IBM WebSphere Host On-Demand Version 8.0

# Host On-Demand Macro Programming Guide

SC31-6378-00

IBM WebSphere Host On-Demand Version 8.0

# Host On-Demand Macro Programming Guide

SC31-6378-00

**Note**
Before using this information and the product it supports, read the information in Appendix B, "Notices", on page 181.

**First Edition (September 2003)**

This edition applies to Version 8.0 of IBM(R) WebSphere Host On-Demand (IBM Host Access Client Package for Multiplatforms V4.0, program number 5724–F69) and to all subsequent releases and modifications until otherwise indicated in new editions.

**© Copyright International Business Machines Corporation 2003. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents



© Copyright IBM Corp. 2003

# About this book

The *Macro Programming Guide* guide helps you write better Host On-Demand macros. This book is written for Host On-Demand macro developers and ordinary users. There are three parts.

Part 1, "Macro basics", on page 1 describes basic concepts, introduces the tools, and gives a step-by-step description of how to record and play back a simple macro.

Part 2, "Developing macros", on page 29 describes in detail the capabilities of the Host On-Demand macro system.

Part 3, "The macro language", on page 133 describes the XML macro language.

The *Macro Programming Guide* is also available on the Host On-Demand CD-ROM and at the Host On-Demand Web InfoCenter at http://www.ibm.com/software/webservers/hostondemand/library/v8infocenter/

## About the other Host On-Demand documentation

In addition to the *Macro Proramming Guide*, Host On-Demand also provides other sources of information to help you use the product. To access the documentation described here, go to the Host On-Demand library page at http://www.ibm.com/software/webservers/hostondemand/library.html . Most of the documentation is also included on the Host On-Demand product or Toolkit CD-ROMs.

- *Online Help*. The Online Help is the primary source of information for administrators and users after Host On-Demand installation is complete. It provides detailed steps on how to perform Host On-Demand tasks. A table of contents and an index help you locate task-oriented help panels and conceptual help panels. While you use the Host On-Demand graphical user interface (GUI), help buttons bring up panel-level help panels for the GUI.
- *Programming, Installing, and Configuring Host On-Demand*. Written for system administrators, this book helps you to plan for, install, and configure the Host On-Demand program.
- *Program Directory*. The program directory instructs you on how to install Host On-Demand on the z/OS and OS/390 platforms.
- *Readme file*. This file, readme.html, contains product information that was discovered too late to include in the product documentation.
- *Web Express Logon Reference*. This book provides a step-by-step approach for understanding, implementing, and troubleshooting Web Express Logon. It offers an overview of Web Express Logon, two scenario-based examples to help you plan for and deploy Web Express Logon in your own environment, as well as several APIs for writing customized macros and plug-ins.
- *Host Printing Reference*. After you configure host sessions, use the Host Printing Reference to enable your users to print their host session information to a local or LAN-attached printer or file.
- *Session Manager API Reference*. This book provides JavaScript APIs for managing host sessions and text-based interactions with host sessions.

© Copyright IBM Corp. 2003

- *Programmable Host On-Demand*. This book provides a set of Java APIs that allows developers to integrate various pieces of the Host On-Demand client code, such as terminals, menus, and toolbars, into their own custom Java applications and applets.
- *Toolkit Getting Started*. This book explains how to install and configure the Host On-Demand Toolkit, which is shipped with the Host Access Client Package, but is installed from a different CD-ROM than the Host On-Demand base product. The Host On-Demand Toolkit complements the Host On-Demand base product by offering Java beans and other components to help you maximize the use of Host On-Demand in your environment.
- *Host Access Beans for Java Reference*. This book is part of the Host On-Demand Toolkit. It serves as a reference for programmers who want to customize the Host On-Demand environment using Java beans and create macros to automate steps in emulator sessions.
- *Host Access Class Library Reference*. This book is part of the Host On-Demand Toolkit. It serves as a reference for programmers who want to write Java applets and applications that can access host information at the data stream level.
- *J2EE Connector Reference*. This book is part of the Host On-Demand Toolkit. It serves as a reference for programmers who want to write applets and servlets that access Java 2 Enterprise Edition (J2EE) compatible applications.
- *Host On-Demand Redbooks*. The Host On-Demand Redbooks complement the Host On-Demand product documentation by offering a practical, hands-on approach to using Host On-Demand. Redbooks are offered "as is" and do not always contain the very latest product information. For the most up-to-date list of all Host On-Demand Redbooks, visit the Host On-Demand library page at http://www.ibm.com/software/webservers/hostondemand/library.html.

## Conventions used in this book

The following typographic conventions are used in the *Macro Programming Guide*:

*Table 1. Conventions used in this book*

| Convention | Meaning |
|---|---|
| `Monospace` | Indicates text you must enter at a command prompt and values you must use literally, such as commands, functions, and resource definition attributes and their values. Monospace also indicates screen text and code examples. |
| *Italics* | Indicates variable values you must provide (for example, you supply the name of a file for *file_name*). Italics also indicates emphasis and the titles of books. |
| Return | Refers to the key labeled with the word Return, the word Enter, or the left arrow. |
| > | When used to describe a menu, shows a series of menu selections. For example, "Click File > New" means "From the File menu, click the New command." |
| | When used to describe a tree view, shows a series of folder or object expansions. For example, "Expand HODConfig Servlet > Sysplexes > Plex1 > J2EE Servers > BBOARS2" means:<br>1. Expand the HODConfig Servlet folder<br>2. Expand the Sysplexes folder<br>3. Expand the Plex1 folder<br>4. Expand the J2EE Servers folder<br>5. Expand the BBOARS2 folder |
| Java 1 | In this book, Java 1 means implemented in a Java 1.1.x JVM. |
| Java 2 | In this book, Java 2 means implemented in a 1.3 and later JVM. |

This graphic is used to highlight notes to the reader.

This graphic is used to highlight tips for the reader.

# Part 1. Macro basics

© Copyright IBM Corp. 2003

# Chapter 1. Introduction

## Host On-Demand macros

### Definition of a macro

A Host On-Demand macro is a XML script that allows a Host On-Demand client to interact automatically with a host application running on a terminal emulator session (a 3270 Display session, a 5250 Display session, a VT Display session, or a CICS Gateway session). A Host On-Demand macro is usually written to perform a specific task or set of tasks involving a single host application.

### Advantages of macros

Compared to a human operator, a macro interacts with a host application more quickly and with a smaller risk of error. A macro does not require that the person who runs it be trained in operating the host application. A macro can in many instances run unattended. A macro can be used again and again. A macro can be copied and distributed to multiple users. And most importantly, a macro can serve as a part of a broader solution to a customer requirement that involves both host applications and workstation applications.

### Unsophisticated users

An unsophisticated user can create a basic macro for automating a tedious or time-consuming interaction with a host application. The user navigates the host application to the screen at which he or she wishes to start, selects the Record Macro icon, and performs a task using the host application. For each application screen that the user visits, Host On-Demand records an impression of the application screen and the user's input to the application screen. When the user plays back the recorded macro starting at the same application screen as before, Host On-Demand recognizes each host application screen based on the previously recorded impression and repeats the actions that the human operator previously performed.

### Sophisticated users

A more sophisticated user can add to or improve a recorded macro using the Host Access Macro Editor (Macro Editor). This tool, which is available by clicking an icon on the session panel, provides a graphical user interface (consisting of input fields, text boxes, checkboxes, and so on) with which a user can modify or add features to each screen interaction with the host application. Besides allowing a user to edit and enhance the macro's screen recognition behavior and user input, the Macro Editor provides features that allow a user to add more intelligent behavior to the macro, such as choosing between alternate paths through an application, skipping a screen that should not be processed, or backing up to a previous screen. And there are more powerful capabilities including the ability to read and process data from the session screen, to notify the operator of status, to prompt the operator for an important decision, to download or upload files from the host, and to automate the printing of application screens.

### Programming features

The Macro Editor also provides programming features. A user with programming experience can add functionality to a Host On-Demand macro by creating and

© Copyright IBM Corp. 2003

manipulating variables, using arithmetic and logical expressions, writing if-then-else conditions, chaining to another macro, calling Java methods stored in external Java libraries, launching native applications, and writing trace information.

The Code Editor, a text editor that is launched from the Macro Editor, allows the user to view and modify the XML elements that make up the macro script, and also to cut and paste text through the system clipboard.

### Samples

This book contains macro code samples throughout. The chapter Chapter 15, “Sample macro code”, on page 173 contains an example of a macro that reads entries from a sample CICS database and writes them into a Microsoft Excel spreadsheet.

## Deploying macros

The Host On-Demand Deployment Wizard includes settings that enable system administrators to deploy macros to users in server libraries located at Web locations or on LAN or CD-ROM drives.

For more information see "Creating and deploying server macro libraries" in the document *Planning, Installing, and Configuring Host On-Demand*.

A local user can save a macro in a session configuration or a user directory if permitted.

## Using macros to integrate your enterprise applications

You can use Host On-Demand macros to integrate your Telnet-accessible applications with your workstation applications. Macros provide a path for data to flow into or out of your Telnet-accessible applications.

Host On-Demand includes two programming interfaces that allow you to drive macros:

- Programmable Host On-Demand

  This is a set of Java APIs that allows developers to integrate various pieces of the Host On-Demand client code, such as terminals, menus, and toolbars, into their own custom Java applications and applets.

  For more information see the document *Programmable Host On-Demand* in the Host On-Demand documentation.
- Session Manager APIs

  Session Manager APIs are JavaScript APIs for managing host sessions and text-based interactions with host sessions.

  For more information see the document *Session Manager API Reference* in the Host On-Demand documentation.

## Host Access Toolkit

The Host Access Toolkit is a separate product that provides programmatic control of the Host On-Demand client and other features, includes Java APIs for launching and interacting with Host On-Demand macros.

## Macros and security

Because a macro is an easily transportable, unencrypted, text-based representation of interactions between a user and a host application, you should consider protecting your macros as valuable pieces of intellectual property.

