++ classes. Except for the restriction on implementation classes described in “Utility Classes” on page 96, you can **subclass** any of the classes.

Subclassing from NDRes Subclasses

The Elements Environment **NDRes** subclasses have to interact with the persistency manager. Therefore, subclassing of **NDRes** is different from subclassing in C++:

- The **NDRes** subclass objects are **not** always created and destroyed explicitly through the C++ `new` and `delete` operators. Instead, in most Elements Environment-based applications:
  - The resource manager automatically creates widgets when a window is loaded from the resource database.
  - The resource manager automatically destroys widgets when a window is terminated. This can happen as a result of a user action from the System menu or because of an explicit call to **NDWin::Terminate()**.

  To maintain this capability, you have to register the C++ classes in the resource manager. That way, it will know how to create and destroy instances of these classes. Also, you must provide information about your widget’s persistent fields so that the Elements Environment resource manager can manage these fields for you.

  The Elements Environment API provides mechanisms to support these different subclassing needs:
  - *Light C++ subclassing*, in which the subclass need not directly interact with the resource manager. See “Light Subclassing” on page 102 for details.
  - *Full C++ subclassing*, in which the subclass can define new fields, virtual functions, constructors, destructors, and persistent fields. See “Full Subclassing” on page 102 for details.
- **NDWin** subclass objects are not always propagated to the instances of the class. To avoid this problem, the Elements Environment provides template classes. *Template classes* allow widgets defined in a template for a window subclass to be automatically inherited by:
  - All the instances of the subclass
  - The templates
  - Instances of derived subclasses

When you use the template class, the system maintains index values for widgets.

Note: You cannot select inherited widgets in the Window Editor. Their names will be followed by an “L” in the widget list to indicate that they are locked. You can only edit these widgets in the template itself.

Light Subclassing

You can use light subclassing when you do not want the persistence manger to create or dispose of instances of the resulting class. In that case, the persistence manager does not have to know anything about the subclass in addition to what it already knows about the superclasses. You have to create and dispose of all instances. They are never stored in the resource files directly.

Note: Even though instances of a class are not stored in a resource-library file, they can be constructed from the information stored for an instance of a superclass. For example, suppose that `mod.win` is a window in a .dat file. Then, this constructor will construct the instance **MyWin** from the information for `mod.win`:

```
MyWin::MyWin() : NDWin("mod", "win")
{
    .../...
}
```

Full Subclassing

Full subclassing is used when you want to define persistent fields in your C++ class.

The Elements Environment resource manager handles these persistent fields and automatically loads them from a resource database (.dat file). You can edit them:

- In text form in the resource (.rc) file
- Through an editor that you display from the Resource Browser

Conceptually, there is no major difference between full subclassing in C++ and resource subclassing in C. The main difference is that you must register the class through a call to **RCLAS_CPlusRegister** instead of a call to **RCLAS_Register**.

To subclass a resource class:

1. Place the **RCLAS_CPPFULL** macro in your subclass definition.
2. Implement the two constructors required, as well as the destructor.

3. Initialize the `OiFields` static variable, which is returned by the **GetOiFields()** function, using one of these methods:
   - Statically initialize the variable with the array of persistent field descriptors:

   ```
   PFldPtr MyClass::OiFields[] = {
      { "Field1", RCLAS_OFFSET(MyClass, Field1),
   PFLD_TYPESTR,
         PFLD_CATTEXT },
      { "Field2", RCLAS_OFFSET(MyClass, Field2),
   PFLD_TYPEINT16,
         PFLD_CATSIZE },
      { NULL, 0, PFLD_TYPEBAD, PFLD_CATNONE }
   };
   ```
   - If your compiler does not let you statically initialize the variable, compute the offsets before registering the class:

   ```
   MyClass * n = NULL;
   OiFields[0].Offset = RES_OFFSETOF(n, Field1);
   OiFields[1].Offset = RES_OFFSETOF(n, Field2);
   Register();
   ```
4. By calling **MyClass::Register()** in an initialization routine, register the C++ class before any resource database containing instances is loaded.
5. Define how your class instances respond to the notifications sent by the resource or widget managers.
6. Define and implement the class API.
7. Rebuild the development environment.
8. Provide an editor for your new class if you want to fully integrate your subclass in the Elements Environment.

Note: The fact that the class is implemented in C++ instead of C does not create any special problems.

To complete these tasks, see the example subclex.cpp and the readme.txt in this directory:

```
ee21\cpp\examples\gui\subclass
```

Defining a C++ Subclass in the Elements Environment

You can define a C++ subclass using the Class Editor in the Resource Browser. The Class Editor allows you to specify the characteristics of the subclass you add.

In the Class Editor, you can also specify that the new class appear in the Window Editor tools palette. If you choose this option, you can create more instances of the subclass by selecting the icon that you specified for your class in the Window Editor palette.

When you save your class, the Elements Environment generates a resource class definition in the .rc text file of the module in which you defined the class. The resource class definition has this syntax:

```
(RClas.Compile
       Name: "MyClass"
       Parent: "MyParent"
       Module: "MyMod"
       Version: 8
       // Define persistent fields here.
)
```

The instances of the subclass are described as in the .rc file:

```
(MyClass.Compile
       Name: "MyMod.MyWgt"
       ...
)
```

Warning: The Elements Environment Class Editor does **not** allow you to modify classes that already exist. You can only edit all the attributes of a new class during the session in which you created the class. If you want to modify the class later, you need to edit the .rc file and recreate the resource database (.dat file) with rescomp.

Registering a C++ Subclass in the Resource Manager

The **RCLAS_CPPFULL** macro defines (among other things) a **Register** static member function that registers the C++ class to the resource manager. You should call this **Register** function before any instance of the subclass is created or loaded.

To register a C++ subclass to the resource manager:

1. Include the **RCLAS_CPPFULL(***class, parent_class***)** macro in the C++ class definition.
2. Implement the two required constructors and the destructor.

These constructors are required:

```
<class>::<class>(RClasPtr rclas, RClasCreateCPtr cptr);
<class>::<class>();
```

For which the default implementation would be

```
<class>::<class>(RClasPtr rclas, RClasCreateCPtr cptr) :
<pclass>(rclas, cptr)
{
       .../...
}

<class>::<class>() : <pclass>(<class>::OiClass, NULL)
{
       .../...
}
```

Note: If any of these constructors or destructor is missing, the class will have undefined symbols at link time.

If you create the class in the Resource Browser, the Elements Environment generates the template code for all this. Thus, you do not need to type the code yourself.

For example, if you define **MyClass** as a subclass of **ParentClass** using the Resource Browser, the following code gets generated:

```
class MyClass : public ParentClass {
      // class information such as fields, member functions
public:
      // resource-manager information
      RCLAS_CPPFULL(MyClass, ParentClass)
}

MyClass::MyClass(RClasPtr rclas, RClasCreateCPtr cptr) :
ParentClass(rclas, cptr)
{
      .../...
}

MyClass::MyClass() : ParentClass(MyClass::OiClass, NULL)
{
      .../...
}
```

Then include this call in one of your initialization routines:

```
MyClass::Register();
```

Note: If you do not call **MyClass::Register()** in your main program before creating an instance of **MyClass**, the program will cause errors that are difficult to diagnose. These errors will violate segmentation when accessing or constructing members of **MyClass**.

Generic Container Classes

The Elements Environment provides a set of *generic container classes* that you can use to define any type of array. There are three types of generic arrays:

- The generic **ArPtr** class, which is defined in arptrpub.h, instantiates the array of pointers classes. *Array of pointers* contains pointers to objects, **not** the objects themselves. Copying an array of pointers copies only the pointers. The original array and the copy share the objects.
- The generic **ArRec** class, which is defined in arrecpub.h, instantiates the array of objects classes. *Array of objects*, or *structures*, contains the objects themselves. All objects must be of the same size. Copying an array of objects copies the objects themselves.

- The generic **ArNum** class, which is defined in arnumpub.h, instantiates the array of numbers classes. *Array of numbers* contains numbers of any integral type.

Arrays provide these services:

- Store and retrieve an element by index
- Look up an element by key
- Store elements without duplication
- Sort the elements according to a sorting method you provide

You define a specific array class by using one of the **ARXXX_DEFCLASS** macros. For example:

```
ARPTR_CLASS(NDWinPtr, NDWinCPtr)
```

defines a new C++ class **ArPtrOfNDWinPtr**, where each element is a `NDWinPtr`. All the methods in the class use the proper types.

You can use this class as follows:

```
ArPtrOfNDWinPtr array;
NDWinPtr        aWin;

array.AppendElt(aWin);
aWin = array.GetNthElt(0);
```

Code Generation

From the Resource Browser in the Elements Environment, you can:

- Generate code
- Separate the code into header and source files

Laying Out Windows

When you use the Window Editor to create a window in the Resource Browser, the widgets you add to your window determine the exact code that gets generated in the C++ source-code template file. The code template has this structure:

- The first section of the code template contains a list of `#include` statements. The list of libraries that are installed depends on the exact product configuration and the types of resources you included in your project.
- The Notification Handlers section contains the handler template, which allows you to customize actions. This section consists of a set of virtual member functions.

You can customize this generated code by inserting your own code in the appropriate notification handlers. The custom code that you add overrides the default behavior of the widgets. For example:

```
void App1Win1::HitTButOk(void)
{
    NDAlrtW::Ok(mTEd->GetStr());
}
```

- The constructor for the window registers the widgets. This allows you to reference the widget using the variable names specified in the class declaration. It also registers the appropriate notifications for the added widgets through the **WIN_SETNFYHANDLER** macro in the window constructor method. For each widget class, the string-list resource `<classmod>.<Myclass>CodeGNfys` controls the default list of generated handlers. For example, for single-line text edits, the list is in `TEd.STEdCodeGNfys`.

  Note: You can access each widget from anywhere as a simple member attribute of the window object. You can add more NfyHandlers if you need to.

- The next section loads the window resource and calls the constructor. The **Init** method initializes the window.
- The last section contains the application **main()** routine and a set of library-installation functions. The list of libraries installed depends on the exact product configuration. These are the resources that define the installation functions:
  - `includes` for the list of includes
  - `CppInstalls` for the list of C++ statements to register the element
  - `CppLoadInits` to load and initialize the element
  - `CppExits` to leave the element

Separating Source Code and Header Files

The generated code can either include a declaration of the class that describes the window or the declaration can be output to a separate header file. The presence or the absence of a header-file name in the Module Editor of the Resource Browser determines whether or not to save the code into separate header and source files:

- If the user module has entries for both—mymod.cpp and mymod.h—two distinct files are generated.
- If the user module has only one source name—mymod.cpp—and no header name, the whole code is placed in mymod.cpp.

The header section contains the declarations of member functions that implement the instance-level callbacks for each window class. It also contains member variables to reference the widgets in a programmer-friendly way with the function **NDWgt::GetNamedWgt()**.

Creating Custom Classes

If you use the Class Editor to define your own custom subclass in the Elements Environment, all the C++ code to define and register this class will be automatically generated. The generated code will use the **RCLAS_CPPFULL** macro.

For example, if you create a subclass **SubBrows** of the class **Brows** with one persistent field `F1` of type string, this code will be generated:

```
class SubBrows : public NDBrows {
protected:
          Str          F1;
public:
          RCLAS_CPPFULL(SubBrows, NDBrows)
          static void CppRegister(void);
};
typedef class SubBrows C_FAR * SubBrowsPtr;

PFldRec SubBrows::OiFields[] = {
{ "F1", 0, PFLD_TYPESTR, PFLD_CATOTHER },
{ (CStr)NULL }
};

SubBrows::SubBrows(RClasPtr rclas, RClasCreateCPtr
cptr) : NDBrows(rclas, cptr)
{
}

SubBrows::SubBrows() : NDBrows(SubBrows::OiClass,
(RClasCreateCPtr)NULL)
{
}

SubBrows::~SubBrows()
{
}

RClasPtr SubBrows::OiClass = NULL;

void SubBrows::CppRegister(void)
{
          SubBrowsPtr n = NULL;
          OiFields[0].Offset = RES_OFFSETOF(n, F1);
          Register();
}
```

The **main()** routine of your application will contain:

```
SubBrows::CppRegister();
```

Code Regeneration

The *code regeneration* process maintains common source files for both regenerated and customized (user-generated) code. Each file usually contains only one module. The Elements Environment generates source files when they are first saved. They are then maintained by you and by the code regenerator.

The code-parsing strategy performs minimum updates in a reliable way. The parser uses code *annotations*, or comments, to greatly improve its reliability. The annotations help you distinguish text areas controlled by the code regenerator from areas where you enter text and edit custom code.

The code regenerator always preserves all the original code changes that you make, if you made changes only where the code regenerator does **not** write to. Therefore, it systematically checks annotated areas before updating them in order to identify modifications. For example, if you insert text in a solid block, the entire section is commented out between `#if 0` and `#endif`. The regenerated block is inserted above the commented-out section. You can later decide whether to keep the changes and merge them with the new version. The same thing happens in flexible areas for key commands that are not recognized or are no longer part of the project you are currently editing (for example, removed menu items).

Warning: Do **not** modify code within areas that contain comments alerting you not to modify the code. Only the code regenerator can write to those areas.

Regeneration is only initiated when no errors are found. If you accidentally alter annotations, the code regenerator provides error comments. The regeneration process always produces an output, even if it is only error messages.

Limitations

Programming in the Elements Environment has certain limitations for creating C++ applications.

Copy and Assignment Operations

You cannot copy any of the C++ objects, and you cannot assign one object to another:

```
NDVStr::myString("Hello");
NDVStr::copyString(myString); // Compilation error
```

```
NDVStr::assignedString;
assignedString = myString; // Compilation error
```

Customizing Editors in C++

You cannot customize editors using standard C++ mechanisms. If your application requires embedding the development environment or building editors for your custom classes, follow the same conventions as the C code listed in the custom.doc file.

Implementation Notes for Current C Users

Use the following guidelines to build the Elements Environment C++ libraries.

Note: Do not mix C and C++ calls in your code, if C++ exception handling is enabled. If you do, it will cause compiler errors.

Member Functions

Use member functions and virtual member functions instead of API calls in C.

For example, use:

```
win->Show();
```

instead of:

```
WIN_Show(win);
```

Note: In some cases, the conversion is not straightforward. The drawing API is in fact defined in the **NDWgt** class. Thus:

```
DRAW_SetColors(wgt, COLOR_Red(), COLOR_Blue());
DRAW_Rect(wgt, rect);
```

is converted into:

```
wgt->DrawSetColors(NDColor::Red(), NDColor::Blue());
wgt->DrawRect(rect);
```

Memory Allocation and Deallocation

Instead of the calls in C, use the Elements Environment overloaded operators `new` and `delete` to allocate or deallocate an instance of a class. These operators are in fact redefined for all classes inheriting from **NDRes**.

For example, use:

```
WinPtr = new ("mod", "win");
win->SetLabel("MyWindow");
```

instead of:

```
WinPtr = WIN_LoadSized("mod", "win",SizeOf(winRec));
WIN_SetLabel(win, "My Window");
      .../...
```

Constructors and Destructors

Use C++ constructors and destructors to create or destroy an instance of a class.

For example, use:

```
NDFilePtr file = new NDFile("data");
```

instead of:

```
FilePtr file = FILE_New();
FILE_ConstructName(file, "data");
```

After the function **AddWgt()** attaches the widget to a container widget (either a panel or a window), you only need to call the destructor for the container. The container widget's destructor will in turn call the destructor for each attached widget.

Overloaded Members

Use C++ overloaded members for the C API calls that are functionally the same but differ only in the type of arguments.

For example, use:

```
Int16 i16 = 4;
Int32 i32 = 6;
ted->Set(i16);
ted->Set(i32);
```

instead of:

```
Int16 i16 = 4;
Int32 i32 = 16;
TED_SetInt16(ted, i16);
TED_SetInt32(ted, i32);
```

Notifications as Virtual Member Functions

Use virtual member functions instead of notifications. For each notification, a virtual member function takes as its argument the notification data it defined. The standard way of customizing class behavior in C is replaced by the standard C++ subclass-customization scheme (reimplementing virtual member functions).

These examples define a subclass of the **PBut** class that customizes the response to the `NfyHit` notification:

The code in C++:

```
class MyPButClass : public NDPBut {
	.../...
public:
	virtual void NfyHit(void);
}

void MyPBut::NfyHit(void)
{
	// Do something.
	NDPBut::NfyHit(); // Call default action if you want.
	// Do something else.
}
```

The code in C:

```
typedef struct _MyPButRec {
	PBUT_REC
	.../...
} MyPButRec, C_FAR *MyPButPtr;

typedef enum {
	PBUT_NFYINHERIT(MYPBUT)
} MyPButNfyEnum;

static void C_FAR S_MyPButDefNfy(MyPButPtr pbut, MyPButNfyEnum
code)
{
	switch (code) {
	case MYPBUT_NFYHIT:
		/* Do something at the class level. */
		break;
	default:
		PBUT_DefNfy(pbut, code);
}
```

Using Custom Constructors and Destructors

This section provides information for developers who want to define custom constructors in C++ resource subclasses:

- Defining default constructors with the **RCLAS_CPLUSFULL** macro
- Defining custom constructors for a C++ subclass

Defining Default Constructors with RCLAS_CPLUSFULL

The **RCLAS_CPLUSFULL** macro is designed to encapsulate all the details of resource subclassing in C++. The **RCLAS_CPLUSFULL** macro defines:

- The special constructor

- The registration logic

This is how the **RCLAS_CPLUSFULL** macro works:

1. To subclass a widget class, you place the **RCLAS_CPLUSFULL** macro in the definition of your subclass:

```
RCLAS_CPLUSFULL(MyClass, ParentClass)
```

2. This macro expands to:

```
private:
static RClasPtr OiClass;
static ResPtr C_FAR OiNew(RClasCreateCPtr cptr){
return (ResPtr)new MyClass(MyClass::OiClass, cptr);}
static void C_FAR OiDelete(ResPtr res)
      { delete (MyClass C_FAR*)res;}
protected:
MyClass(RClasPtr rclas, RClasCreateCPtr cptr);
public:
PFldRec SubMyClass::OiFields[] =
{{ (CStr)NULL }};

void C_FAR* operator new(size_t size)
      { return MyClass::OiClass->AllocObj(size);}
void operator delete(void C_FAR* obj)
{ MyClass:OiClass->DeallocObj(obj); }

MyClass();
virtual ~MyClass();
static RClasPtr Class() { return MyClass::OiClass; }
static PFldPtr GetOiFields() { return OiFields; }
static void Register()
{ MyClass::OiClass = RCLAS_CPlusRegister("MyClass",
              OiNew, OiDelete, (ResNfyProc) NULL,
              ParentClass::Class(), MyClass::GetOiFields()); }
```

3. The `OiClass` private static variable is set to the resource class the resource manager creates when the class is registered.
4. The resource manager calls the **OiNew** and **OiDelete** private functions to create and delete instances.
5. The protected constructor takes two arguments. It is used by the resource manager to load instances or by subclasses to perform the base-class initialization.
6. The public constructor does not take any arguments. You can use it to construct new instances of the class from scratch.
7. The public **Class** function returns the resource class the resource manager created when the class was registered.

8. The public **GetOiFields** function returns the `OiFields` associated with the class.
9. The public **Register()** static function registers the class. You should call it in an initialization routine of the C++ program.

Defining Custom Constructors

To define custom constructors in the resource subclasses, you must understand the role of the **protected constructor**.

The protected constructor takes two arguments:

- The first is the resource class of the resource that is being instantiated. This will be different from the `OiClass` of the current C++ class if you call the protected constructor as a base-class constructor from a subclass constructor.
- The second contains information that is private to the resource manager. It describes the context in which the instance is created, such as by a calling the `new` operator or by loading from the .dat file. When this argument is set to `NULL`, the resource manager considers that the resource is created dynamically through the `new` operator.

C++ Exception Handling

Two types of C++ libraries can be created from the Elements Environment source code:

- Libraries that use the C++ `try`/`catch`/`throw` mechanisms internally.

  Note: This is the default.

  These libraries are intended for C++ programmers who:
  - Use C++ exception handling in their code
  - Use the Elements Environment with other C++ libraries that rely on C++ exception handling
- Libraries that use the **setjmp / longjmp** calls for exception handling. These libraries are intended for C++ programmers who:
  - Do not want to use C++ exception handling
  - Rely on tools that do not support C++ exception handling, such as the GNU C++ compiler on most UNIX platforms

When C++ exception handling is enabled, these actions take place:

- The libraries signal abnormal conditions by throwing an instance of the **NDExcept** C++ class.
- The **NDExcept** class defines a function to get the frame-stack pointer.
- The **NDErrFrame** class defined in the errpub.h file provides the functions to query the error stack for either C or C++ libraries.
- Queries are performed in your error handler before calling the **ERR_RETRY** and **ERR_RECOVER** macros.
- If you do not want an alert dialog, the error handler can use the **ERR_RETRYSILENT** or **ERR_RECOVERSILENT** macros to prevent the alert from being displayed.

The C++ exception handling is implemented as follows:

- The `CPP_EXCEPTION` compilation flag controls whether or not C++ exception handling is enabled. This flag is automatically defined when the C++ compiler is configured for C++ exception handling.
- The **ErrFrame** structure is redefined as a class with a constructor and a destructor, which link and unlink the stack frames. The destructor ensures that the error frames are properly unlinked when a C++ exception is thrown.
- The **ERR_CATCH** macro is defined differently, depending on whether the compiler's C++ exception handling is used or not. In C, its expansion contains a **setjmp()** call; in C++ it expands into the beginning of a `try` block.
- The **err_catch** label becomes a macro and expands into a pair of `catch` blocks and a `goto` so that the code compiles when the label is immediately followed by a colon (`:`).
- The **ERR_Signal** function is implemented with a C++ `throw` rather than a call to **longjmp**.

Existing exception-handling code based on the **ERR_** macros will compile and work as they currently do, because they are automatically remapped in C++ `try`/`catch` constructs.

Note: There are a few cases where existing code will not compile with the `CPP_EXCEPTION` flag on. For example, code that branches to the **err_catch** label with an explicit `goto` will not compile. However, it can easily be modified—for example, by introducing a second label with a different name.

For More Information about the C++ API

- See the *Open Interface Element C++ Programmer's Guide*.
- See the *Open Interface Element C++ API Reference, Vol. 1, Widget Classes*.
- See the *Open Interface Element C++ API Reference, Vol. 2, GUI Helper Classes*.
- See the *Intelligent Rules Element C++ Programmer's Guide*.
- See the *Data Access Element C++ Programmer's Guide*.
- See the *Web Element C++ Programmer's Guide*.
- See the *Elements Application Services C++ Programmer's Guide*.

Chapter

10 Building Applications in the Elements Environment

You can use the Neuron Data Elements Environment to build your application after you:

1. Install the Development libraries and Deployment kits
2. Modify your environment variables with the Elements Environment libraries you want

This release of the Elements Environment provides these libraries for your applications:

- The Open Interface Element
- The Data Access Element, including:
  - Sybase driver
  - Oracle driver
  - ODBC driver (for PCs only)
  - ProtoDB driver (provided by Neuron Data)
- The Intelligent Rules Element
- The Distributed Messaging Element
- The Web Element

Configuring the Elements Environment

You can run applications in the Elements Environment by:

- Dynamically loading the Elements Environment libraries
- Statically linking the Elements Environment libraries

Note: In the development environment, you can **only** dynamically load the libraries. The Distributed Messaging Element is a runtime product that is linked statically. For information about building Distributed Messaging Element applications, see the *Distributed Messaging Element Programmer’s Guide*.

See "Default Configuration" on page 118 for the configuration files the Elements Environment supports.

The file **nd.h** contains all the library-initialization statements you need to link optional Neuron Data Elements libraries with your application. You can find this file in this directory:

`$ND_HOME\c\include` **or** `$ND_HOME\cpp\include`

This file also contains *preprocessor compilation flags*. These let you enable the required libraries for your particular type of application. For a list of flags, see "Building Applications" on page 125.

Default Configuration

Installing the Elements Environment involves only one executable program—**ee**. This file dynamically loads Neuron Data Elements based on a configuration file that you specify. The default configuration file, which enables all the Elements you install, is **ee.cfg**. This file is in this directory:

```
$ND_HOME\dat
```

Warning: Make sure you install only the Elements that you are licensed to use. If you install unlicensed Elements, the Elements Environment configuration file will have incorrect information.

To verify that you installed only the Elements that you are licensed to use, enter this command:

```
c:\authfeat l
```

This gives you a list of Elements that you have installed. Delete the Elements that you are not licensed to use from the Elements Environment directory on your hard drive.

The following table identifies which configuration flags have been enabled in specific configuration files in the above directory.

Note: The `ND_IM_XXX` flag in this table provides the language-input method for international support.

Name of Configuration File	Elements Enabled	Environment	Configuration Flags Enabled
`oie.cfg`	Open Interface Element	Development	`ND_OI`
			`ND_GUI`
			`ND_IM_XXX`
			`ND_EDITORS`
			`ND_SCRIPTING`
`gui.cfg`	Open Interface Element	Runtime	`ND_OI`
			`ND_GUI`
			`ND_IM_XXX`

Name of Configuration File	Elements Enabled	Environment	Configuration Flags Enabled
`we.cfg`	Open Interface Element and Web Element	Development	ND_OI ND_GUI ND_IM_XXX ND_WE ND_RUNSCRIPT ND_EDITORS ND_SCRIPTING
`web.cfg`	Open Interface Element and Web Element	Runtime	ND_OI ND_GUI ND_IM_XXX ND_WE
`dae.cfg`	Open Interface Element and Data Access Element	Development	ND_OI ND_GUI ND_IM_XXX ND_DA ND_EDITORS ND_SCRIPTING
`da.cfg`	Open Interface Element and Data Access Element	Runtime	ND_OI ND_GUI ND_IM_XXX ND_DA ND_DA_XXX (driver flags)
`ire.cfg`	Intelligent Rules Element and Open Interface Element	Development	ND_OI ND_GUI ND_IM_XXX ND_IR ND_IR_DB ND_IR_EXE ND_EDITORS ND_SCRIPTING
`rules.cfg`	Intelligent Rules Element and Open Interface Element	Runtime	ND_IR ND_IR_DB ND_IR_EXE

The Data Access Element drivers are disabled by default. To enable a database driver, uncomment the corresponding database-driver flag among these flags:

```
ND_DA_ORA7
ND_DA_SYB
ND_DA_PDB
ND_DA_ODBC (for PCs only)
ND_DA_DB2
```

Starting the Elements Environment

- To start the Elements Environment with the default configuration from your PC or Macintosh, double-click the EE icon.
- To start the Elements Environment with the default configuration from a command line, enter:

  ```
  ee
  ```
- To run the Elements Environment with a specific configuration file, enter:

  ```
  ee -ND_DYNCONFIG=configurationfile
  ```
  where *configuration file* is the name of the file, without its extension, that specifies the libraries and the options that you want to load.