In particular, you should consider not storing passwords or other sensitive data in a macro script. Instead, you can design the macro so that it obtains sensitive information from an outside source, for example by prompting the user for a password or by obtaining data from a host or local application.

## This book focuses on 3270 applications

Although macros can be used with 3270 Display sessions, with 5250 Display sessions, with VT Display sessions, and with CICS Gateway sessions, this book focuses almost entirely on 3270 Display sessions and 3270 host applications.

# Chapter 2. Macro components

## Overview

This chapter describes the components that you will encounter as you use the macro capabilities of Host On-Demand. Here is an overview of the main components:

- **Macro Manager**. This a broad term that refers to all the user interfaces in Host On-Demand that allow you to work with macros. The Macro Manager consists of three main user interfaces:
  - Macro Manager toolbar. This is a toolbar with icons for commonly used macro functions, such as Record a Macro, Play Macro, Edit Macro Properties, and others. See "Macro Manager toolbar" in this chapter.
  - Macro Editor. This is the main user interface for editing a macro script. See "Macro Editor" on page 8 in this chapter.
  - Code Editor. This is a text editor that you lets you directly edit the XML language of a macro script. See "Code Editor" on page 9 in this chapter.
- **Macro runtime**. This is the program module that plays macros. See "Macro runtime" on page 10.
- **Macro object**. This is the underlying Java object that provides macro capabilities. See "Macro object" on page 10.

The remaining sections of this chapter describe these components in more detail. The final section of this chapter defines other macro terms that you might encounter in this book.

## Macro Manager

The Macro Manager is the umbrella term for all the macro user interfaces. There are three main interfaces: the Macro Manager toolbar, the Macro Editor, and the Code Editor.

### Macro Manager toolbar

The Macro Manager toolbar is a toolbar that contains icons for common macro operations. All of these icons can be used by macro developers, while at least one of the icons is also for users of macros (the Play macro icon). Figure 1 shows the Macro Manager toolbar. (For the purpose of illustration, this figure is inaccurate in one respect: it shows all the buttons on the toolbar enabled at the same time.)


*Figure 1. Macro Manager toolbar*

To view the Macro Manager toolbar follow these steps:

1. Start a Host On-Demand client.
2. Start a terminal emulator session (3270 Display session, 5250 Display session, VT session, or CICS Gateway session).
3. Click View > Macro Manager

© Copyright IBM Corp. 2003

Depending on your company's configuration of the display session, you might notice that a few of the icons on the Macro Manager toolbar also appear on the main toolbar. This placement is for extra convenience and should not cause you any concern. The icons work the same no matter which toolbar they appear on.

Here is a quick summary of the function of each part of the Macro Manager toolbar, from left to right. You will learn more about each of these functions later in the book.

- Currently selected macro. This white text field at the left side of the toolbar is not an input field but a text field in which the Macro Manager displays the name of the currently selected macro. Here the currently selected macro is ispf_usp.mac.
- Select a macro. This icon is the big downward-pointing arrowhead. Click this icon to select a current macro for playing, editing, copying, or deleting.
- Edit current macro properties. Click this icon to bring up the Macro Editor. See "Macro Editor"
- Delete current macro from list. Click this icon to delete the currently selected macro.
- Play macro. Click this icon to play the currently selected macro.
- Record macro. Click this icon to record a new macro.
- Stop playing or recording macro. Click this icon to end playing or recording a macro.
- Pause playing or recording macro. Click this icon to temporarily suspend the playing or recording of a macro.
- Add a prompt.
- Add a Smart Wait.
- Add an Extraction.

To step through the process of recording and playing back a simple macro, see Chapter 3, "Recording and playing back a simple macro", on page 13.

## Macro Editor

The Macro Editor (the full name is the Host Access Macro Editor) is a graphical user interface (with buttons, input fields, list boxes, and so on) for editing the parts of a macro. Figure 2 on page 9 shows the Macro Editor.

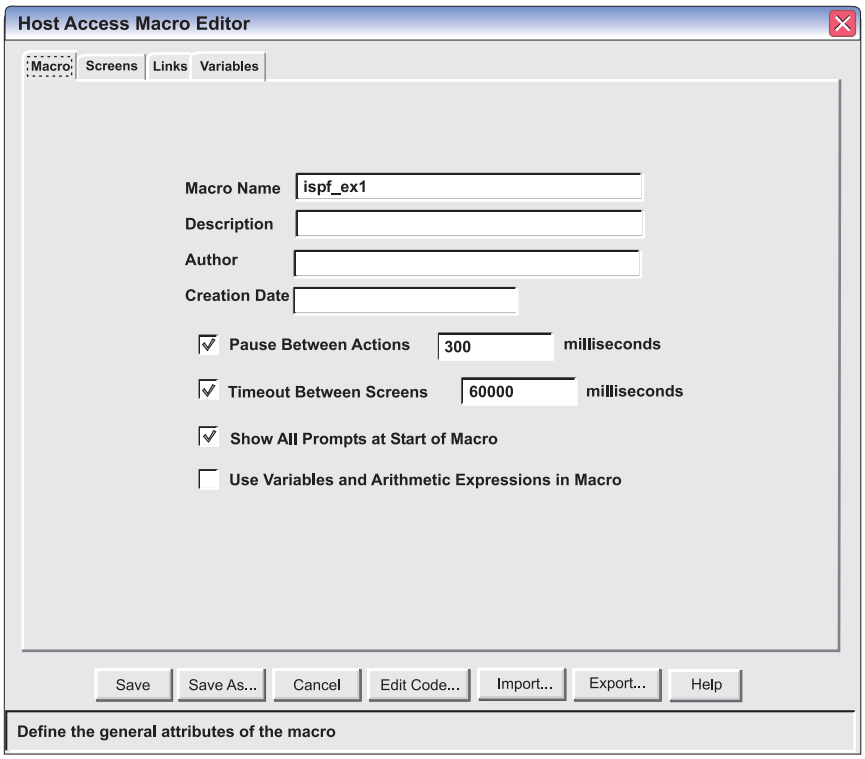


*Figure 2. Macro Editor*

You will probably use the Macro Editor for most of your macro development. It is more powerful (in terms of managing data more easily) than the Code Editor described in the next subsection, although it cannot do everything that the Code Editor can do.

To bring up the Macro Editor go to the Macro Manager toolbar.

1. Click the Select a macro icon to select a macro to edit.
2. Click the Edit current macro properties icon to edit the macro with the Macro Editor.

## Code Editor

Both the Macro Editor and the Code editor edit macros. Both tools read from and write to the same type of underlying macro source, an XML script. However, each tool is better for certain tasks.

The Macro Editor provides a graphical user interface that is powerful and user-friendly. It is the superior tool for creating and editing macros.

On the other hand, the Code Editor provides a text editor interface that allows you to edit directly the XML elements of which a macro is made. Figure 3 on page 10 shows the Code Editor displaying a macro script.

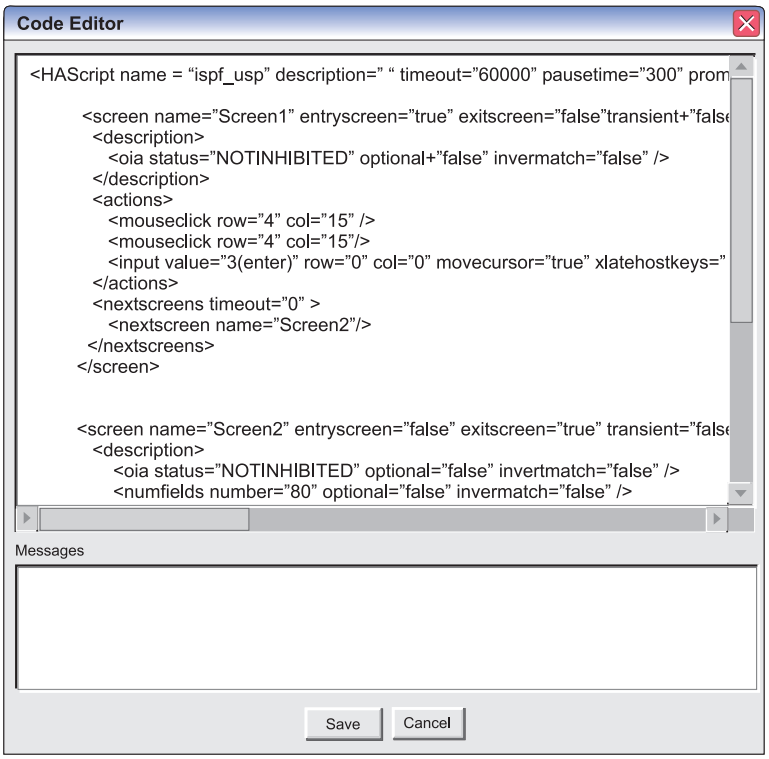


*Figure 3. Code Editor*

You should use the Code Editor for more technical editing tasks, such as:

- Modifying a few attributes of the XML elements that the Macro Editor does not provide access to.
- Looking at the XML elements to check your understanding of how a macro will execute.
- Debugging a macro.
- Cutting and pasting XML code from other sources using the Windows clipboard.

To bring up the Code Editor:

1. Use the Macro Editor to open the file that you want to edit.
2. Click Code Editor in the row of buttons at the bottom of the Macro Editor window.

## Macro runtime

The macro runtime is the program module that plays back a macro when a user clicks the Play Macro icon. Specifically, the macro runtime reads the contents of the current macro script and generates the macro playback.

## Macro object

The Macro object is the Java instance that provides the capabilities underlying the Macro Manager Toolbar, the Macro Editor, the Code Editor, and the macro runtime.

The IBM Host Access Toolkit, a separately purchased product, provides programming access to the Macro object through the many methods in the Macro class. This book does not describe how to use IBM Host Access Toolkit.

## Definitions of other terms

Here are the definitions of a few other terms that you will encounter in this book.