Including Additional Configuration Files

To include another configuration file, put the keyword `Load` in the command line, using this syntax:

```
load configurationfilenamewithoutextension
```

Compiler-Flag Options

The configuration of the Elements Environment software is controlled by a set of compiler flags. You can set each flag to 0 or 1 using either of these methods:

- Explicitly in a `#define` statement in the main source-code file **before** including nd.h
- From the command line of the compiler. For example, this will set the flag to 1:

  ```
  -ND_DA_ORA7
  ```

If you do not set a flag explicitly, it will take a default value as defined by your environment.

The keyword `Option` lets you specify the software components that you want to enable for the Elements Environment. Components include the individual Elements, as well as libraries that provide additional or specific functionality. See the following sections for the options available.

Enabling and Disabling Options

- To turn an option on, use this syntax:

  `*OPTIONNAME* on`

- To turn an option off, use this syntax:

  `*OPTIONNAME* off`

Note: Use only **uppercase** for the option name.

- To ignore options, comment them out by placing them between these characters:

  `/* */`

Note: If multiple definitions of the same option appear in a configuration file, only the last definition is used.

Common Options

You can use these options with all the Neuron Data Elements, **except** the Distributed Messaging Element:

Option	Description
`ND_DYNCONFIG`	Permits dynamic loading instead of static linking.
`ND_EDITORS`	Provides the libraries for the graphical editors for all enabled Elements.
`ND_SCRIPTING`	Provides the libraries for the scripting servers for all enabled Elements.
`ND_RUNSCRPT`	Starts an application with an **AppStartup** script. This requires `ND_SCRIPTING` to be turned on.You can launch scripts without turning on `ND_OI` or `ND_GUI`—for example, in an automatic tester or in an application using only script procedures.
`ND_GUI`	Provides the libraries for the graphical user interface (GUI), instead of character mode. **Note:** Character mode is not available for all Elements.
`ND_IM_NATIVE`	Allows the Asian language input method as defined by the native operating system.
`ND_IM_JAPANESE`	Japanese input method.
`ND_IM_KOREAN`	Korean input method.
`ND_SCRIPT_SERVERS`	Controls whether the script servers are initialized.

Options Controlling the Main Neuron Data Elements

Option	Description
ND_OI	The Open Interface Element for portable graphics and toolkit. This enables ND_GUI.
ND_IR	The Intelligent Rules Element (formerly NEXPERT Object) for rules and objects.
ND_DA	The Data Access Element for transparent access to databases and integration of the GUI with data.
ND_DM	The Distributed Messaging Element for distributed objects and other advanced communication capabilities. **Note:** The Distributed Messaging Element is statically linked.
ND_WE	The Web Element for browsing the World Wide Web. This requires ND_OI.

Element-specific Options

Option	Description
The Intelligent Rules Element	
ND_IR_DA	A bridge between the Intelligent Rules Element and the Data Access Element for direct access from rules to relational databases. **Note:** The system enables this flag when you specify ND_DA with your compiler.
ND_IR_EXE	Library that provides the Intelligent Rules Element's built-in execute functions. This requires ND_IR.
ND_IR_DB	Library that provides access to spreadsheets and nonrelational databases. This requires ND_IR. For access to relational databases, it requires ND_DA and ND_DA_XXX.
The Data Access Element (DAE)	
Note: All these options require ND_DA.	
ND_DA_ORA7	DAE Oracle7 driver.
ND_DA_SYB	DAE Sybase driver.
ND_DA_ODBC	DAE ODBC driver.
ND_DA_PDB	DAE ProtoDB driver.
ND_DA_DB2	DAE Informix driver.

Note: The Elements Application Services libraries (Core and Res) are enabled by default.

Running the Elements Environment Examples

To check if you can create Elements Environment C and C++ applications, try to recreate one of the sample applications installed in this directory:

`ND_HOME\c\examples\`*`xxx`* **or** `ND_HOME\cpp\examples\`*`xxx`*

These sections show you how to run the examples. Before you run an example, compile it using the make file provided.

Examples for the Open Interface Element

1. Change to this directory:

   ```
   examples\gui\lbox
   ```
2. If you are using a Microsoft compiler, enter:

   ```
   nmake -f makefile.pc
   ```
   or

   If you are using a UNIX compiler, enter:

   ```
   make
   ```
   The Elements Environment resource compiler (rescomp) runs on the file lboxex.rc to generate lboxex.dat and creates the lboxex object file. The linker creates the application executable, lboxex.

   Note: The order of execution depends on the make program used, and it is not important.
3. Enter this command to build and run the lboxex application:

   ```
   lboxex
   ```
   A window with several pushbuttons appears.

   Note: If the window does not appear, check the nd.dbg file generated by the ee executable.

Examples for the Data Access Element

1. Change to this directory:

   ```
   examples\da\dataview
   ```
2. If you are using a Microsoft compiler, enter:

   ```
   nmake -f makefile.pc
   ```
   or

   If you are using a UNIX compiler, enter:

   ```
   make
   ```
   The Elements Environment resource compiler (rescomp) runs on the file dataview.rc to generate dataview.dat and creates the dataview object file. Then the linker creates the application executable, dataview.

Note: The order of execution depends on the make program used, and it is not important.

3. Enter this command to build and run the dataview application:

```
dataview
```

   A window with datasource connection options appears.

   Note: If the window does not appear, check the ND.DBG file generated by the ee executable.

Examples for the Intelligent Rules Element

1. Change to this directory:

```
examples\rules\hello
```

2. If you are using a Microsoft compiler, enter:

```
nmake -f makefile.pc
```

   or

   If you are using a UNIX compiler, enter:

```
make
```

   The compiler creates the hello*xx* object files. Then, the linker creates the hello*xx* application executables.

3. Enter this command to run the hello1 application:

```
hello1
```

   A window with a NXP> prompt should appear.

Note: For a tutorial based on the hello*xx* examples, see the *Intelligent Rules Element Programmer's Guide.*

Examples for the Web Element

1. Change to this directory:

```
examples\web\webwgt
```

2. If you are using a Microsoft compiler, enter:

```
nmake -f makefile.pc
```

   or

   If you are using a UNIX compiler, enter:

```
make
```

   The Elements Environment resource compiler (rescomp) runs on the file webwex.rc to generate webwex.dat and creates the webwex object file. Then, the linker creates the application, webwex.

   Note: The order of execution / creation depends on the make program used, and it is not important.

3. Enter this command to build and run the webwex application:

   ```
   webwex
   ```

   A window that tries to connect to the World Wide Web over your network connection appears. Establishing the connection might take some time. The window appears only when the connection is established; otherwise, it times out.

Examples for the Distributed Messaging Element

See the *Distributed Messaging Element Programmer's Guide.*

Building Applications

After generating applications from your Elements Environment executable, you can:

- Statically link them with the appropriate libraries (the default)
- Load the libraries dynamically. To choose dynamic loading, you must use the `ND_DYNCONFIG` flag. Alternately, you can uncomment this flag in your configuration file.

You must also compile and link your application with the appropriate libraries. The file nd.h contains all the library-initialization statements required to link the Elements Environment libraries with your application. You can find it in this directory:

`ND_HOME\c\include` **or** `ND_HOME\cpp\include`

This file is included in the main source-code file generated by the Elements Environment executable. To support the initialization statements provided by nd.h, there are compiler flags that let you link your application with the appropriate libraries.

See "Compiler-Flag Options" on page 120 for a list and description of the options currently available.

Using Makefiles

The Elements Environment includes two files in the ND_HOME/mkinc directory for the supported compilers:

- For a specific application, makedef.inc defines the compilation and linking flags, and the required libraries.
- makerule.inc contains the rules to compile and link.

The main make file, which the Elements Environment generates, has the target files needed to build the Elements in your application. The main make file must include makedef.inc and makerule.inc.

Chapter

11 Porting and Deploying Applications in the Elements Environment

In the Elements Environment, you can develop applications on one platform and port them to other platforms. You can then deploy the applications to end-users.

There are two versions of the Elements Environment kits:

- The Development kit
- The Deployment kit

The only difference between the two kits is that the Development kit has the Resource Browser and related files; the Deployment kit does **not** have these files. All other libraries and header files are the same. The kits also have the same layout on all platforms.

Note: You must have **both** the Development kit and the Deployment kit to build and deploy your applications.

Follow these guidelines when you port or deploy applications that you develop using the Elements Environment:

- When you rebuild the application for the first time, always use the DEBUG libraries—libdbg (**except** for the Intelligent Rules Element). This lets you easily detect errors.
- Start porting the application that you are developing as soon as you can. This helps you discover:
  - Problems specific to a platform
  - Common problems that are easier to track on one system than on another

  For example, if you develop on both Windows and UNIX, it is often easier to debug your code on UNIX than on Windows.
- When porting an application that is ready to be delivered to users, use a nonrestricted version of the Open Interface Element libraries.
- Initialize the *nd.dat* file. This is a special resource file containing user and security information for all Neuron Data products you have installed. After you initialize nd.dat, you can use it for deploying your application to end-users.

See your *Installation Guide* for information about initializing this file.

Note: In all the procedures in this chapter, *source platform* is the platform on which you developed your application. *Target platform* is the platform to which you are porting your application.

Porting a C or C++ Application across Platforms

1. Install the Elements Environment Development or Deployment kit on the target platform.

   Follow the instructions in the *Installation Guide* for the appropriate platform.

2. Copy all the application files from the source platform to the target platform, including:

   – C and C++ source files
   – header files
   – .rc files
   – makefiles
   – knowledge bases
   – bitmap files
   – flat files

   Note: When you copy files from one system to another, make sure that you copy compiled knowledge-base files and bitmap files in **binary mode**. Copy all text files in **text mode**. Verify that the end-of-line characters convert properly.

3. Edit the makefile for the target platform.

   See ee21.txt in the ee21\doc directory of the Elements Environment CD-ROM for instructions.

   Note: If you do not have the Elements Environment Development kit on the target platform, you can create a makefile from the Elements Environment Development kit on the source platform. You can also modify a makefile from an example in the Open Interface Element.

4. Rebuild your application.

5. Use rescomp to compile the .rc resource files and generate a new .dat library file.

6. Modify the environment variables for your system.

   See the appropriate .pdf files for your platform in the Sys_conf directory of the documentation CD-ROM.

Porting a Script Application across Platforms

1. Install the Elements Environment Development or Deployment kit on the target platform.

   Follow the instructions in the *Installation Guide* for the appropriate platform.

2. Copy these Elements Environment files from the source platform to the target platform:

   – .rc files
   – knowledge-base files
   – .nxp flat-file databases

   Note: If your application uses a mixture of C or C++ and OOScript, also copy the source file, header file, and makefile from the source platform to the target platform.

3. Rebuild your application.

4. Use rescomp to:

   – Compile the .rc resource files
   – Generate a new .dat library file

5. Modify the environment variables for your system.

   See the appropriate .pdf files for your platform in the Sys_conf directory of the documentation CD-ROM.

Deploying Applications

Applications built with the Elements Environment are **not** protected by the hardware key (Macintosh and PC) or security server (UNIX and OpenVMS). Only the development-environment tools are so protected. However, you **cannot** deliver an application with a Deployment kit containing this restricted banner: “This is for development only.” Therefore, you **must** purchase a porting kit, which is exactly the same thing without this restriction.

Note: Deploy your application with the same nd.dat file as the one you used when you built it. Do **not** reinstall the .dat files from the master disks or tape, because your nd.dat file contains a valid serial number.

Using the installation process of your application, you **must** provide a way to protect files, such as .dat files and libraries, from being modified.

Deploying a C or C++ Application

This section assumes that you have already developed the application in the Open Interface Element and ported it to the target user system.

1. Rebuild your application without the debugging information.
2. Make sure you have the following files to run your Elements Environment application on similar platforms:
   - The executable program of the application
   - The Elements Environment libraries for platforms using DLLs or shareable libraries

   Note: Modify your makefile to point to the /dll directory instead of the /dlldbg directory, and use the nondebug and nonrestricted libraries.

   - Any database client libraries if you based your application on the Data Access Element
   - Your application's .dat files
   - The Elements Environment .dat files (taken from the /dat/$ND_LANG and /dat directories)

   Note: If you are **not** using scripts, you do **not** need ndresed.dat and ndlm*.dat.

   - Any additional files specific to your application, such as bitmaps, knowledge bases, and flat files

Note: Maintain the same directory structure when you copy the Elements Environment files to the target platform.

3. Instruct your users to set the ND_PATH environment variable to the directory that contains these files.

   Note: If users get error messages about ND_PATH, or if they cannot install the application, copy all the Elements Environment .dll and .dat libraries to the same directory as your project and try again.

4. If you are using other environment variables, provide a script or an installation program for the users to correctly set the environment.

5. After referring to the *Installation Guide* for the appropriate platform, specify other special platform settings for the target systems.

   For example, you have to set FILES=50 or more in CONFIG.SYS on DOS, and the correct memory size of the application on the Macintosh.

6. Modify the environment variables for the target system.

   See the appropriate .pdf files for your platform in the Sys_conf directory of the documentation CD-ROM.

Deploying a Script Application

This section assumes that you have already developed the application in the Open Interface Element and ported it to the target user system.

1. Make sure you have the following files to run your Elements Environment application on similar platforms:

   – The Elements Environment executable program with option `-ND_DYNCONFIG=runscrpt`

   – The Elements Environment libraries for platforms using DLLs or shareable libraries

   Note: Use nondebug and nonrestricted libraries.

   – Any database client libraries if you based your application on the Data Access Element

   – Your application’s .dat files

   – The Elements Environment .dat files (taken from the /dat/$ND_LANG directory)

   – Any additional files specific to your application, such as bitmaps, knowledge bases, and flat files

Note: Maintain the same directory structure when you copy the Elements Environment files to the target platform.

2. Instruct your users to set the ND_PATH environment variable to the directory that contains these files.

3. If you are using other environment variables, provide a script or an installation program for the users to correctly set the environment.

4. After referring to the *Installation Guide* for the appropriate platform, specify other special platform settings for the target systems.

   For example, you have to set FILES=50 or more in CONFIG.SYS on DOS, and the correct memory size of the application on the Macintosh.

5. Modify the environment variables for the target system.

   See the appropriate .pdf files for your platform in the Sys_conf directory of the documentation CD-ROM.

Chapter

12 Localizing Applications in the Elements Environment

Localizing an application means converting an application from its original language to a target language by translating strings from one language to another. It includes changing alphabets to characters that are completely different from the characters in the original language. The Elements Environment provides the following features to help you localize your applications:

Support for Multibyte Characters

The Elements Environment supports *multibyte characters*, which can handle Asian and European characters as well as ASCII. The Elements Environment also provides a set of string functions designed for multibyte characters.

These modules contain multibyte APIs:

- Str
- VStr
- Char
- Ct

Language-independent APIs

The Elements Environment provides *language-independent application programming interfaces* (APIs), which are designed specifically for multibyte characters. Also, the Elements Environment APIs can accommodate applications that accept characters from two or more alphabets at the same time. These interfaces take advantage of industry-standard character encoding. You can use the same set of APIs for all locales. Therefore, you do not need different versions of the Elements Environment, and switching from one language to another is simple.

Input Methods

The Elements Environment offers an expanded set of input methods for multibyte characters:

- The standard *Canna* input method for inputting Japanese Nihongo characters. Canna is a public-domain Kana-Kanji conversion library.

- The *X input method* (XIM) for UNIX / X Windows environments. XIM offers a broad range of support for multibyte characters, enabling you to input Japanese, Korean, Chinese, Taiwanese, and other character types. The X11 versions provide libraries that communicate with the XIM-compliant input servers.
- On-the-spot multibyte input for Microsoft Windows.

Processing Input and Output Strings

The Elements Environment APIs provide the functions you need to process and output strings. For example, string APIs enable you to:

- Create and destroy strings
- Search for characters in strings
- Extract numeric values
- Compare strings

The Elements Environment also has interfaces for:

- Characters
- Code types
- Code sets

Fonts

You can specify multiple native fonts for the strings displayed on the screen. For example, in JEUS or SJIS environments on UNIX, you can display a string containing a combination of Kanji, Kana, and ASCII characters.

String-Resource Editor

The *string-resource editor* allows you to build tables of string resources. You can substitute these resources with resources from other languages—including multibyte languages—and then recompile. This is useful for internationalizing applications developed for different locales.

Translating Resources with the Resource Compiler

If you want to adapt your user interface for a different native language but do not want to change the underlying structure of your software, you can set the appropriate environment variables and then translate the static strings in your resource files into the target language.

Rather than separately maintaining resource files for each language version of your application, you can create your application in one language, then translate the resources into other languages as required.

To translate your string and label resources, perform the following tasks.

Note: These steps are for UNIX systems, but you can adapt them for other environments through utilities such as MKS tools on the PC or streamedit with MPW on the Macintosh.

Task 1: Generating the Text Resource File for the .dat Libraries

Note: If you are starting with a .rc file, skip to the next task.

1. Run rescomp to generate the text form of the resources (the .rc file) for the .dat libraries. For example:

   To generate the all.rc file for lib1.dat ... lib2.dat, enter:

   ```
   rescomp -output lib1.dat ... lib2.dat > all.rc
   ```

   To localize the Neuron Data .dat files so that you can change the built-in strings, generic windows, and Open Edit itself, enter:

   ```
   cd $ND_HOME/lib
   rescomp -output nd*.dat > nd_all.rc
   ```

2. Make sure that the all.rc file contains the text form of all the resources you want to localize, with data from all the libraries combined into a single file.

Task 2: Generating a Skeleton Dictionary

Use the text file you have just created to generate a skeleton dictionary containing your strings and labels in their original language.

To do this, use the rescomp utility with this syntax:

```
rescomp -gendict mydict all.rc
```

The skeleton dictionary lists all the strings and labels in your application in the format required by the dictionary.

Task 3: Editing the Dictionary

1. For each entry in the skeleton dictionary, place the original word in quotation marks, followed by a right arrow, followed by the translation in quotation marks:

   ```
   “Original Word1"=>"Word1 Translation"
   "Original Word2"=>"Word2 Translation"
   ...
   "Original Wordn"=>"Wordn Translation"
   ```

   For example, an English / French dictionary might contain these entries:

   ```
   "Window"=>"Fenêtre"
   "New Check Button"=>"Nouveau Bouton Marqué"
   "Button"=>"Bouton"
   ```

2. As you translate the dictionary entries, remove lines that do not need to be translated.

Task 4: Creating the Localized .dat Files

1. Rename your existing .dat files or move them to another directory.
2. In the .rc file that you generated, verify that the `Lib.compile` definitions do not contain a `DirName` field that points to the location of your existing .dat files.

   Tip: You can empty the `DirName` fields or set them to a ".", which points to the current directory.
3. Enter a command with this syntax to create a .dat file:

   ```
   rescomp -t mydict filename.rc
   ```
   This creates a .dat file in the current directory.
4. Use this command with the verbose (`-v`) option to display a message whenever a string is translated:

   ```
   % rescomp -tv dict filename.rc
   ```
   For example, using the French / English dictionary created above, this command:

   ```
   % rescomp -tv dict filename.rc
   ```
   produces output that might look like this:

   ```
   Translating "Window" to "Fenêtre"
   Translating "New Check Button" to "Nouveau Bouton
   Marqué"
   Translating "Button" to "Bouton"
   ```

Task 5: Checking Your Application

1. After all the resources are translated, check the appearance of your strings and widgets.

   Depending on the translation, widgets could be too small or too large for the new text they contain.
2. If necessary, adjust the size of buttons, text edits, and choice boxes.

   Note: Menu objects adjust automatically when the string-length changes, so you don't have to change them manually.

Task 6: Changing the Date and Time Formatting

If necessary, change the output formatting of the date and time fields in the application to match the format of these fields for the specific locale.

Enabling the Input Methods for Multibyte Characters

To set up your environment for multibyte input, you do not need special APIs or special code for handling input methods with the Elements Environment. All you have to do is:

1. Set the appropriate environment variables
2. Start the input method server

Note: You must first install either Canna or XIM on your system before you can input multibyte characters.

Enabling the Canna Input Method

To use Canna on a UNIX system running the Elements Environment:

1. Go to the directory containing the Canna software by entering:

```
cd $ND_HOME/canna
```

2. To become the super user, enter:

```
su
```

3. Run the script to install Canna by entering:

```
/canna_install
```

4. To start the Canna server on any machine on your network, enter:

```
$ND_HOME/canna/bin/cannaserver
```

5. To set the `OIT_CANNAHOST` environment variable to the host name running the Canna server, enter:

```
setenv OIT_CANNAHOST myhost
```

   where *myhost* is the name of the Canna-server host.

6. Start your application.

Enabling the XIM

To use XIM on a UNIX system running X Windows and the Elements Environment:

1. Start your XIM-compliant input server.

   For example, in the Solaris 2.4J environment, enter:

```
htt
```

2. Set the LANG environment variable.

   For example, to set it to Japanese, enter in a C shell in the Solaris 2.3J environment:

   ```
   setenv LANG ja
   ```

   Note: On other systems, the name for the Japanese environment may be different.

3. Start your application.

These variables control the XIM environment:

Variable	Value	Description
`ND_XIM`	`TRUE`	Use XIM.
	`FALSE` (the default)	Do **not** use XIM.
`ND_XIMPREEDIT`	`NOTHING`	Preedit on the root window.
	`AREA`	Preedit on the bottom of the Elements Environment window.
	`POSITION` (the default)	Preedit over the cursor position in the text edit.
`ND_XIMSTATUS`	`NOTHING`	Status is displayed on the root window.
	`AREA` (the default)	Status is displayed on the bottom of the Elements Environment window.

Note: The `XIMPreeditCallbacks` and `XIMStatusCallbacks` styles are **not** supported in the current version of the Elements Environment.

Fonts and Font-Family Resources

A *font* is an object that defines how text appears on the screen. From the windowing system's perspective, a font is an object that allows the drawing engine to measure and render pieces of text.

The Elements Environment lets you specify a *font family* through the FFam resource, which is described in the ffampub.h file. This resource, instead of individual font resources, describes the mapping between the font family and native fonts. This new scheme simplifies the dynamic creation and modification of fonts.

Note: The *Xxx*`Family` persistent fields are there for compatibility with earlier releases of the Open Interface Element. Convert your existing font resources to specify the font family through the FFam mechanism.

API sections that are bracketed with the `FONT_COMPAT` compilation flag in the fontpub.h file are there for compatibility with earlier releases. Avoid using these, and convert any code that uses them to the new APIs.

Font-Family Resources

A *font-family resource* describes the mapping between a portable font family and the native fonts that implement the family and its variants on the various windowing systems. Font families enable you to:

- Easily specify portable font resources
- Build portable user interfaces for selecting font styles and sizes

The font-family resources:

- Encapsulate the low-level X11 font names (XLFD)
- Provide high-level management of X11 fonts

Instead of having to specify font names in the XLFD format, these resources allow you to use a simple user interface to change:

- Font family
- Font size
- Font style

Font families also provide critical support for multibyte text. You can set up a font family to describe a set of fonts covering several code sets (for example, JIS-0201 and JIS-0208). Then, the font manager associates several system fonts with a given font resource. The drawing routines switch system fonts transparently when drawing multibyte text.

Specifying a font family is usually simple. However, it can become complex if:

- You have to specify the family on different windowing systems.
- You need to specify a set of fonts covering several code sets for multibyte text.

A font family is expressed in a simple language that provides predefined operators to test various system parameters. This example shows how to use the specification language:

```
x_family "courier"; x_foundry = "adobe";
mac_family "Chicago";
```

```
pm_family "System Proportional";
if is_x11 then (scaling_quality 80) else (scaling_quality 110)
```

The specification is a list of statements separated by semicolons (`;`).

Note: The **last** statement should **not** be followed by a semicolon.

For more information about the Elements Environment font manager and font-family resource, see the *Open Interface Element Users' Guide*. For additional information, see the ffampub.h file.

Key Concepts for Multibyte Characters and Strings

After translating an application's resource strings, add these resource strings to your application code or rewrite the code to support one or more locales. To do this, you need to use the Elements Environment APIs directly.

The Elements Environment APIs support multibyte characters needed for non-English languages. Multibyte characters require:

- Code sets
- Code mappings
- Code types

These represent the characters in an alphabet as numeric codes and determine how these codes are placed within a multibyte-character structure.

Code Sets

A *code set* is a numeric representation of each character in an alphabet. The numeric codes in each code set vary in their hexadecimal range. Most code sets are extensions to the ASCII character set. Code sets are combined with mappings to form a code type. Elements Environment supports these code sets:

Elements Environment Version	Supported Code Sets
Version 2.1	ASCII
	ISO_LATIN1
	JIS_0201
	JIS_0208
Korean	ASCII
	KSC_5601

Elements Environment Version	Supported Code Sets
Simplified Chinese	ASCII
	GB2312
Traditional Chinese	ASCII
	BIG5
	For Solaris and SunOs:
	ASCII
	CNS11643-1
	CNS11643-2
	CNS11643-3

Code Mapping

Code mapping determines the representation of the encoded character within a multibyte character. A *multibyte character* is an unsigned 32-bit integer. Code mapping includes placing the bytes within the character and manipulating the bytes if necessary.

Code mapping can be more complex than byte placement: The JIS code set defines codes in which the first and second bytes are in the 0x21–0x7e range. JIS bytes cannot be inserted into a string regardless of the byte order. This is because the JIS code would then be indistinguishable from the ASCII codes. These mappings address this problem:

- The SJIS mapping is complex. The SJIS transposes the JIS_0208 code in these ranges:
  - the first byte in 0x81–0x9f or 0xe0–0xfc
  - the second byte in 0x40–0x7e or 0x80–0xfc

  It includes a transposition of JIS-0201 code in these ranges:
  - the first byte in the 0xa1–0xdf range
  - the second byte in the 0x21–0x7e range
- The JEUC mapping transposes the first byte of a JIS_0201 code to 0x8e and the second byte in the 0xa1–0xfe range.