*Table 2. Definitions of terms*

| | |
|---|---|
| action | An action is an instruction that specifies some activity that the macro runtime is to perform when it plays back the macro (such as sending a sequence of keys to the session window, displaying a prompt in a popup window, capturing a block of text from the screen, and other actions). You can edit or create actions in the Macro Editor. You can view and modify individual action elements in the Code Editor. See Chapter 8, "Macro actions", on page 69. |
| application screen | An application screen is a meaningful arrangement of characters displayed on the Host On-Demand session window by a host application. See "Application screen" on page 23. |
| descriptor | A descriptor is an instruction that describes one characteristic of an application screen. You can edit or create descriptors in the Macro Editor. You can view and modify individual descriptor elements in the Code Editor. See "Introduction to the Description tab" on page 50. |
| macro screen | A macro screen is a set of instructions that tells the macro runtime how to manage a particular visit to a particular application screen. See "Macro screen" on page 24. |
| macro script | A macro script is an XML script in which a macro is stored. You can edit a macro script directly using the Code Editor or indirectly using the Macro Editor. When you play a macro, the macro runtime executes the instructions in the script. See "Macro script" on page 19. |
| valid next screen | A valid next screen is a macro screen that, during macro playback, is a valid candidate to be the next macro screen to be processed. See "Closer look at stage 1" on page 42. |

# Chapter 3. Recording and playing back a simple macro

The purpose of this chapter is to give you a hands-on introduction to the Macro Manager by stepping through three basic tasks:

- Recording a simple macro.
- Playing back the recorded macro.
- Assigning the playback of the macro to a particular key combination.

You can follow these same steps yourself by starting a Host On-Demand 3270 Display session, connecting to an MVS system, and logging on to TSO. The ISPF Primary Option Menu is the first application screen to appear after you log on (see Figure 5 on page 14).

Be sure to get permission from your systems administrator before doing this, and have an experienced ISPF user sitting next to you, if necessary.

## Recording a simple macro

This section shows you how to record a very simple macro. This macro changes the application screen from the ISPF Primary Option Menu to the Data Set List Utility screen (passing through the Utility Selection Panel screen on the way).

Before recording this macro, assure that:

- The ISPF Primary Option Menu is displayed on the session window. See Figure 5 on page 14.
- The Macro Manager toolbar is displayed above the session window. Figure 4 shows the Macro Manager toolbar (this is the same illustration that is displayed in Figure 1 on page 7).


*Figure 4. Macro Manager toolbar*

If the Macro Manager toolbar is not displayed on your system, click View > Macro Manager.

For more information on the Macro Manager toolbar, see "Macro Manager toolbar" on page 7.

To record the macro follow these steps:

1. Is TSO displaying the ISPF Primary Options screen? If not, then go to the ISPF Primary Options screen. See Figure 5 on page 14.
2. Click the Record macro icon to start recording. (This icon displays a single dot over an image of a cassette.)
3. The Record Macro window appears. Follow these steps:
   a. Click New.
   b. Type a name in the Name field, such as `ispf_ex1`.
   c. Type a description in the Description field, such as `Simple macro`.
   d. Under Save To, click Personal Library.

© Copyright IBM Corp. 2003

e. Click OK.

f. The Record Macro window disappears.

4. The ISPF Primary Option Panel is still displayed. See Figure 5.

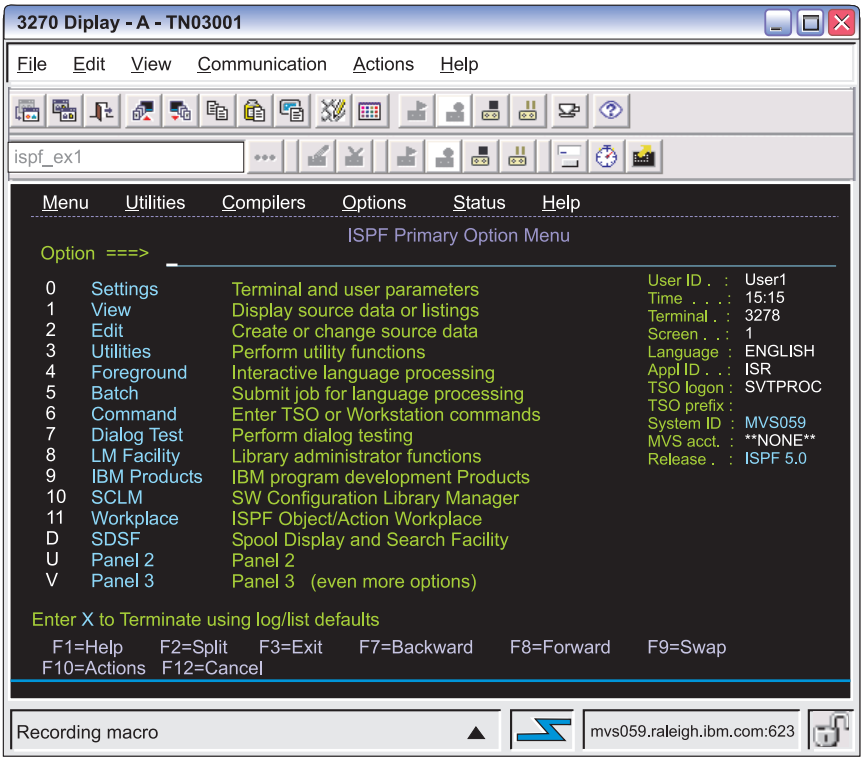


*Figure 5. The ISPF Primary Option Menu*

In the figure above, Host On-Demand is displaying the application screen and waiting for user input in the same way that it always does. But at the same time, the Macro object is waiting to record the user input when it occurs. There are two visual cues in the figure above that show that a macro is being recorded:

- In the status line at the bottom of the session panel is displayed the message Recording macro.
- On the Macro Manager toolbar the five icons on the right are enabled (Stop Macro, Pause Macro, and the three Add-A icons), while the five icons on the left are temporarily disabled. See Figure 5.

5. Click on the Option line near the top of the application screen. The Option line is the fourth line of the ISPF Primary Options Menu and begins at the left side of the screen with the label `Option ===>`. You should see the text cursor appear at the location that you clicked. Is the cursor displayed on the Option line? If not, then click again.
6. Type 3 and the enter key. As the figure above shows, 3 is the selection for Utilities.
7. The application screen changes to the Utility Selection Panel. See Figure 6 on page 15.

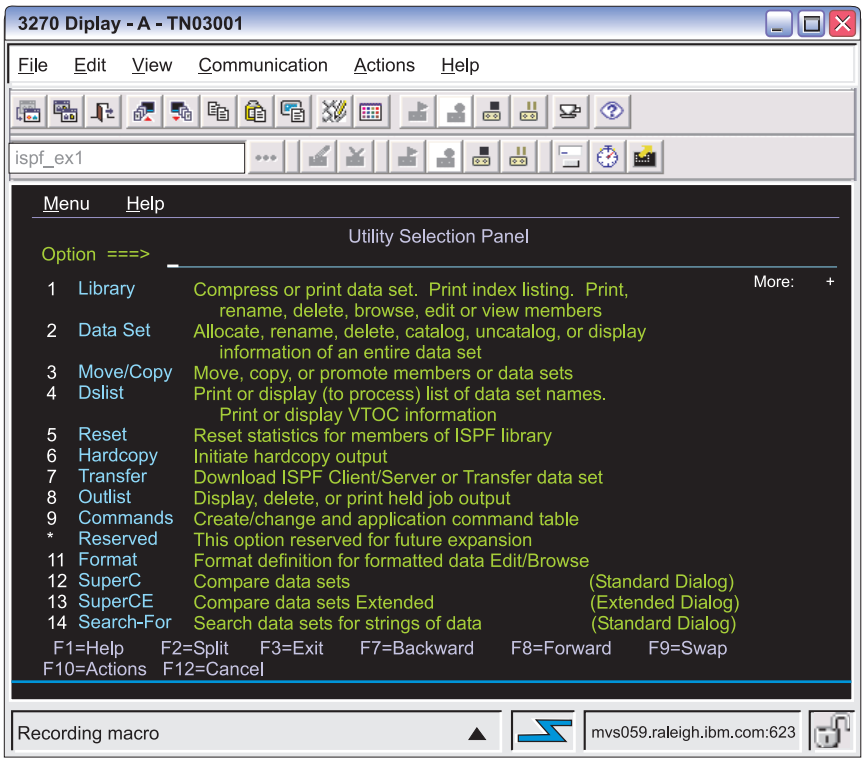


*Figure 6. The Utility Selection Panel application screen*

In the figure above, ISPF has automatically placed the text cursor on the Option line of this menu. On your screen, is the text cursor on the Option line? If not then click on the Option line to move the text cursor there.

8. Type 4 followed by the enter key. As the figure above shows, 4 is the selection for Dslist.
9. The application screen changes to the Data Set List Utility. See Figure 7 on page 16.