  It transposes the first byte of a JIS_0208 code in the 0xa1–0xfe range and the second byte in the 0xa1–0xfe range.
- The KSC 8-bit mapping code set transposes a code in the 0xa1–0xfe, 0xa1–0xfe range.
- The GB 8-bit mapping code set transposes a code in the 0xa1–0xfe, 0xa1–0xfe range.

- The BIG5 code set's first byte is in the 0xa1–0xfe range, and the second byte is in the 0x40–0x7e or 0xa1–0xfe range. This allows it to be used with ASCII as it is.
- The CNS code set defines 16 planes of 2-byte characters in the 0x21–0x7e and 0x21–0x7e ranges. In CNS code mapping, the first plane is transposed to the 0xa1–0xfe range, and the rest of the planes are mapped to 4-byte characters as follows:

  `0x8e, 0xa0+`*`n`*`, 0xa1-0xfe, 0xal-0xfe`

  where *n* is the number of CNS character planes.

Note: CNS defines 16 planes of character mapping, but actual character codes are defined in only 7 planes. The Elements Environment supports only the first 3 planes.

Code Types

A *code type*, or coding scheme, combines one or more code sets with a code mapping.

For single-byte-ASCII or extended-ASCII characters, the byte value maps directly to the code value. For these alphabets, the code set and the code type are identical.

For multibyte characters, different code types can be based on the same code set but on different code mappings. For example, the Japanese EUC code type offered by Sun and the SJIS code type offered by Sony are two different mappings of the JIS code set.

The Elements Environment provides two levels of support for code sets:

- Tested
- Untested

Fully Supported and Tested Code Types

Code types supported and tested under the current version of the Elements Environment include:

ASCII Code Type

The CT_ASCII code type contains the CS_ASCII code set.

CJK Code Types

In the CJK code-type group, the Elements Environment supports:

- CT_SJIS, which combines CS_ASCII with CS_JIS_0201 and CS_JIS_0208
- CT_JEUC, which combines CS_ASCII with CS_JIS_0201, CS_JIS_0208, and CS_JIS_0212
- CT_KSC, which consists of CS_ASCII plus CS_KSC_5601
- CT_GB, which is a combination of CS_ASCII and CS_GB_2312
- CT_BIG5, which is a combination of CS_ASCII and CS_BIG5
- CT_CNS, which consists of CS_ASCII, CS_CNS11643-1, CS_CNS11643-2, and CS_CNS11643-3

Untested Code Types

Code types that are supported by the current version of the Elements Environment but are not tested include:

ISO 8859_X Code Types

The CT_ISO... code types combine the CS_ASCII, CS_EMPTY_809f, and CS_ISO... code sets. The Elements Environment supports these ISO 8859_X code types:

- CT_ISO_LATIN1
- CT_ISO_LATIN2
- CT_ISO_LATIN3
- CT_ISO_LATIN4
- CT_ISO_CYRILLIC
- CT_ISO_ARABIC
- CT_ISO_GREEK
- CT_ISO_HEBREW
- CT_ISO_LATIN9

Adobe Code Types

The Elements Environment supports CT_ADOBE_STD, which combines the CS_ASCII, CS_EMPTY_809f, and CS_ADOBE_STD code sets:

- The CT_LATIN1 code type contains CS_ASCII, CS_ADOBE_LATIN1, and CS_ISO_LATIN1.
- The CT_AD0BE_SYMBOL code type contains CS_ASCII (00–f only), CS_EMPTY_809f, and CS_ADOBE_SYMBOL.
- The CT_ADOBE_ZAPFDB code type contains the CS_ADOBE_ZAPFDB code set only.

Macintosh Code Types

- The CT_MAC_ROMAN code type combines the CS_ASCII and CS_MAC_ROMAN code sets.
- The CT_MAC_ARABIC code type combines CS_ASCII with CS_ISO_ARABIC and CS_MAC_ARABIC.
- The CT_MAC_HEBREW code type combines the CS_ASCII, CS_ISO_HEBREW, and CS_MAC_HEBREW code sets.

Microsoft Windows Code Types

The Microsoft Windows code types have two forms:

- The 1252 code type contains the CS_ASCII, CS_MSW_ANSII, and CS_ISO_LATIN1 code sets.
- The 125X code type combines CS_ASCII and CS_MSW_125X.

The Elements Environment supports these Microsoft Windows code types:

- CT_MSW_EASTEURO
- CT_MSW_CYRILLIC
- CT_MSW_ANSI
- CT_MSW_GREEK
- CT_MSW_TURK
- CT_MSW_HEBREW
- CT_MSW_ARABIC

PC Code Types

The PC code types combine CS_ASCII with the specific PC code sets. The Elements Environment offers these code types:

- CT_PC_437
- CT_PC_850
- CT_PC_852
- CT_PC_855
- CT_PC_857
- CT_PC_860
- CT_PC_861
- CT_PC_863
- CT_PC_864
- CT_PC_865
- CT_PC_869
- CT_PC_M4

Unicode Code Type

The CT_UNICODE code type contains these code sets:

- CS_ASCII
- CS_EMPTY_809
- CS_ISO_LATIN1
- CS_UNICODE

HP ROMAN8

The CT_HP_ROMAN8 code type contains the CS_HP_ROMAN8 code set.

Key Character and String Types

Two basic datatypes control how characters and strings are manipulated in the application code:

- *Native datatype* supports applications operating in one language at a time.

  Note: Systems dedicated to a specific locale already have a native code type specified.
- *UniCode* type supports Unicode characters contained in Unicode strings.

Character Type Definitions

The Elements Environment Char module defines `Char` and `UniCode` character types.

- The `Char` type is a 1-byte section of a global string. The `ChCode` type encodes a multibyte character in a 32-bit unsigned integer.
- The `UniCode` character type encodes a character in a 16-bit unsigned integer. `UniCodePtr` is a pointer to a `UniCode` character.

4-Byte Character Format

The `ChCode` type encodes characters in an unsigned 32 bit integer. `ChCode` characters contain four bytes: Byte1, Byte2, Byte3, and Byte4. Byte1 is the least significant byte, and Byte4 is the most significant.

Multibyte-character encoding is shown in this table:

Byte Number	Contents
`Byte1`	First byte of the multibyte character
`Byte2`	Second byte of the multibyte character, or NULL
`Byte3`	Third byte of the multibyte character, or NULL
`Byte4`	NULL

Note: `Char` and `ChCode` values are always identical for pure ASCII characters, but are not necessarily the same for multibyte characters.

Basic String types

The Elements Environment Str module defines native and Unicode string types.

- A `Str` string is an array of `Char` and / or `ChCode` characters. There are also types to accommodate these cases:
  - where the global string is constant
  - where the pointer to the string is constant
  - where both are constant
- A `UniStr` string contains Unicode characters only. There are also pointers to `UniStr` pointers. The `UniStr`, the pointer to the `UniStr`, or both can be constants.

Environment Variables and Flags

The `ND_LANG` environment variable defines the native language for the application. When you want to change from one native language to another, you must reset this environment variable. This table shows the languages supported and the possible settings.

`ND_LANG`	Language	`ND_CHARNATIVE`
`enusasc`	US English.	No setting (the default)
`jajpeuc`	Japanese EUC.	`CT_JEUC`
`jajpsjis`	Japanese ShiftJIS.	`CT_SJIS`
`kokrksc`	Korean KSC.	`CT_KSC`
`zhtwbig5`	Taiwanese BIG5.	`CT_BIG5`
`zhtwcns`	Taiwanese CNS.	`CT_CNS`
`zhcngb`	Chinese GB.	`CT_GB`

ND_LANG	Language	ND_CHARNATIVE
enusutf8	For European Unicode / UTF8 version. The .dat file remains in English.	CT_UTF8
jajputf8	For global UTF8 version.	CT_UTF8
kokrutf8	For global UTF8 version.	CT_UTF8
zhtwutf8	For global UTF8 version.	CT_UTF8
zhcnutf8	For global UTF8 version.	CT_UTF8

Note: The Macintosh and the Power Macintosh do **not** support Unicode.

If you are using an Asian language, set the `ND_CHARNATIVE` environment variable when running OLE-based applications under Windows 95 and Windows NT. The table in this section shows the settings.

If you are using a European language, set the `ND_CHARNATIVE` environment variable to support PostScript printing. This list shows the possible settings:

- CT_ISO_LATIN1
- CT_ISO_LATIN2
- CT_ISO_LATIN3
- CT_ISO_LATIN4
- CT_ISO_CYRILLIC
- CT_ISO_ARABIC
- CT_ISO_GREEK
- CT_ISO_HEBREW
- CT_ISO_LATIN9

Character APIs in the Elements Environment

The APIs in the Elements Environment Char module enable you to manipulate characters and obtain information about them in these ways:

- Get a character code
- Obtain an ASCII character's classification
- Convert ASCII characters
- Convert characters between datatypes
- Convert between ASCII and EBCDIC
- Get a character length
- Get a specified byte of a character

Note: APIs for basic character classification enable you to obtain information such as whether the character is alphanumeric, hexadecimal, a control character, or a space. The `CHAR_AsciiIs` APIs assume that the character is in the specified C Runtime library (`C RTL`) classification.

This table shows the Char operations and the corresponding APIs.

Character Operation	API

Get a character code	`CHAR_GetByte`
	`CHAR_GetByte1`
	`CHAR_GetByte2`
	`CHAR_GetByte3`

Basic character classification	`CHAR_IsAscii`
	`CHAR_IsAsciiAlpha`
	`CHAR_IsAsciiUpper`
	`CHAR_IsAsciiLower`
	`CHAR_IsAsciiAlNum`
	`CHAR_IsAsciiDigit`
	`CHAR_IsAsciiHexDigit`
	`CHAR_IsAsciiOctDigit`
	`CHAR_IsAsciiSpace`
	`CHAR_IsAsciiPunct`
	`CHAR_IsAsciiControl`
	`CHAR_IsAsciiPrint`
	`CHAR_IsAsciiGraph`

Basic character conversion	`CHAR_AsciiDigitGetInt`
	`CHAR_AsciiHexDigitGetInt`
	`CHAR_AsciiOctDigitGetInt`
	
	`CHAR_AsciiAlphaGetBase`
	
	`CHAR_AsciiGetLower`
	`CHAR_AsciiGetUpper`
	
	`CHAR_AsciiGetControl`
	`CHAR_AsciiGetGraph`

Conversions between ASCII and EBCDIC	`CHAR_AsciiGetEbcdic`
	`CHAR_EbcdicGetAscii`
	
	`CHAR_ToAscii`
	`CHAR_FromAscii`

Get length	`CHAR_GetLen`
	`CHAR_CodeGetLen`
	`CHAR_NatGetLen`

Character Operation	API
Get byte	`CHAR_GetByte`

C RTL classification	`CHAR_IsAscii`
	`CHAR_AsciiIsAlpha`
	`CHAR_AsciiIsUpper`
	`CHAR_AsciiIsLower`
	`CHAR_AsciiIsAlNum`
	`CHAR_AsciiIsDigit`
	`CHAR_AsciiIsHexDigit`
	`CHAR_AsciiIsOctDigit`
	`CHAR_AsciiIsSpace`
	`CHAR_AsciiIsPunct`
	`CHAR_AsciiIsControl`
	`CHAR_AsciiIsPrint`
	`CHAR_AsciiIsGraph`
	`CHAR_AsciiDigitGetInt`
	`CHAR_AsciiOctDigitGetInt`
	`CHAR_AsciiHexDigitGetInt`
	`CHAR_AsciiAlphaGetBase`
	`CHAR_AsciiGetLower`
	`CHAR_AsciiGetUpper`

String APIs in the Elements Environment

The APIs in the Elements Environment Str module enable you to manipulate strings and obtain information about them in these ways:

- Extract characters and numeric values from strings
- Create, copy, and dispose of strings
- Set the contents of a string
- Append to strings
- Determine the string length
- Iterate through stings
- Write into strings
- Compare and match strings
- Search for characters in strings
- Find word boundaries
- Format numeric values
- Convert strings between upper and lower cases

Note: The Elements Environment provides separate APIs for strings and substrings.