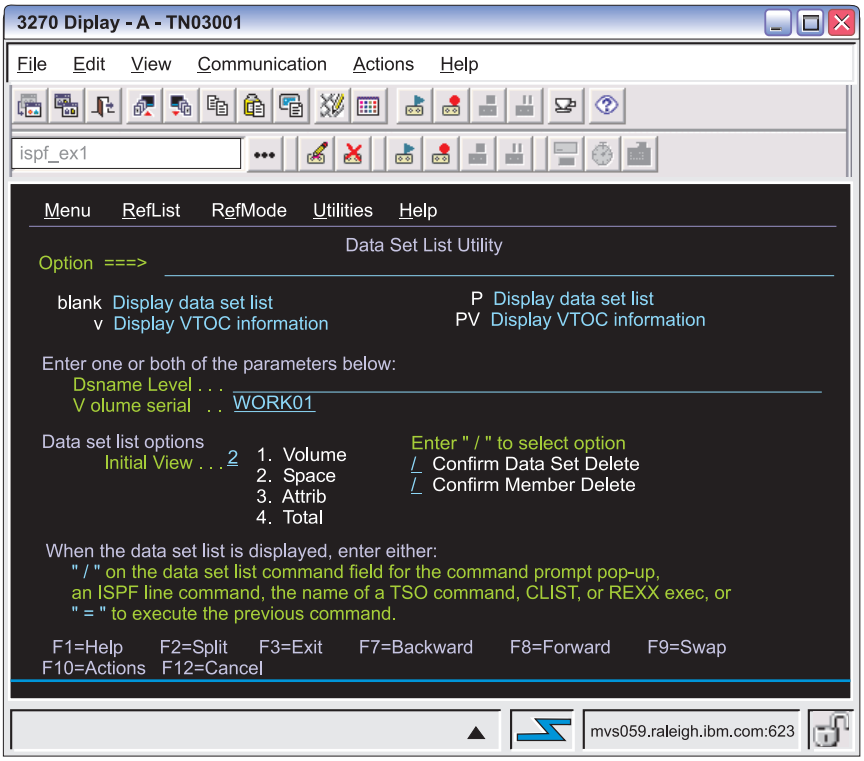


*Figure 7. The Utility Selection Panel application screen*

10. Click the Stop playing or recording macro icon to stop recording. This icon is the one that displays a black square with the image of a cassette below it. The five icons on the right side of the Macro Manager toolbar become disabled, and the five icons on the left side become enabled.
11. Recording is complete. The Data Set List Utility screen is displayed, as in Figure 7.

Some observations:

- You clicked the Record macro icon to start recording.
- You entered input (a mouse click and some keystrokes) just as you normally would when using the application, and the host application, ISPF, displayed the application screens and responded to the user input as it normally does.
- The Macro object recorded the application screens (or rather, a few identifying characteristics of each application screen) and recorded the mouse clicks and keystrokes as you entered them.
- You clicked the Stop recording or playing macro icon to stop recording the macro.

## Playing back a simple macro

This section shows how to play back the macro that you just recorded. Before you start, go back to ISPF Primary Option Menu. This is the starting point for this macro.

1. Verify that the application screen is the ISPF Primary Option Menu. See Figure 5 on page 14.
2. Select a macro to run. If you have just recorded the macro used in this example, then the name of the macro is displayed in the currently selected

macro field (the white text field on the left of the Macro Manager toolbar.) If not, then follow these steps to make that macro the currently selected macro:

- On the Macro Manager toolbar click Select a Macro. This icon is the large downward-pointing arrowhead.
- The Available Macros window appears.
- Under Macro Location click Personal Library.
- Under Macro List, click the name, such as `ispf_ex1.mac`, that you assigned to the macro recorded in the previous section of this chapter.
- Click OK

3. Verify that the name of the macro that you want to run is displayed as the currently selected macro.
4. Play the selected macro by clicking the Play macro icon. (This icon displays a small rightward-pointing arrowhead over the image of a cassette).
5. You should see the application screen change quickly to the Utility Selection Panel screen and then to the Data Set List Utility screen. Also, during the playback the icons on the left side of the Macro Manager toolbar are briefly disabled. After the playback is complete these icons are re-enabled.
6. Playback is complete.

Some observations:

- You had to position the application to a particular application screen before playing the macro. As is almost always the case with a simple macro, the starting point for playing back the macro is the point at which you started recording the macro. After all, this user input makes sense only in a certain context, the context in which it was recorded.
- You played the macro by:
  - Clicking the Select a macro icon and then selecting a particular macro.
  - Clicking Play macro to play the selected macro.
- While the macro played, the Macro runtime re-created the mouse click and keystrokes that it recorded earlier. The application responded as it normally does. The application could not tell the difference between a human entering input and the Macro runtime entering input during playback.
- By playing back the macro, you were able to accomplish the action of moving from one ISPF menu through another to a third menu quickly and accurately.

## Assigning the macro to a key combination

Host On-Demand allows you to assign a macro to a particular keystroke combination. To assign the macro that you just recorded to a keystroke combination, follow these steps:

1. Click Edit > Preferences > Keyboard. The Keyboard window appears.
2. Click the Key Assignment tab.
3. In the Category listbox select Macros.
4. In the list of macros select the name of the macro to which you want to assign a key, such as `ispf_ex1.mac` .
5. Click Assign a Key. The message Press a key is displayed.
6. Type Ctrl+i. After you type this key sequence, the label Ctrl+I is displayed beside the macro name.
7. Click Save to save this assignment.
8. Click OK to close the Keyboard window.

To play back the macro using the assigned key combination, follow these steps:

1. Position the application to the starting point for this macro, which is the ISPF Primary Option Menu.
2. Press Ctrl+i.
3. The macro is played back.

# Chapter 4. Macro structure

This chapter has two purposes, first to describe the general structure of a macro as it can be seen in an XML macro script, and second to show some of the connections between the Macro Editor and specific XML elements in the macro script.

- "Macro script" describes the <HAScript> element and its connections with the Macro tab of the Macro Editor.
- "The macro screen and its subcomponents" on page 23 describes the <screen> element and its connections with the Screens tab of the Macro Editor.

## Macro script

A macro script is an XML script used to store a Host On-Demand macro. You can view and edit the XML text of a macro script by using the Code Editor (shown in "Code Editor" on page 9). The Macro Editor displays the same information that you see in the Code Editor, but the Macro Editor displays the information in a more user-friendly format, using listboxes, checkboxes, input fields, and the other controls of the graphical user interface (see "Macro Editor" on page 8).

Learning a little about the XML elements of the macro language will greatly increase your understanding of important topics, including the following:

- How to use the Macro Editor.
- How macro playback works.
- How to build effective macros.

Therefore this book frequently refers not only to the input fields, buttons, and listboxes of the Macro Editor but also to the corresponding XML elements in which the same information is stored.

### XML elements

To understand macro scripts you do not need to learn a great deal about XML, just the basics of the syntax. If your knowledge of XML syntax needs brushing up, you can learn more about it in "XML syntax in the Host On-Demand macro language" on page 135. However, almost all of what you need to know is covered in this subsection.

As you probably know already, an XML script consists of a collection of XML elements, some of which contain other XML elements, in much the same way that some HTML elements contain other HTML elements. However, unlike HTML, XML allows a program developer to define new XML elements that reflect the structure of the information that the developer wishes to store. The Host On-Demand macro language contains approximately 35 different types of XML elements for storing the information needed to describe a macro. This macro language is described at length in Part 3, "The macro language", on page 133.

This book, when referring to an XML macro element, uses the element name enclosed in angle brackets. Examples: <HAScript> element, <screen> element.

Figure 8 on page 20 shows an example of an XML element:

© Copyright IBM Corp. 2003

```
<SampleElement attribute1="value1" attribute2="value2">
...
</SampleElement>
```

*Figure 8. Sample XML element*

The <SampleElement> element shown in the figure above contains the key components of every macro element. The first line is the begin tag. It consists of a left angle bracket (<), followed by the name of the XML element (SampleElement), followed by attribute definitions, followed by a right angle bracket (>). The second line consists of an ellipsis (...) that is not part of XML syntax but is used in the figure above to indicate the possible presence of other elements inside the <SampleElement> element. The third line is the end tag. It contains the name of the element enclosed in angle brackets with a forward slash after the first angle bracket (</Sample Element>).

In the begin tag, the attributes are specified by using the attribute name (such as attribute1), followed by an equals sign (=), followed by an attribute value enclosed in quotation marks (such as ″value1″). Any number of attributes can occur in the begin tag.

If the macro element does not contain other XML elements then it can be written in the shorthand fashion shown in Figure 9:

```
<SampleElement attribute1="value1" attribute2="value2"  />
```

*Figure 9. Sample XML element written in the shorthand format*

In the figure above the <SampleElement> element is written with a left angle bracket (<) followed by the name (SampleElement), followed by the attributes, followed by a forward slash and a right angle bracket (/>). Thus the entire XML element is written within a single pair of angle brackets.

## Conceptual view of a macro script

A macro script consists of a single <HAScript> element that can contain up to three major types of subelements:

- One <import> element. (Optional)
- One <vars> element. (Optional)
- One or more <screen> elements.

Figure 10 on page 21 shows a conceptual view of a sample macro script containing three <screen> elements.

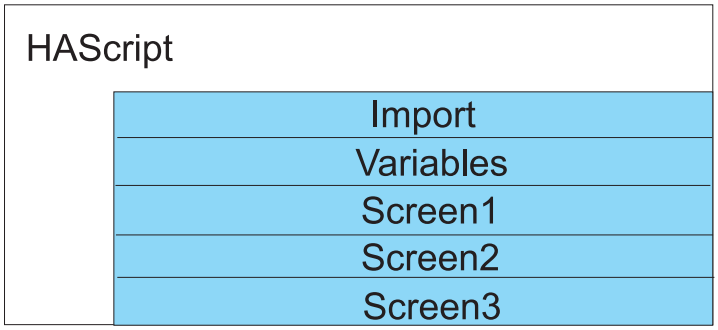


*Figure 10. Conceptual view of a macro script*

The figure above shows an <HAScript> element (HAScript) that contains instances of the major types of subelements: an <import> element (Import), a <vars> element (Variables), and three <screen> elements (Screen1, Screen2, and Screen3).

All macro scripts are structured like this, except that most have more screens. If there were 50 screens in the above macro, then the diagram above would look much the same, except that after Screen3 there would be additional screens: Screen4, Screen5, and so on, up to Screen50. (However, the order in which the screens are stored does not necessarily represent the order in which the screens are executed when the macro is played.)

The <HAScript> element is the master element of a macro script. (HAScript stands for Host Access Script.) It encloses the entire macro and also contains, in its begin tag, attributes that contain information applicable to the entire macro, such as the macro's name. For an example of an <HAScript> element see Figure 12 on page 23.

The <import> element is used to import Java classes and is optional. Importing Java classes is an advanced topic that is not discussed until "Creating an imported type for a Java class" on page 121.

The <vars> element is used to declare and initialize variables belonging to one of the standard data types (boolean, integer, double, string, or field). Using standard variables is an advanced topic that is not discussed until Chapter 11, "Variables and imported Java classes", on page 117.