This table shows the Str operations and the corresponding APIs:

String Operation	API

Get characters from strings	Native and `Ct` versions of these functions:
	`STR_GetCode`
	`STR_GetFwrd`
	`STR_GetBwrd`

Get numeric values from strings	`STR_GetDec...`
	`STR_GetHex...`
	`STR_GetRadix...`
	`STR_GetDouble`
	`STR_SubGetDec...`
	`STR_SubGetHex...`
	`STR_SubGetRadix...`
	`STR_SubGetDouble`

Create, clone, and dispose of strings	`STR_NewSet`
	`STR_NewSetSub`
	`STR_Clone`
	`STR_Dispose`
	`STR_Dispose0`

Set strings	`STR_Set` `STR_SetSub`
Append strings	`STR_Append` `STR_AppendSub`
Get string length	`STR_GetLen` `STR_GetTruncLen`

Iterate through strings	Native and `Ct` versions of these functions:
	`STR_GetCode`
	`STR_GetFwrd`
	`STR_GetBwrd`

Write into string buffers	Native versions of these functions:
	`STR_Put`
	`STR_PutSub`
	`STR_PutAscii`
	`STR_WriteAscii`
	
	`STR_PutCode`
	`STR_WriteCode`

Basic string comparisons	`STR_Cmp`
	`STR_CmpSub`
	`STR_Equals`
	`STR_EqualsSub`

String Operation	API

Match strings	`STR_MatchesChar`
	`STR_Matches`
	`STR_MatchesPat`
	`STR_MatchesSub`
	`STR_MatchesPatSub`

Search for character	Substring versions of `STR_Find...`:
	`STR_FindFirstChar`
	`STR_FindLastChar`
	`STR_FindFirst`
	`STR_FindLast`
	`STR_IFindFirst`
	`STR_IFindLast`
	`STR_FindIFirst`
	`STR_FindILast`

Find word boundaries	`STR_FindWord` `STR_FindWordSub`

Format numeric values	`STR_PutDec...`
	`STR_PutHex...`
	`STR_PutRadix...`
	`STR_PutDouble`

Basic conversions	`STR_AsciiUpCase`
	`STR_AsciiUpCaseSub`
	`STR_AsciiDownCase`
	`STR_AsciiDownCaseSub`
	`STR_UpCase`
	`STR_UpCaseSub`
	`STR_DownCase`
	`STR_DownCaseSub`

	`STR_PutAsciiUpper`
	`STR_PutAsciiLower`
	`STR_PutAsciiUpperSub`
	`STR_PutAsciiLowerSub`
	`STR_PutUpper`
	`STR_PutLower`
	`STR_PutUpperSub`
	`STR_PutLowerSub`

Variable-String APIs in the Elements Environment

The APIs in the Elements Environment VStr module enable you to manipulate variable strings and obtain information in these ways:

- Allocate and deallocate memory for variable strings
- Initialize and destroy variable strings
- Change the contents of variable strings
- Obtain the string length and string contents
- Concatenate, insert, and delete strings and characters
- Compare variable strings
- Load resources into variable strings
- Copy, initialize, and dispose of arrays

This table shows the VStr module's operations and APIs:

Operation	API

New and dispose	Native and `Ct` versions of `VSTR_New...`:
	`VSTR_New`
	`VSTR_NewSetStr`
	`VSTR_NewSetStrSub`
	`VSTR_NewSet`
	`VSTR_Clone`
	`VSTR_Dispose`
	`VSTR_Dispose0`

Initialization/destruction	Native and `Ct` versions of `VSTR_Init...`:
	`VSTR_Init`
	`VSTR_InitSetStr`
	`VSTR_InitSetStrSub`
	`VSTR_InitSet`
	`VSTR_End`

Changing contents	Native and `Ct` versions of `VSTR_Set...`:
	`VSTR_SetStr`
	`VSTR_SetStrSub`
	`VSTR_Set`
	`VSTR_Copy`

Queries	`VSTR_GetLen`
	`VSTR_GetStr`
	`VSTR_QueryStrSub`

Operation	API

Concatenation, insertion, deletion	`VSTR_AppendStr`
	`VSTR_AppendStrSub`
	`VSTR_Append`
	`VSTR_AppendChar`
	`VSTR_TruncAt`
	`VSTR_Truncate`
	`VSTR_Clear`

Comparisons	`VSTR_CmpStr` `VSTR_Cmp`

Loading resources	`VSTR_NewSetRes`
	`VSTR_InitSetRes`
	`VSTR_SetRes`

Operations for arrays of strings	`VSTR_ArrayClone`
	`VSTR_ArrayInit`
	`VSTR_ArrayEnd`
	`VSTR_ArrayReset`
	`VSTR_ArrayDispose`

Using Code Sets and Code Types

Code sets and code types are used in the process of mapping alphabetic characters into a numeric code in a 4-byte word.

Code-Set Operations and APIs

A *code set* is a numeric representation for each character in an alphabet. The Elements Environment code-set APIs are **not** public.

See "Code Sets" on page 140 for more information.

Code-Type Operations and APIs

Code types identify a complete character-coding system, which associates a code set with a mapping.

See "Code Types" on page 142 for more information.

The Ct module APIs use two types of data:

- `CtId` for the code-type ID
- `ChCode` for a code value within a code type

Code types are identified with a unique ID of the form `CT_`*XXX*.

See the Ct module for a complete list of code-type IDs.

The Ct module APIs allow you to:

- Create and destroy a code type
- Initialize a code type
- Get code-type IDs
- Iterate through a code type
- Convert to and from code sets

This table describes the code-type operations and the corresponding APIs:

Code-Type Operations	API

Create / destroy code type	`CT_New`
	`CT_Dispose`
	`CT_Dispose0`

Initialization	`CT_DefInit`

General APIs	`CT_GetCtId`
	`CT_GetCharLen`
	`CT_GetFwrd`
	`CT_GetBwrd`
	`CT_GetInfo`
	`CT_CvtChar`
	`CT_CvtCtToCs`
	`CT_CvtCsToCt`
	`CT_GetMaxCharLen`
	`CT_IsSingleOnly`
	`CT_GetUpper`
	`CT_GetLower`

About Unicode

Unicode is a worldwide character-encoding standard that includes characters used in most language types, such as:

- European
- Indic
- Arabic
- Asian

In addition, it includes various symbols, such as:

- Mathematical
- Technical
- Phonetic
- Punctuation

In the Unicode standard, a *character* is defined by 16 bits; therefore, there can be up to 65,536 characters. With Unicode, you can mix many languages in one application. You can also combine character and string processing for various languages.

Note: Windows NT uses Unicode internally, and some applications process Unicode internally.

Using Unicode

This section describes the problems with using Unicode and how the Elements Environment resolves those problems.

ASCII Compatibility

Since Unicode characters are coded in 16 bits, they are **not** compatible with ASCII characters. For example, ASCII "A" (0x41 in hexadecimal) is coded 0x0041 in Unicode.

Unicode strings can include "0x00" in the string. Programs using C-language string handling regard "0x00" as a string terminator; thus, they cannot handle Unicode strings. This means that an ASCII-based application cannot even display Unicode-encoded English strings.

Most applications supporting Unicode, except Windows NT, use pure Unicode. To keep ASCII compatibility and enable Unicode, without modifying existing API calls that take strings as arguments, the Elements Environment uses *Unicode Transformation Format* (UTF8) as an internal code type. UTF8 allows you to convert any Unicode value to UTF8, and UTF8 to Unicode. In UTF8, ASCII code keeps ASCII values. Also, this format does not include "0x00" in the string.

Note: UTF8 mode is available **only** if you set the `ND_CHARNATIVE` environment variable to `CT_UTF8`.

The following table shows the UTF8 encoding scheme. In UTF8, a character can be 1, 2, 3, 4, 5 or 6 bytes in length. The actual character value is the concatenation of the v bits. If the character is 1 byte in size, it is the same as ASCII. Code point 0 is allowed only in the 1-byte encoding.

Bytes	Bits	Hex Min	Hex Max	Byte Sequence in Binary
1	7	00000000	0000007F	0vvvvvvv
2	11	00000080	000007FF	110vvvvv 10vvvvvv
3	16	00000800	0000FFFF	1110vvvv 10vvvvvv 10vvvvvv
4	21	00010000	001FFFFF	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv

Bytes	Bits	Hex Min	Hex Max	Byte Sequence in Binary					
5	26	00200000	03FFFFFF	111110vv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	
6	31	04000000	7FFFFFFF	1111110v	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv

This specification supports up to 6-byte encoding. However, the Elements Environment supports only 1-, 2-, and 3-byte encoding.

Unicode characters whose code value is less than 0x80 are mapped to 1-byte encoding, which is the same as ASCII. Unicode characters whose value is equal to, or greater than, 0x80 and less than 0x800 are mapped to 2-byte encoding. Other Unicode characters are mapped to 3-byte encoding.

For more detailed information about UTF8, see this site on the World Wide Web:

`http://www.stonehand.com/unicode/standard/utf8.html`

Conversion with Existing Character Sets

Unicode includes many characters that are already defined in other standard code sets. Thus, it is possible to convert characters in existing code sets to Unicode.

However, there is no way to arithmetically convert Unicode to or from existing code sets except for ASCII, ISO8859-1 (used for Western European Latin characters), and a few other code sets.

Neuron Data provides functions to convert Unicode from and to existing character sets. This section gives an overview of the conversion functions. For a detailed description of these functions, see “Specifying Code Types for Unicode” on page 158.

To convert Unicode strings to or from current code types defined by `ND_CHARNATIVE`, use:

`STR_ToUni()` and `STR_FromUni()`

To convert one Unicode character to or from a specific code type, use:

`CT_ToUni()` and `CT_FromUni()`

To convert Unicode characters to a specified code set, use:

`CS_ToUni()` and `CS_FromUni()`

If you use Unicode as an intermediate code, it is possible to convert one code type to another code type. For example, you can convert MacRoman code type to Unicode and then convert it again to Windows ANSI code type. For those code-type conversions, use:

`STR_ToCt()` and `STR_FromCt()`

The available code types or code sets for these conversion APIs are described in “Code Types for Unicode Conversion” on page 163.

You need the conversion tables to convert Unicode from or to existing character sets except for ASCII and ISO Latin1. These tables usually reside in the data file named unicvt.tab. They are loaded automatically, when they are needed, so that they do not occupy memory all the time.

Font for Drawing and Printing

There are almost no Unicode-compatible fonts. Only Windows NT provides a Unicode font, which covers some European characters but not Indic, Arabic, and Asian characters.

For PostScript printing, though some font vendors provide some Unicode fonts, these do not fully support Unicode.

When you set the `ND_CHARNATIVE` variable to `CT_UTF8`, the `is_universal` clause is available. You can put any number of the native fonts in the font-family description language by using the `nextfont` keyword.

When the Elements Environment draws a UTF8 character, it tries to map the UTF8 code to a specified font code. If the UTF8 code successfully maps, the Elements Environment uses it for drawing. The order of mapping corresponds to the order of font descriptions.

Not all the fonts are available UTF8. For the available fonts for code mapping, see “Unicode Font Mapping” on page 164. This font-mapping function is available only on Windows 95, Windows NT, and UNIX/X Windows. Other platforms cannot draw UTF8 code.

The Elements Environment uses the same strategy for PostScript printing. It tries to map UTF8 code to the specified PostScript fonts. For the available PostScript fonts and their notation in the font-family description language, see “Unicode Font Mapping” on page 164.

Unicode Input

There is almost no Unicode-based input method. Windows NT provides a simple table-based Unicode input method, but it is difficult to use. The Elements Environment provides code-type conversions for these types of input:

- Keyboard input
- Clipboard input

Keyboard Input

The Elements Environment provides the ND_KEYBOARDCODETYPE environment variable, which specifies the keyboard-input code type. The Elements Environment assumes that the keyboard-input code type is the specified code type, and it converts each keyboard input to the internal code type specified by the ND_CHARNATIVE environment variable.

For example, if you have the Japanese ShiftJIS-based input method on your system and your Elements Environment application runs in UTF8 mode, you can specify ND_KEYBOARDCODETYPE to be CT_SJIS. Your keyboard input will then convert to UTF8.

In addition, you can:

- Dynamically change the keyboard code type by calling the EVENT_SetKbCt API
- Get the current keyboard code type by calling EVENT_GetKbCt

Clipboard Input

You can dynamically change the Elements Environment Clipboard input/output (I/O). You can set the default Clipboard code type with the ND_CLIPCODETYPE environment variable. You can then dynamically set and get it using the CLIP_SetCt and CLIP_GetCt APIs.

The Elements Environment provides a simple table-based Unicode input method as a sample program. This shows all Unicode characters in a list box, and you can select any character with the mouse and copy it to the Elements Environment Clipboard. You can copy and paste it into any Elements Environment application.

See “Examples” on page 162 for more information about this program.

Specifying Code Types for Unicode

ND_CHARNATIVE

ND_CHARNATIVE specifies the code type for the Elements Environment. If you specify CT_UTF8 for this variable, you can use UTF8-encoded code for all the Elements Environment APIs. If you do not set ND_CHARNATIVE, CT_ASCII is assumed, and it can only process ASCII codes.

These are other possible values for `ND_CHARNATIVE`:

`CT_ISO_LATIN1`	ISO8859-1 code type. Used for Western European languages for most of UNIX and X Windows.
`CT_SJIS`	For ShiftJIS. Used for Japanese Macintosh, PC, and UNIX.
`CT_JEUC`	For Japanese EUC. Used for most of Japanese UNIX.
`CT_KSC`	For Korean. Used for Korean.
`CT_GB`	For GB2312. Used for mainland China.
`CT_BIG5`	For BIG5. Used for Taiwan.
`CT_CNS`	For CNS. Used for Sun in Taiwan.

You will need a license to use some of these settings.

ND_KEYBOARDCODETYPE

`ND_KEYBOARDCODETYPE` specifies the keyboard-input code type. The Elements Environment assumes that all the codes are in this code type, and it converts the code to the internal code type specified by `ND_CHARNATIVE`.

If `ND_KEYBOARDCODETYPE` is not set, no keyboard-input conversion occurs.

If the keyboard-input code cannot be converted to the internal code, it is just ignored. No error message is displayed.

You can change this keyboard code type dynamically by calling **EVENT_SetKbCt()**. You can obtain the current keyboard code type by calling **EVENT_GetKbCt()**.

ND_CLIPCODETYPE

`ND_CLIPCODETYPE` specifies the Clipboard input code type. The Elements Environment assumes all Clipboard codes are in this code type, and it converts the code to the internal code type specified by `ND_CHARNATIVE`. In addition, it converts all Clipboard output from the internal code to the Clipboard code type.

If `ND_CLIPCODETYPE` is not set, no Clipboard I/O conversion occurs.

If the Clipboard I/O code cannot be converted, it is just ignored. No error message is displayed.

You can change this Clipboard code type dynamically by calling **CLIP_SetKbCt()**. You can obtain the current Clipboard code type by calling **CLIP_GetKbCt()**.

EVENT_SetKbCt, EVENT_GetKbCt

void EVENT_SetKbCt(CtIdEnum *ctid***);**

CtIdEnum EVENT_GetKbCt(void);

EVENT_SetKbCt() sets the keyboard code type. The Elements Environment converts all keyboard input codes to the internal code type based on this information.

EVENT_GetKbCt() returns the current keyboard code type.

CLIP_SetClipCt, CLIP_GetClipCt

void CLIP_SetKbCt(CtIdEnum *ctid***)**

CtIdEnum CLIP_GetKbCt(void)

CLIP_SetKbCt() sets the Clipboard code type. The Elements Environment converts all Clipboard I/O codes to/from the internal code type based on this information.

CLIP_GetKbCt() returns the current keyboard code type.

STR_ToUni, STR_FromUni

StrIVal STR_ToUni(UniStr *buf***, StrIVal** *unisize***, CStr** *str***, CharCvtSet** *flags***, StrCvtCtxPtr** *ctx***);**

StrIVal STR_FromUni(Str *buf***, StrIVal** *size***, UniCStr** *unistr***, CharCvtSet** *flags***, StrCvtCtxPtr** *ctx***);**

STR_ToUni() converts an internal string whose code type is defined by `ND_CHARNATIVE` to Unicode. It then puts the converted string into *buf*. The *unisize* is the size in 16-bit integers, **not** the size in bytes.

STR_FromUni() converts a Unicode string to an internal string. It then puts the converted string into *buf*. The *size* is the size of *buf* in 16-bit integers, **not** the size in bytes.

These routines always terminate *buf* with `NULL` bytes and never write more than `size` bytes into *buf*, including the terminating `NULL`. They return the number of characters that have been written to *buf* without terminating `NULL`.

Note: The *flags* are for future use to convert various characters defined in charpub.h. You can specify 0 for now.

ctx may be NULL or a pointer to a StrCvtCtx, which will be filled with *buf* and *str* positions where the conversion can resume after the destination buffer has been reallocated.

STR_ToCt, STR_FromCt

StrIVal STR_ToCt(NatStr *buf*, **StrIVal** *size*, **Cstr** *str*, **CtCPtr** *ct*, **StrCvtCtxPtr** *ctx***)**

StrIVal STR_FromCt(Str *buf*, **StrIVal** *size*, **NatCStr** *str*, **CtCPtr** *ct*, **StrCvtCtxPtr** *ctx***)**

STR_ToCt() converts *str* to *ct*-encoded string and puts the result into *buf*. **STR_FromCt()** converts *ct*-encoded *str* to an internal string and puts the result into *buf*.

The *size* is size of *buf*. These routines always terminate *buf* with NULL bytes and never write more than *size* bytes into *buf*, including the terminating NULL. They return the number of characters that have been written to *buf* without the terminating NULL.

ctx may be NULL or a pointer to a StrCvtCtx, which will be filled with *buf* and *str* positions where the conversion can resume after the destination buffer has been reallocated.

CT_ToUni, CT_FromUni

BoolEnum CT_ToUni(CtCPtr *ct*, **ChCode** *ch*, **UniCodePtr** *uni***)**

BoolEnum CT_FromUni(CtCPtr *ct*, **UniCode** *uni*, **ChCodePtr** *ch***)**

CT_ToUni() converts a *ct*-encoded *ch* code to Unicode *uni*.

CT_FromUni() converts a Unicode *uni* to *ct*-encoded *ch* code.

If the conversion succeeds, these functions return BOOL_TRUE; otherwise, they return BOOL_FALSE.

CS_ToUni, CS_FromUni

BoolEnum CS_ToUni(CsCPtr *cs*, **CsCode** *code*, **UniCodePtr** *uni)*

BoolEnum CS_FromUni(CsCPtr *cs*, **UniCode** *uni*, **CsCode** *code***)**

CS_ToUni() converts *code* in the *cs* codeset to Unicode *uni*.

CS_FromUni() converts Unicode *uni* to *code* of the *cs* code set.

If the conversion succeeds, these functions return BOOL_TRUE; otherwise, they return BOOL_FALSE.

Limitations

The Elements Environment does **not** support these Unicode features:

Nonspacing Marks

For example, the Elements Environment does not treat these as the same:

```
Unicode 0x0061 LATIN_SMALL_LETTER_A + Unicode 0x0308
NON_SPACING_DIAERESIS
```

```
Unicode 0x00E4 LATIN_SMALL_LETTER_A_DIARESIS
```

Right-to-Left Languages

You **cannot** use Arabic or Hebrew, for example, with the Elements Environment version of Unicode.

Language-specific Rendering Issues

The following, for example, are **not** supported:

- Korean syllable composing
- Thai characters' tone marks
- Arabic or Greek's context-dependent shapes
- Arabic ligatures

Platform-Specific Issues

On the Macintosh, OS/2 Presentation Manager, and Windows 3.1, UTF8 **cannot** be used as an `ND_CHARNATIVE` to specify the internal code.

Examples

You can find these examples in these directories:

- $ND_HOME/C/examples/unicode
- $ND_HOME/CPP/examples/unicode

tedit

This is a simple text editor for editing UTF8 strings. Keyboard input and Clipboard input can also be changed dynamically through menu selections.

The code type of the I/O file also can be specified. For example, it can read a Japanese EUC file and convert it to a UTF8 file.

uniin

This is an example of a simple Unicode input method. Users should set the `ND_CHARNATIVE` environment variable to `CT_UTF8` before starting this application.

This program displays all Unicode characters in the list box. Users can click any characters to select them and then click Copy to copy them to the Clipboard as a UTF8 string.

Any UTF8-based Elements Environment application can retrieve the copied strings with the Paste function text edit (CTRL-V).

Code Types for Unicode Conversion

You can convert these code types to / from Unicode:

`CT_ASCII`	ASCII code
`CT_ISO_LATIN1`	ISO8859-1 for Western Europe
`CT_ISO_LATIN2`	ISO8859-2 for Eastern Europe
`CT_ISO_LATIN3`	ISO8859-3 for Southeast Europe
`CT_ISO_LATIN4`	ISO8859-4 for Scandinavian
`CT_ISO_CYRILLIC`	ISO8859-5 for Russian
`CT_ISO_ARABIC`	ISO8859-6 for Arabic
`CT_ISO_GREEK`	ISO8859-7 for Greek
`CT_ISO_HEBREW`	ISO8859-8 for Hebrew
`CT_ISO_LATIN9`	ISO8859-9 for Turkish
`CT_MAC_ROMAN`	Macintosh code type for Western Europe
`CT_SJIS`	Japanese ShiftJIS Windows codepage 932
`CT_JEUC`	Japanese EUC
`CT_BIG5`	Taiwanese Big5 encoding
`CT_CNS`	Taiwanese CNS11643 EUC encoding
`CT_KSC`	Korean KSC5601 8-bit encoding
`CT_GB`	Mainland China’s 2312 8-bit encoding
`CT_ADOBE_STD`	Adobe Systems standard encoding
`CT_ADOBE_LATIN1`	Adobe Systems ISOLatin1 encoding
`CT_ADOBE_ZAPFDB`	Adobe Systems ZapfDingbats encoding
`CT_ADOBE_SYMBOL`	Adobe Systems Symbol encoding
`CT_MSW_ANSI`	MS Windows codepage 1252
`CT_MSW_EASTEURO`	MS Windows codepage 1250
`CT_MSW_CYRILLIC`	MS Windows codepage 1251
`CT_MSW_GREEK`	MS Windows codepage 1253
`CT_MSW_TURK`	MS Windows codepage 1254
`CT_MSW_ARABIC`	MS Windows codepage 1256
`CT_MSW_HEBREW`	MS Windows codepage 1255

Unicode Font Mapping

X11

The following X11 fonts can be used for Unicode (UTF8) code drawing on X11. (X11 fonts are listed by the last two fields of the XLFD description.)

`iso8859-1`	ISO8859-1 for Western Europe, Latin America
`iso8859-2`	ISO8859-2 for Eastern Europe
`iso8859-3`	ISO8859-3 for Southeast Europe
`iso8859-4`	ISO8859-4 for Scandinavian
`iso8859-5`	ISO8859-5 for Russian
`iso8859-6`	ISO8859-6 for Arabic
`iso8859-7`	ISO8859-7 for Greek
`iso8859-8`	ISO8859-8 for Hebrew
`iso8859-9`	ISO8859-9 Latin5 for Turkish
`jisx0201.1976-0`	Japanese Half-width Kana
`jisx0208.1983-0`	Japanese Kanji, and so on
`ksc5601.1987-0`	Korean KSC5601
`gb2312.1980-0`	Mainland China's GB2312
`big5.et-0`	Taiwanese Big5
`big5.eten-0`	Taiwanese Big5
`cns11643-1`	Taiwanese CNS11643-1
`cns11643-2`	Taiwanese CNS11643-2
`cns11643-3`	Taiwanese CNS11643-3
`unicode1.1-0`	Unicode font
`-dingbats`	Adobe Systems ZapfDingbats font

You can specify these fonts in the font-family description language (`x_charset "iso8859-1"`).

Note: You can get most of these X11 fonts from these URLs:

```
ftp://cair-archive.kaist.ac.kr/pub/hangul/fonts/
ftp://etlport.etl.go.jp/pub/mule/fonts/
ftp://ftp.ifcss.org/pub/software/fonts/{big5,cns,gb,misc}/
      bdf/
ftp://ftp.kuis.kyoto-u.ac.jp/misc/fonts/jisksp-fonts/
ftp://ftp.ora.com/pub/examples/nutshell/ujip/unix/
ftp://ftp.vszbr.cz/pub/X11-fonts
```

Windows 95 and Windows NT

You can use one of these methods to install Unicode fonts in Windows 95 and Windows NT:

- Install the "Lucida Sans Unicode" True Type font file included with Windows 95 and Windows NT. This font covers ASCII and most of the European code set, including Greek and Russian. Extract the file 1_10646.tt from the Windows 95 or Windows NT CD-ROM, and install it on your system.

  Note: If you cannot find this file, check other .tt files in the Windows 95 or Windows NT CD-ROM.

- Use the Trial or Times New Roman fonts to support European, Greek, and Russian characters.
- If you have third-party Unicode fonts, use them.
- Install code-page-specific True Type fonts from the International Windows 3.1 kit.

These CHARSET fonts can be mapped from UTF8. To use these fonts, install the font file from the Control Panel. Some double-byte fonts cannot be installed onto nonnative systems.

CHARSET Name	CHARSET Value
`ANSI_CHARSET`	0 (including Unicode)
`SHIFTJIS_CHARSET`	128
`HANGUL_CHARSET`	129
`GB2312_CHARGET`	134
`CHINESEBIG5_CHARSET`	136
`GREEK_CHARSET`	161
`TURKISH_CHARSET`	162
`HEBREW_CHARSET`	177
`ARABIC_CHARSET`	178
`RUSSIAN_CHASET`	204

PostScript

These PostScript fonts can be mapped from UTF8. To use these fonts for UTF8 PostScript printing, put the font name into the font-family description language using the listed keyword (for example, `ps_roman "Times"`):

PostScript Encoding	Keyword	Sample Font Name
`AdobeStandard`	`ps_roman`	Times, Helvetica, Courier
`Adobe Symbol`	`ps_symbol`	Symbol
`JISX0208`	`ps_j0208`	GothicBBB-Medium-H
`JISX0201`	`ps_j0201`	GothicBBB-Medium.Hankaku
`ZapfDingbats`	`ps_zapfdb`	ZapfDingbats
`KSC5601`	`ps_ksc5601`	KSC5601 7-bit encoded fonts
`JISX0212`	`ps_j0212`	JISX0212 7-bit encoded fonts
`BIG5`	`ps_big5`	Big5 encoded fonts
`GB2312`	`ps_gb2312`	GB2312 7-bit encoded fonts
`ISOLatin1`	`ps_isolatin1`	ISO8859-1 encoded fonts
`ISOCyrillic`	`ps_isocyrllic`	ISO8859-5 encoded fonts

Note: To enable UTF8 PostScript printing, the PostScript printer should have the font you specified.

For More Information about the APIs

- For detailed information about the language-independent APIs, see the Char, Str, VStr, Cs, and Ct modules in the *Elements Application Services C/C++ Programmer's Guide.*
- For multibyte API information, see the *Elements Application Services Programmer's Guide.*

Appendix

A PVCS Integration with the Elements Environment 2.1

The Elements Environment 2.1 and later versions support integration of the PVCS software configuration management (SCM) software. This software allows you to control the revision of source code in various ways, including:

- Privilege / access
- Login / logout
- Archiving

Note: The Elements Environment does **not** include the PVCS software. You must purchase the PVCS software separately from Intersolv or a licensed vendor.