The <screen> element is used to define a macro screen. The <screen> element is the most important of the elements that occur inside the <HAScript>. As you can see in Figure 10 above, a macro script is composed mostly of <screen> elements (such as Screen1, Screen2, and Screen3 in the figure). Also, most of the other kinds of XML elements in a macro script occur somewhere inside a <screen> element.

## Introduction to the Macro tab

For the purpose of getting you acquainted with the Macro Editor, this section consists of a very simple comparison between the Macro tab of the Macro Editor and the <HAScript> element described in the previous section.

The Macro Editor has four tabs: Macro, Screens, Links, and Variables. The first tab, the Macro tab, corresponds very closely to the <HAScript> element. In fact, the Macro tab is the graphical user interface for some of the information that is stored in the attributes of the begin tag of the <HAScript> element.

Therefore, as the <HAScript> element is the master element of a macro script and contains in its attributes information that applies to the entire macro (such as the macro name), similarly the Macro tab is the first tab of the Macro Editor and provides access to some of the same global information.

Figure 11 shows the Macro Editor with the Macro tab selected.

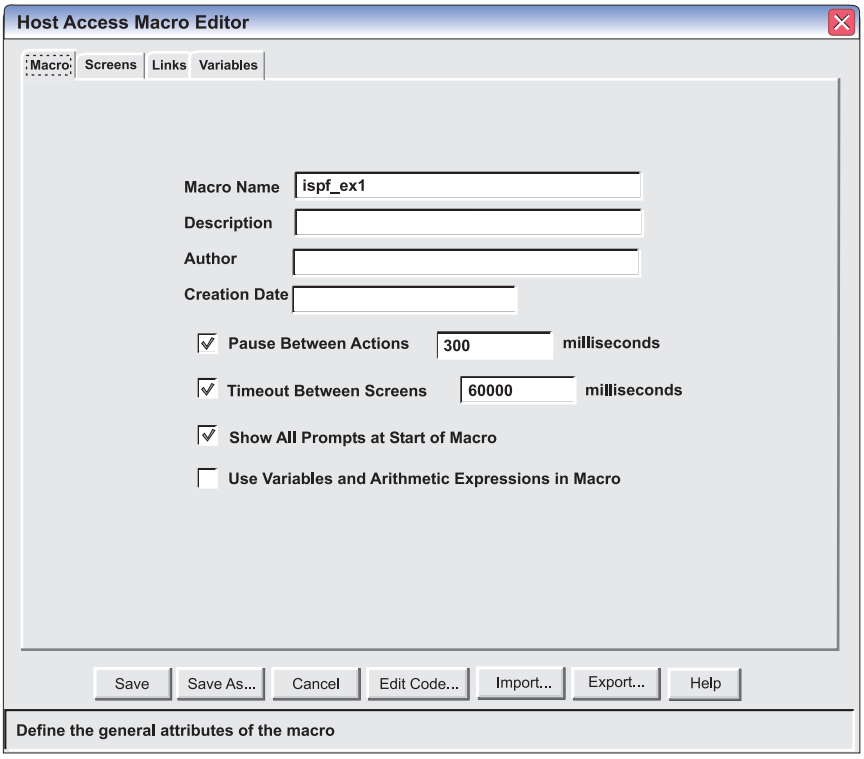


*Figure 11. Macro tab of the Macro Editor*

In the figure above you can see that the Macro tab has input fields for Macro Name, Description, and other information, along with several checkboxes. You should notice two fields:

- The Macro Name field contains the name that you assign to the macro. This is the same name that you will select when you want to edit the macro or run the macro.
- The Use Variables and Arithmetic Expressions In Macro checkbox determines whether the Macro object uses the basic macro format or the advanced macro format for this macro. In the figure above this checkbox is not selected, indicating that the basic macro format will be used (see "Choosing a macro format" on page 31).

Figure 12 on page 23 shows a sample <HAScript> element that contains the same information as is shown on the Macro tab in Figure 11, as well as some additional information. In the Code Editor an <HAScript> element is written on a single line, but here the element is written on multiple lines so that you can see the attributes.

```
<HAScript
     name="ispf_ex1"
     description=" "
     timeout="60000"
     pausetime="300"
     promptall="true"
     author=""
     creationdate=""
     supressclearevents="false"
     usevars="false"
     ignorepauseforenhancedtn="false"
     delayifnotenhancedtn="0">

...

</HAScript>
```

*Figure 12. A sample <HAScript> element*

In the <HAScript> element in the figure above you should notice that there is an attribute corresponding to each input field of the Macro tab shown in Figure 11 on page 22. For example, the **name** attribute in the <HAScript> element (`name="ispf_ex1"`) corresponds to the Macro Name field on the Macro tab. Similarly, the **usevars** attribute in the <HAScript> element (`usevars="false"`) corresponds to the Use Variables and Arithmetic Expressions checkbox on the Macro tab.

# The macro screen and its subcomponents

This section describes the macro screen and its major subcomponents. The definition of macro screen depends on another term that needs defining, application screen.

## Application screen

An application screen is a meaningful arrangement of characters displayed on the Host On-Demand session window by a host application.

As you probably realize, you are already very familiar with the concept of an application screen. An example of an application screen is the ISPF Primary Option Menu, which is displayed in Figure 13 on page 24. (This same application screen is displayed in Figure 5 on page 14.)

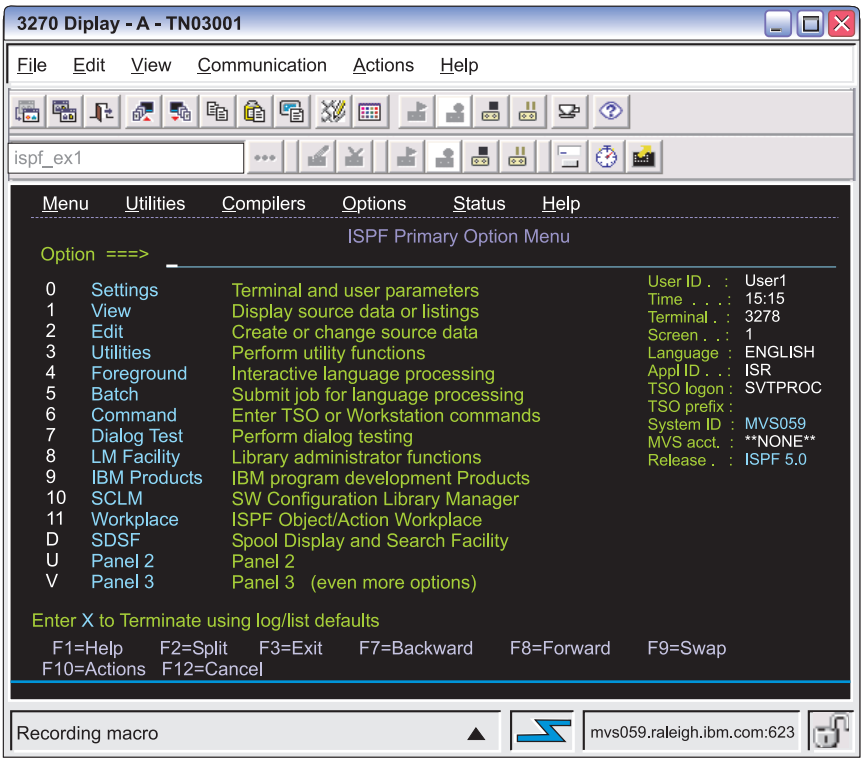


*Figure 13. A sample application screen, the ISPF Primary Option Menu*

In the figure above you can see that this application screen has menu selections displayed in a row across the top (`Menu`, `Utilities`, `Compilers`, `Options`, and so on), function key assignments displayed in a row across the bottom (`F1=Help`, `F2=Split`, and so on), a title near the top (`ISPF Primary Option Menu`), a list of options along the left side (`0` through `V`), and an input field in which to type an option number or letter (`Option ===>`). When the user provides input, for example by typing a 3 (for Utilities) followed by the enter key, the ISPF application removes all these visible items from the session window and displays a different application screen.

## Macro screen

A macro screen is a set of instructions that tell the macro runtime how to manage a visit to a particular application screen. A macro screen includes:

- A description of a particular application screen.
- The actions to take when visiting this particular application screen.
- A list of the macro screens that can validly occur after this particular application screen.

Although the concept is not very intuitive at this point, there might be in the same macro several macro screens that refer to the same application screen. Because of the way macro screens are linked to one another, the macro runtime might visit the same application screen several times during macro playback, processing a different macro screen at each visit.

Also, one macro screen might refer to more than one application screen. When several application screens are similar to each other, a macro developer might build a macro screen that handles all of the similar application screens.

Nevertheless, each macro screen corresponds to some application screen. When you record a macro, the Macro object creates and stores a macro screen for each application screen that you visit during the course of the recording. If you visit the same application screen more than once, the Macro object creates and stores a macro screen for each visit. Similarly, when you play back a recorded macro, the macro runtime processes a single macro screen for each application screen that it visits during the course of the playback.

## Conceptual view of a macro screen

A macro screen consists of a single <screen> element that contains three required subelements:

- One <description> element. (Required)
- One <actions> element. (Required)
- One <nextscreens> element. (Required, except in an Exit Screen)

Each of the subelements is required, and only one of each can occur.

Figure 14 shows a conceptual view of a <screen> element:

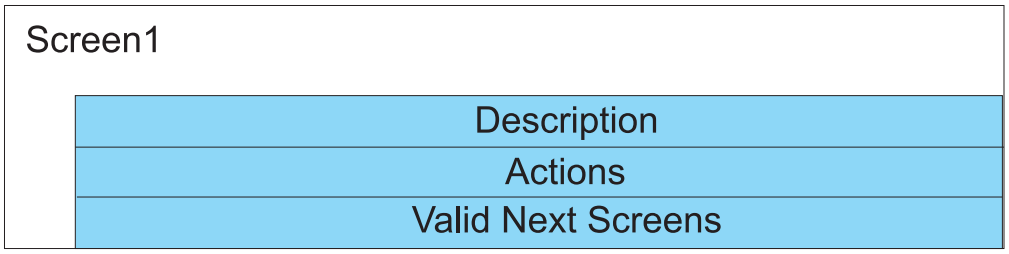


*Figure 14. Conceptual view of a <screen> element*

The figure above shows a <screen> element (Screen1) that contains the three required subelements: a <description> element (Description), an <actions> element (Actions), and a <nextscreens> element (Valid Next Screens).