The integration of the Elements Environment with PVCS allows you to use many PVCS functions without exiting the Elements Environment and using the PVCS interface. Instead, you have access to PVCS Level 1 (basic) integration from the File menu within any Elements Environment interface.

The available PVCS functions include:

- Configuring the Elements Environment so it can read necessary information, such as the location of archive files, from the PVCS configuration (.cfg) files
- Checking out source-code files
- Checking in source-code files
- Generating basic software-control status reports
- Managing revised source-code files located on your development platform

However, many other PVCS features are still controlled through the PVCS interface and not through the Elements Environment, including:

- User-access control
- Checkin authorization
- Build authorization
- Miscellaneous user privileges
- File access

Note: The current version of Elements Environment only supports PVCS version 5.2.*x*.

Note: Consult the *Intersolv PVCS User Guide and Reference* for specific information on configuring and using the PVCS software.

The Elements Environment 2.1 currently provides basic (Level 1) integration with PVCS. Neuron Data plans to offer advanced (Levels 2 and 3) integration in future releases of the Elements Environment.

Requirements for Using Level 1 PVCS Integration with the Elements Environment

When purchasing the Elements Environment software, you must:

- Obtain a license from Neuron Data to integrate PVCS with the Elements Environment
- Install the PVCS software on your system before installing the Elements Environment
- Have the Elements Environment 2.1 or a later version that includes the PVCS recognition module

To determine if PVCS integration is enabled:

1. Launch the Elements Environment.
2. Choose **File** from any Browser.
3. If PVCS is enabled, the PVCS option appears on the drop-down menu. If PVCS is not enabled, this option does not appear.

PVCS Features Supported in Level 1 Integration

- Default configuration files
- Checkout with locking and version labels
- Checkin with change description, user ID, date / time stamp, and version labels
- Access to project files and selection lists
- Grouping of files using version labels
- Generating basic reports
- Capturing PVCS error alerts and displaying them

PVCS Features Not Supported in Level 1 Integration

- Macintosh development platforms
- *Merging*—combining two sets of revisions to create a new source-code file
- *Branching*—developing alternate versions of source-code files simultaneously
- Multiple locking of files

Neuron Data plans to support these features in future releases of the Elements Environment.

PVCS Integration Tests

Every time you launch the Elements Environment, it:

1. Determines whether you have configured it to use PVCS integration
2. Checks to see if the PVCS libraries and development environment are on your system
3. Determines if you have a Neuron Data Elements Environment license for PVCS integration
4. Checks to see that you have enabled PVCS integration by setting `ND_PVCS` to “on” in ee.cfg, oie.cfg, and runscrpt.cfg

If these requirements are met, PVCS integration is enabled for the current development session.

Setting Up the PVCS Integration Environment

To activate PVCS integration, you must turn the environment variable `ND_PVCS` “on.” During installation of the Elements Environment, the environment variable `ND_PVCS` is declared in the files ee.cfg, oie.cfg, and runscrpt.cfg, but is set to “off.” To enable PVCS integration, open these text files and set `ND_PVCS` to “on.”

Note: The files ee.cfg, oie.cfg, and runscrpt.cfg are located in the \ee21\dat subdirectory.

Note: Do **not** confuse the Elements Environment configuration files with the PVCS project configuration files.

Accessing Integrated PVCS Options

You can choose integrated PVCS options from the File menu anywhere within the Elements Environment:

1. Choose **File - PVCS**.
2. Choose a PVCS option.

   Note: If you have not loaded a PVCS configuration (.cfg) file, all other options are disabled.

3. Go to the appropriate directory and select the file(s) you want.

   Note: Depending on the option you choose, only some of the files may appear.

   Tip: You can select multiple files by dragging or by **Shift + click**.

Configuring the Elements Environment for PVCS

You use the PVCS Configure option to tell the Elements Environment what PVCS configuration file (.cfg) to use. The Elements Environment uses the information from this file to determine:

- Your access privileges (based on your login ID)
- The location of your project archive files

Note: Before choosing any other PVCS option, you must first choose Configure so the Elements Environment knows which configuration file to use.

Tip: You do not have to enter your user ID. The Elements Environment gets your last login ID and uses that for comparison.

1. Choose **File - PVCS - Select Config File**.
2. Go to the appropriate directory and double-click the project's configuration file.

Checking Out Files

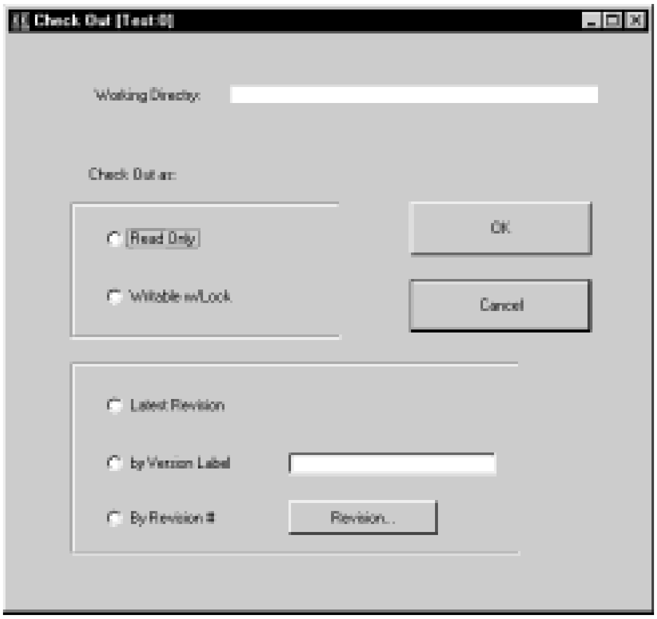
Checking out a file gives you access to it for browsing or editing. When you check out one or more files, PVCS:

- Allows you to select the file(s) to be checked out
- Checks your login ID against data in the project's PVCS configuration file to see if you are authorized to check out, write to, or lock the selected file(s)
- Notifies you if you are not authorized to perform these actions
- Allows you to lock files if you are authorized to do so

  Note: A locked file cannot be checked out by anyone else until you unlock it.
- Alerts you if another user has a requested file locked out
- Allows you to add version labels to group source-code files
- Allows you to check out files by revision level

To check out files:

1. Choose **File - PVCS - Check Out**.
2. Go to the appropriate directory and select the file(s) you want.
3. Click **OK**.



Note that the default settings on the Check Out window are **Read Only** and **Latest Revision**, and that the **by Version Label** and **by Revision #** options are disabled.

4. If you are authorized, you can select **Writable w/ Lock**.

   This allows you to make revisions to the file and prevents other users from accessing it until you unlock it. If you select this option, this enables the **by Version Label** and **by Revision #** options.

5. To add a new version label or select a current one, select **by Version Label** and enter the version label.

6. To check out a specific revision, select **by Revision #** and choose **Revision...**. Double-click the appropriate revision.
7. Click **OK** to check out the file.
8. If you selected multiple files in step 2, repeat steps 5–7 as appropriate.

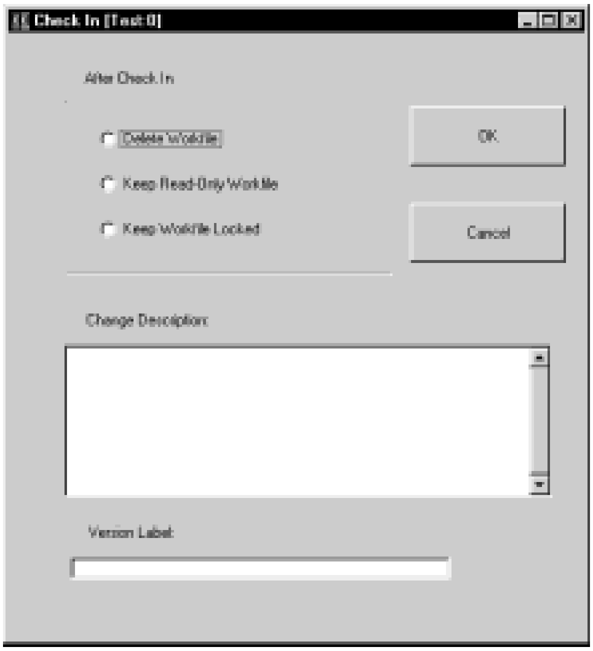
Checking In Files

Checking in a file lets you log a file back into the project archives after you have made revisions. Each time you check in a file, PVCS creates a new revision level for it. When you check in one or more files, PVCS:

- Lets you select file(s) to check in from the current working directory
- Checks your login ID against data in the project’s PVCS configuration file to see if you are authorized to check in the selected file(s)
- Notifies you if you are not authorized to check in the file(s)
- Allows you to add a revision description that is saved in the project archives
- Saves any unsaved work file(s)
- Updates revision information in the project archives
- Deletes the file(s) from your working directory if you selected that option
- Automatically unlocks checked-in file(s) to give other authorized users access

To check in files:

1. Choose **File - PVCS - Check In**.
2. Go to the appropriate directory and select the file(s) you want.
3. Click **OK**.



Note that the default setting on the Check In window is **Keep Read-Only Workfile**.

4. If you wish, change the default setting to **Delete Workfile** or **Keep Workfile Locked**

   If you select **Keep Workfile Locked**, the file will be checked in but other users will not have access to it until you check it in again with **Keep Workfile Locked** disabled.

5. If you want to add a description of your revisions, select **Change Description** and enter a description.
6. If you want to add a new version label, select **Version Label** and enter the version label.

   If the file already has a version label that is different from the version label you enter, you will be asked if you want to overwrite the version label. Click **OK** or **Cancel**.

7. Click **OK** to check in the file
8. If you selected multiple files in step 2, repeat steps 5–7 as appropriate.

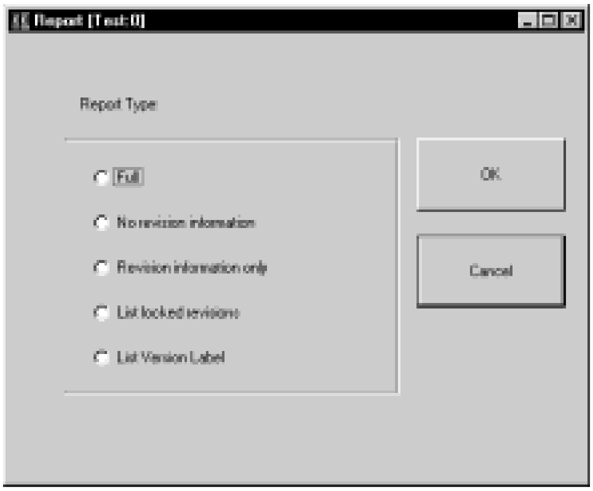
Generating Reports

The **Report** option allows you to create summaries about your project file(s) and / or revisions. When you use the Report option, PVCS:

- Lets you select archive file(s) to include in the report from the archive directory and subdirectories
- Checks your login ID against data in the project's PVCS configuration file to see if you are authorized to generate reports
- Notifies you if you are not authorized to generate reports
- Retrieves the PVCS log(s) on the file(s) you have selected
- Generates the report
- Saves the report in your current working directory

To generate reports:

1. Choose **File - PVCS - Archive Report**.
2. Go to the archive directory and select the archive file(s).
3. Click **OK**.



Note that the default setting on the Report window is **Full**, meaning that all reporting information is printed.

4. If you wish, select the appropriate option for a report containing only the information you want.
5. Click **OK** to accept your selection.

6. Enter the directory path and filename for the report.
7. Click **OK** to generate the report.

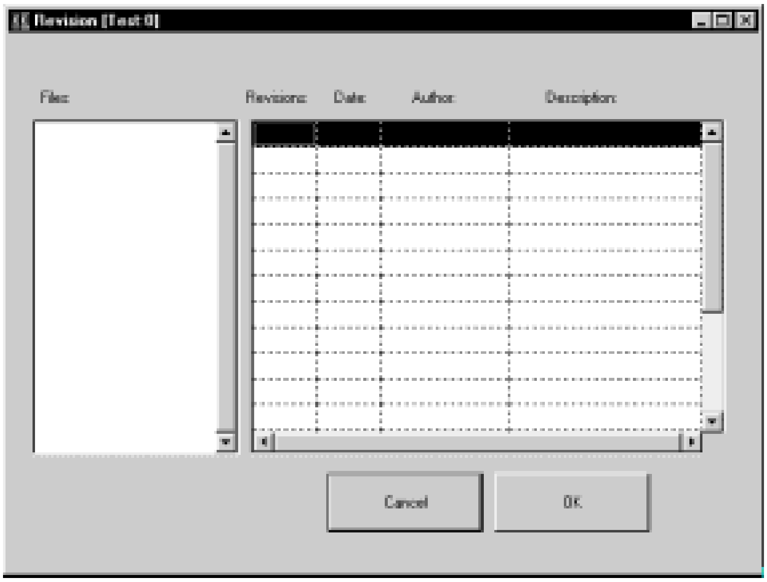
Deleting Revisions

The **Delete Revisions** option allows you to delete revisions you have made from the project archive.

Warning: Make sure you want to delete your revisions before proceeding.

To delete revisions:

1. Choose **File - PVCS - Delete Revisions**.



2. Select the revision you want to delete.
3. Click **OK** to delete the revision and all associated files.

   If you are unsure about deleting the revision, click **Cancel**.

Index