All <screen> elements are structured in this way, with these three subelements. (A fourth and optional type of subelement, the <recolimit> element, is discussed later in this book.)

The <screen> element is the master element of a macro screen. It contains all the other elements that belong to that particular macro screen, and it also contains, in its begin tag, attributes that contain information applicable to the macro screen as a whole, such as the macro screen's name.

The <description> element contains descriptors that enable the macro runtime to recognize that the <screen> element to which the <description> element belongs is associated with a particular application screen. The descriptors and the <description> element are described in Chapter 7, "Screen description and recognition", on page 49.

The <actions> element contains various actions that the macro runtime performs on the application screen, such as reading data from the application screen or entering keystrokes. The actions and the <actions> element are described in Chapter 8, "Macro actions", on page 69.

The <nextscreens> element (Valid Next Screens in Figure 14 on page 25) contains a list of the screen names of all the <screen> elements that might validly occur after the current macro screen. The <nextscreens> element and the elements that it encloses are described in Chapter 9, “Screen Recognition, Part 2”, on page 103.

## Introduction to the Screens tab

This section shows some of the ways in which the Screens tab of the Macro Editor is related to the XML <screen> element described in the previous section. Figure 15 shows the Macro Editor with the Screens tab selected:

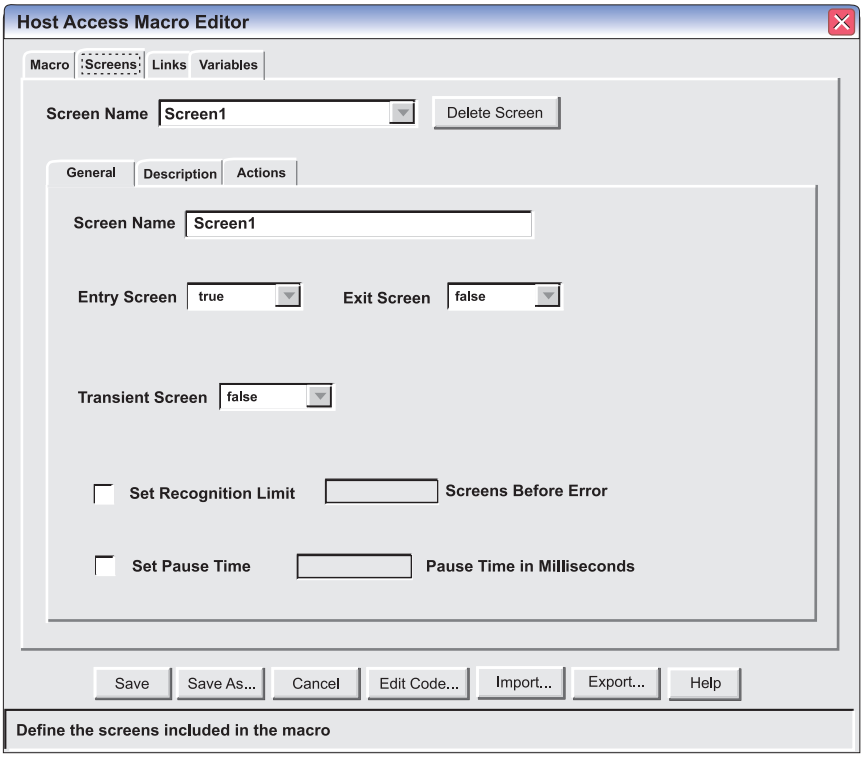


*Figure 15. Screens tab of the Macro Editor*

In the figure above, notice that the Screens tab contains:

- A Screen Name listbox at the top of the tab.
- Three subordinate tabs, labeled General, Descriptions, and Actions.

Currently the General tab is selected.

You should notice that there are two Screen Name fields on the Screens tab:

- The Screen Name field at the top of the Screens tab is a listbox that contains the names of all the macro screens in the macro.
- The Screen Name field at the top of the General subtab is an input field in which you can type the name that you want to assign to the currently selected screen.

In the Screen Name listbox at the top of the Screens tab, you click the name of the macro screen that you want to work on (such as Screen1), and the Macro Editor displays in the subtabs the information belonging to that macro screen. For example, in Figure 15 the listbox displays the macro screen name Screen1 and the subtabs display the information belonging to Screen1. If the user selected another

macro screen name in the listbox, perhaps Screen10, then the Macro Editor would display in the subtabs the information belonging to macro screen Screen10.

In the Screen Name input field under the General tab, you type the name that you want to assign to the currently selected macro screen. A screen name such as Screenx, where x stands for some integer (for example, Screen1), is a temporary name that the Macro object gives to the macro screen when it creates the macro screen. You can retain this name, or you can replace it with a more descriptive name that is easier to remember. (When all your macro screens have names such as Screen3, Screen10, Screen24, and so on, it is difficult to remember which macro screen does what.)

You have probably already noticed that the subtabs General, Description, and Actions on the Screens tab correspond to the main parts of the XML <screen> element described in the previous section. Specifically,

- The General subtab presents the information stored in the attributes of a <screen> element.
- The Description subtab presents the information stored in the <description> subelement of a <screen> element.
- The Actions subtab presents the information stored in the <actions> subelement of a <screen> element.

But what about the <nextscreens> subelement? For usability reasons the information belonging to the <nextscreens> element is presented in a higher-level tab, the Links tab. You can see the Links tab immediately to the right of the Screens tab in Figure 15 on page 26.

Figure 16 shows the XML begin tag and end tag of a sample <screen> element named Screen1:

```
<screen name="Screen1" entryscreen="true" exitscreen="false" transient="false">
...
</screen>
```

*Figure 16. Begin tag and end tag of a <screen> element*

In the figure above, the ellipsis (...) is not part of the XML text but indicates that the required elements contained inside the <screen> element have been omitted for simplicity. You should notice that the attributes in the begin tag correspond to fields on the General tab in Figure 15 on page 26. For example, the **name** attribute (`name="Screen1"` ) corresponds to the Screen Name input field on the General tab, and the **entryscreen** attribute (`entryscreen="true"`) corresponds to the Entry Screen listbox on the General tab.

Figure 17 on page 28 shows the XML text for the entire <screen> element including the enclosed elements:

```
<screen name="Screen1" entryscreen="true" exitscreen="false" transient="false">
   <description>
      <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
   </description>
   <actions>
      <mouseclick row="4" col="15" />
      <input value="3[enter]" row="0" col="0" movecursor="true"
                xlatehostkeys="true" encrypted="false" />
   </actions>
   <nextscreens timeout="0" >
      <nextscreen name="Screen2" />
   </nextscreens>
</screen>
```

*Figure 17. Sample XML <screen> element*

In the figure above you should notice that the <screen> element contains the required <description>, <actions>, and <nextscreens> elements.

# Part 2. Developing macros

© Copyright IBM Corp. 2003

# Chapter 5. Data types, operators, and expressions

## Choosing a macro format

### The basic macro format versus the advanced macro format

You must choose the format that you want your macro to be stored in: either the basic macro format or the advanced macro format.

The basic macro format is the default format. It supports a basic level of function but it does not include support for expression evaluation, variables, or some other features supported by the advanced macro format. Nevertheless, you should choose the basic macro format initially unless you already know that you are going to use expressions and variables. You can easily switch your macro to the advanced macro format later on. (In contrast, if you start out with the advanced macro format it is much more difficult to switch your macro to the basic macro format.)

You indicate which format you want with the Use Variables and Arithmetic Expressions in Macro checkbox on the Macro tab of the Macro Editor:

- Clear this checkbox to select the basic macro format (default).
- Set this checkbox to select the advanced macro format.

The basic macro format:

- Allows you to enter literal values, including integers, doubles, boolean (true or false), and strings.

In contrast the advanced macro format:

- Likewise allows you to enter literal values, including integers, doubles, boolean (true or false), and strings.
- Allows string concatenation using the '+' string operator.
- Allows arithmetic expressions.
- Allows conditional expressions.
- Allows variables.
- Allows imported Java variable types and methods

### Representation of strings and special characters, treatment of operator characters

You must write strings and the two special characters single quote (') and backslash (\) differently in the macro depending on whether you have chosen the basic macro format or the advanced macro format. Also, some characters that are ordinary characters in the basic macro format are used as operators in the advanced macro format.

However, these rules affect only input fields located on the following tabs:

- Description tab of the Screens tab
- Actions tab of the Screens tab
- Variables tab

© Copyright IBM Corp. 2003

The input fields that are affected on these tabs are as follows:

- Of course, text input fields on the surface of the tab, such as the Number of Fields text input field on the Field Counts and OIA window of the Description tab.
- But also, the text input field in the window that pops up when you select the <Expression> entry in a listbox on the tab, such as the <Expression> entry in the Ignore Case listbox on the String descriptor window.

For input fields on all other tabs than those listed above, always use the rules for the basic macro format.

The following two sections describe these differing rules.

### In the basic macro format, rules for representation of strings, etc.

If you have chosen the basic macro format, use the following rules for input fields on the Description tab, Actions tab, and Variables tab:

- Strings must be written without being enclosed in single quotes. Examples:

```
apple
banana
To be or not to be
John Smith
```

- The single quote (') and the backslash (\) are represented by the characters themselves without a preceding backslash. Examples:

```
New Year's Day
c:\Documents and Settings\User
```

- The following characters or character sequences are not treated as operators: +, -, *, /, %, ==, !=, >, <, >=, <=, &&, ||, !.

### In the advanced macro format, rules for representation of strings, etc.

If you have chosen the advanced macro format, use the following rules for input fields on the Description tab, Actions tab, and Variables tab:

- All strings must be written enclosed in single quotes. Examples:

```
'apple'
'banana'
'To be or not to be'
'John Smith'
```

- The single quote (') and the backslash (\) are represented by the characters themselves preceded by a backslash. Examples:

```
'New Year\'s Day'
c:\\Documents and Settings\\User
```

- The following characters or character sequences are treated as operators:
  - String concatenation operators: +
  - Arithmetic operators: +, -, *, /, %
  - Conditional operators: ==, !=, >, <, >=, <=
  - Logical operators: &&, ||, !

## Converting your macro to a different format

### Converting your macro to the advanced macro format

You can easily convert your macro from the basic macro format to the advanced macro format, just by checking the ″Use Variables and Arithmetic Expressions in Macro″ checkbox on the Macro tab. As a result the Macro object does the following:

- It enables all the advanced features for your macro.
- It automatically converts, in all input fields where conversion is required, all the strings in your macro and all occurrences of the two special characters single quote (') and backslash (\) from their basic representations to their advanced representations.

That is, the Macro object will find all the strings in your macro and surround them with single quotes, and the Macro object will change all occurrences of ' and \ to \' and \\. Also, any operator characters will be treated as operators.

### Converting your macro to the basic macro format

Converting your macro from the advanced macro format to the basic macro format can be very difficult. There are no automatic conversions when you uncheck the ″Use Variables and Arithmetic Expressions in Macro″ checkbox. The only automatic result is that the Macro object:

- Disables all the advanced features for the macro.

You yourself must change, one at a time, by hand, all the representations of strings and of the two special characters back to the basic representations. Also, you will have to delete any instances where advanced features have been used in the macro. If you do not do so then you might encounter errors or unexpected results when you try to save or run the script. Any remaining operator characters will be treated as literal characters rather than as operators.

---

## Standard data types

The Macro object supports the following standard data types:

- boolean
- integers
- doubles
- strings

These types are described in the subsections below.

## Boolean data

The boolean values `true` and `false` can be written with any combination of uppercase and lower case letters (such as True, TRUE, FALSE, falsE, and so on).

An example of a input field that requires a boolean value is the Entry Screen field on the General tab of the Screens tab. Enter `true` to set the condition to true or `false` to set the condition to false.

### Boolean values are not strings

The boolean values are not strings and therefore never need to be enclosed in single quotes. To repeat, whether you use the basic macro format or the advanced macro format, booleans are always written `true` and `false`, not `'true'` and `'false'`.

However, string values are converted to boolean values in a boolean context (see "Conversion to boolean" on page 37). Therefore with the advanced macro format you could enter the string `'true'` in a boolean field, because the Macro Editor would convert the string `'true'` to the boolean value `true`.

## Integers

Integers are written without commas or other delimiters. Examples:

```
10000
0
-140
```

### Integer constants

The Macro Editor has a number of integer constants that are written using all uppercase characters. These values are treated as integers not strings. Examples:

- NOTINHIBITED
- FIELD_PLANE
- COLOR_PLANE

## Doubles

Doubles are written without commas or other delimiters. Examples:

```
3.1416
4.557e5
-119.0431
```

## Strings

A string is any sequence of characters and can include leading, trailing, or intervening blank characters. Strings in some input fields must be represented differently depending on whether the macro has been set to use the basic macro format or the advanced macro format. See "Representation of strings and special characters, treatment of operator characters" on page 31.

The following examples use the representation for the advanced macro format:

```
'apple'
'User4'
'Total number of users'
'   This string has 3 leading blanks.'
'This string has 3 trailing blanks.   '
```

Here are the same examples using the representation for the basic macro format.

```
apple
User4
Total number of users
   This string has 3 leading blanks.
This string has 3 trailing blanks.
```

Notice that with the basic macro format trailing blanks are still allowed but are difficult to detect. If in doubt see the representation of the string in the Code Editor.

# Fields

See "Field variables" on page 123.

## The value null

The value `null` is a reserved word, not a string. When used in place of an object belonging to an imported Java class, it has the same meaning as it does in the Java language.

Do not use `null` to signify an empty string. To signify an empty string, use a pair of single quotes (’’) in the advanced macro format, or nothing at all in the basic macro format. If you use the value `null` in a string context (for example, by assigning it to a string variable), then the Macro Editor or the macro runtime will convert the value `null` to the string ’null’.

## Arithmetic operators and expressions

In order to use arithmetic expressions you must first check the ″Use Variables and Arithmetic Expressions in Macro″ checkbox on the Macro tab (see “Representation of strings and special characters, treatment of operator characters” on page 31).

### Operators and expressions

The arithmetic operators are:

*Table 3. Arithmetic operators*

| Operator | Operation |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulo |

In an arithmetic expression the terms are evaluated left to right. The order of precedence of the operators is: *, /, %, +, -. For example, the result of :

```
4 * 2 + 16 / 8 - 1 * 2
```

is 8. You can use parentheses to indicate the order in which you want expressions to be evaluated:

```
(4 * 2) + (16 / 8) - (1 * 2)    evaluates to 8
but
4 * (( 2 + 16) / (8 - 1)) * 2   evaluates to 20.571
```

### Where arithmetic expressions can be used

You can use an arithmetic expression almost anywhere that you can use an arithmetic value. Examples:

- As a parameter for a screen. For example,
  - To specify a recognition limit for a screen.
  - To specify a pause time for a screen.
- As a parameter for a descriptor. For example,
  - To specify a row or column in cursor descriptor.
  - To specify the number of fields in a numeric field count descriptor.
  - To specify a row, column, or attribute value in an attribute descriptor.
  - As a term in a conditional descriptor.

- As a parameter in an action. For example,
  - To specify a row or column in a mouse click action.
  - To specify a row or column in a box selection action.
  - As the number of milliseconds in a pause action.
  - To specify a value in a variable update action.
  - As a term in a conditional action.
- As an initial value for a variable.

## String concatenation operator (+)

You can use the string concatenation operator '+' only if you check the "Use Variables and Arithmetic Expressions in Macro" checkbox on the Macro tab. See "The basic macro format versus the advanced macro format" on page 31.

### Operators and expressions

The string operators are shown in the table below.

*Table 4. Arithmetic operators*

| Operator | Operation |
|---|---|
| + | Concatenate |

You can write a string expression containing multiple concatenations. The following examples use the string representation required for the advanced format (see "Representation of strings and special characters, treatment of operator characters" on page 31).

```
Expression:              Evaluates to:

'Hello ' + 'Fred' + '!'   'Hello Fred!'
'Hi' 'There'              (Error, a + operator is required to concatenate strings)
'Hi' + 'There'            'HiThere'
```

## Conditional and logical operators and expressions

The conditional operators are:

*Table 5. Conditional operators*

| Operator | Operation |
|---|---|
| == | Equal |
| != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

The logical operators are:

*Table 6. Logical operators*

| Operator | Operation |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

If you are entering && in an HTML or XML editor you might have to enter &&

In a conditional expression the terms are evaluated left to right. The order of precedence of the operators is the same order in which they are listed in the tables above. You can use parentheses to indicate the order in which you want expressions to be evaluated. Examples:

```
Expression:                 Evaluates to:

(4 > 3)                     true
!(4 > 3 )                   false
(4 > 3) && (8 > 10)         false
(4 > 3) || (8 > 10)         true
```

### Conditional expression can include complex terms

A conditional expression can contain arithmetic expressions, variables, and calls to methods of imported Java classes.

### Where conditional expressions can be used

Conditional and logical operators can be used only in two contexts:

- The Condition field of a conditional descriptor
- The Condition field of a conditional action

## Automatic data type conversion

### Effect of context

If an item of data belongs to one standard data type (boolean, integer, double, or string) but the context requires a different standard data type, then when the data is evaluated (either when the Macro Editor saves the data or else when the macro runtime plays the macro) it is automatically converted to the standard data type required by the context, if possible.

Examples of context are:

- The Condition field of a Condition descriptor (expects a boolean value)
- The Message Text field of a Message action (expects a string value)
- The Value field of a Variable update action when the variable is a field variable (expects a location string)
- The Row value of an Input action (expects an integer value)

However, if the data cannot be converted to the new data type (for example, the string `123apple` cannot be converted to an integer) then an error occurs. The Macro Editor displays an error message. The macro runtime stops the macro playback and displays an error message.

The following subsections discuss the conversions that can occur for each standard data type.

### Conversion to boolean

The string `'true'` (or `'TRUE'`, `'True'`, and so on) in a boolean context is converted to boolean true. Any other string in a boolean context (including `'false'`, `'1'`, `'apple'`, and any other) is converted to boolean false.

```
'true' (in an input field that requires a boolean)  converts to true
'apple' (in an input field that requires a boolean)  converts to false
```

## Conversion to integer

A string in valid integer format and occurring in an integer context converts to integer.

```
'4096'                converts to       4096
'-9'                  converts to       -9
```

## Conversion to double

A string in valid double format occurring in a double context converts to double.

```
'148.3'                converts to 148.3
```

An integer combined with a double results in a double:

```
10 +  6.4             evaluates to 16.4
```

## Conversion to string

A boolean, integer, or double in a string context converts to a string. (Remember, the boolean values true and false are not strings. See "Boolean data" on page 33.)

```
'The result is ' + true                         evaluates to 'The result is true'
FALSE (in an input field that requires a string) converts to 'FALSE'
'The answer is ' + 15                           evaluates to 'The answer is 15'
22 (in an input field that requires a string)   converts to  '22'
('4.5' == .45e1)                                evaluates to true
14,52 (in an input field that requires a string) evaluates to'14,52'
```

## Conversion errors

If the context requires a conversion but the format of the data is not valid for the conversion then the Macro Editor displays an error message. If the error occurs while a macro is playing then the macro runtime display an error message and terminates the macro with a run-time error.

```
'123apple'       in an integer context    Error
'22.7peach'      in a double context      Error
```

# Equivalents

Any context that accepts an immediate value of a particular standard data type also accepts any entity of the same data type.

For example, if an input field accepts a string value, such as `'Standard Dialog'`, it also accepts:

- An expression that evaluates to a string.
- A value that converts to a string.
- A string variable.
- A call to an imported method that returns a string.

Similarly, if an input field accepts a boolean value (true or false), it also accepts:

- An expression that evaluates to a boolean value.
- A value that converts to a boolean value.
- A boolean variable.
- A call to an imported method that returns a boolean.

Recognizing this flexibility in the macro facility will help you write more powerful macros.

## Significance of a negative value for a row or column

In the String descriptor and in several other descriptors and actions, a negative value for a row or column of the session window indicates an offset from the last row or the last column of the session window. The macro runtime calculates the row or column location as follows:

```
actual row    = (number of rows in text area)    + 1 + (negative row offset)
actual column = (number of columns in text area) + 1 + (negative column offset)
```

For example, if the session window has 24 rows of text then a row coordinate of -1 indicates an actual row coordinate of 24 (calculated as: 24 + 1 - 1). Similarly if the session window has 80 columns of text then a column coordinate of -1 indicates an actual column coordinate of 80 (calculated as 80 + 1 - 1).

The row calculation above ignores the OIA row. For example, if the session window is 25 rows high, it has only 24 rows of text.

The advantage of this convention is that if you want to specify a rectangle at the bottom of the session window, then this calculation gives the right result whether the session window has 25, 43, or 50 rows. Similarly, if you want to specify a rectangle at the right side of the session window, then this calculation gives the right result whether the session window has 80 columns or 132 columns.

The following tables shows the results for a few calculations:

*Table 7. Negative value for row*

| Negative value for row: | Actual value in session window with 24 columns of text (OIA row is ignored): | Actual value in session window with 42 columns of text (OIA row is ignored): | Actual value in session window with 49 columns of text (OIA row is ignored): |
|---|---|---|---|
| -1 | 24 | 42 | 49 |
| -2 | 23 | 41 | 48 |
| -3 | 22 | 40 | 47 |

*Table 8. Negative value for column*

| **Negative value for column:** | **Actual value in session window with 80 columns:** | **Actual value in session window with 132 columns:** |
|---|---|---|
| -1 | 80 | 132 |
| -2 | 79 | 131 |
| -3 | 78 | 130 |

Whether you make use of this convention or not, you should at least remember that a rectangular area with coordinates of (1,1) and (-1,-1) means the entire text area of the session window.

# Chapter 6. How the macro runtime processes a macro screen

This section describes the activities that occur when the macro runtime processes a macro screen. Although this topic is tedious to read about it is important, and sooner or later you are likely to have a question about it. You might want to read through the the first part of this chapter (the Overview) now and skip the remaining parts until you have a specific question.

## Overview

### Scenario used as an example

As an example, this chapter uses a scenario from a macro recorded in a previous chapter, Chapter 3, "Recording and playing back a simple macro", on page 13. This macro contains only two macro screens, Screen1 and Screen2.

The scenario begins at the point at which the macro runtime has performed all the actions in Screen1 and is ready to search for the next macro screen to be processed.

Screen1 is the macro screen that handles the ISPF Primary Option Menu (see Figure 5 on page 14). Table 9 shows a conceptual view of the contents of Screen1:

*Table 9. Contents of macro screen Screen1*

| **XML element contained in <screen> element Screen1:** | **Contents of XML element:** |
|---|---|
| <description> | Descriptors:<br>• The input inhibited indicator is cleared (input is not inhibited). |
| <actions> | Actions:<br>1. Move the text cursor to row 4 and column 16.<br>2. Type '3[enter]'. |
| <nextscreens> | Names of macro screens that can validly occur after this macro screen:<br>• Screen2 |

Screen2 is the macro screen that handles the Utility Selection Panel (see Figure 6 on page 15). Table 10 shows a conceptual view of the contents of Screen2:

*Table 10. Contents of macro screen Screen2*

| **XML element contained in <screen> element Screen2:** | **Contents of XML element:** |
|---|---|
| <description> | Descriptors:<br>• The input inhibited indicator is cleared (input is not inhibited).<br>• There are 80 fields.<br>• There are 3 input fields. |

© Copyright IBM Corp. 2003

*Table 10. Contents of macro screen Screen2 (continued)*

| XML element contained in <screen> element Screen2: | Contents of XML element: |
|---|---|
| <actions> | Actions (the host application pre-positions the text cursor in the correct input field):<br>1. Type '4[enter]'. |
| <nextscreens> | Names of macro screens that can validly occur after this macro screen:<br>• (None. This is the last macro screen in the macro.) |

## Stages in processing a macro screen

During macro playback the macro runtime loops through the same three stages of activity again and again until the macro terminates:

1. Determine the next macro screen to be processed.
2. Make the selected macro screen the new current macro screen.
3. Perform the actions in the new current macro screen's <actions> element.

*Figure 18. Stages in processing a macro screen*

## Closer look at stage 1

Stage 1 requires a more detailed explanation than stage 2 or 3. Stage 1 itself contains three steps:

- 1(a) Add the names of candidate macro screens to the list of valid next screens.
- 1(b) Do screen recognition to match one of the candidate macro screens to the actual application screen that is currently displayed in the session window.
- 1(c) Remove the names of candidate macro screens from the list of valid next screens.

*Figure 19. Three steps in stage 1*

Each of these steps involves the list of valid next screens.

The list of valid next screens is just a list that can hold macro screen names. The macro runtime creates this list at the beginning of macro playback (before playing back the first macro screen), and discards this list after macro playback is complete. Initially the list is empty (except possibly for transient screens, which are described later in this chapter).

During macro playback, each time the macro runtime needs to determine the next macro screen to be processed, it performs the three steps 1(a), 1(b), and 1(c) using the list of valid next screens.

## Overview of the entire process (all 3 stages)

In stage 1 the macro runtime determines the next macro screen to be processed. As stated in the previous section, stage 1 includes three steps.

In step 1(a) the macro runtime collects the names of macro screens that can occur after the current macro screen, and adds these names to the list of valid next

screens. There may be just one such screen on the list or several. In the example scenario, the macro runtime would look in the <nextscreens> element of Screen1, find one name (Screen2), and add that name to the list (see Table 9 on page 41).

In step 1(b), the macro runtime periodically checks each macro screen on the list to determine whether it matches the application screen.

There is a time factor here. Because of an action that the macro runtime has just performed in the current macro screen (in Screen1, typing '3[enter]' as the last action of the <actions> element), the host application is in the process of changing the session window so that it displays the new application screen (the Utility Selection Panel) instead of the old application screen (ISPF Primary Option Menu). However, this change does not occur immediately or all at once. The change takes some hundreds of milliseconds and may require several packets of data from the host.

Therefore, during step 1(b), every time the OIA line or the session window's presentation space is updated, the macro runtime again checks the macro screen (or screens) named in the list of valid next screens to determine whether one of them matches the application screen in its current state.

Eventually the session window is updated to the extent that the macro runtime is able to match one of the macro screens on the list to the application screen.

In step 1(c), the macro runtime removes all the macro screen names from the list of valid next screens (except transient screens if any).

In stage 2, the macro runtime makes the selected macro screen (the one that matched the application screen in step 1(b)) the new current macro screen.

Finally, in stage 3, the macro runtime performs the actions in the <actions> element of Screen2.

### Conclusion of the overview

At this point, if you are reading this book for the first time, you might want to skip the rest of this chapter and begin the next chapter. Later, if you have a question about how the macro runtime processes a macro screen, you can return to this chapter to learn more.

The rest of this chapter describes the same processing sequence as in the overview but provides more information about each step.

## Stage 1: Determining the next macro screen to be processed

As stated earlier, stage 1 contains three steps: adding macro screen names to the list of valid next screens, doing screen recognition, and removing the macro screen names from the list of valid next screens.

### Adding macro screen names to the list of valid next screens (step 1(a))

In this step the macro runtime places the names of candidate macro screens on the list of valid next screens.

## Valid next screens

When a host application has displayed an application screen in the session window, and a user input has occurred, then usually only a few application screens (frequently just one) can occur next.

In the example scenario, the current macro screen is Screen1, the current application screen is the ISPF Primary Option menu, and the input is '3' plus the enter key (see Table 9 on page 41). In this context, only one application screen can occur next, the Utility Selection Panel. Therefore the name of only one macro screen needs to be added to the list of valid next screens: Screen2.

But wait a minute, you say. The ISPF Primary Option Menu has about 30 different possible inputs (15 options, 6 menu selections, and 8 function keys). There should be 30 names of macro screens on the list, not just 1.

The reason that the list of valid next screens usually has only one or a few names on it is that the macro is executing a series of instructions that are aimed at accomplishing some specific task. In Screen1, the instructions are aimed at getting from the ISPF Primary Option Menu to the Utility Selection Panel. The necessary actions have been performed to make this transition occur ('3[enter]') and the macro screen is now just waiting for the expected application screen to appear.

## How the macro runtime selects the names of candidate macro screens

This section describes how the macro runtime selects the macro screen names that it places on the list of valid next screens. There are two cases:

- For the very first macro screen to be played back, the macro runtime selects the name of any macro screen in the macro that is marked as an entry screen.
- For all subsequent macro screens being played back, the macro runtime uses the names that it finds in the <nextscreens> element of the current macro screen.

**First macro screen:** When macro playback begins, the list of valid next screens is empty (except possibly for transient screens, see "Transient screens" on page 45).

To get candidates for the first macro screen to be processed, the macro runtime searches the entire macro, finds each macro screen that is marked as an entry screen, and adds the names of these macro screens to the list.

The entry screen setting (an attribute of the <screen> element) exists for exactly this purpose, to mark macro screens that can occur as the first screen to be processed.

When a macro is recorded, the Macro object by default marks just the first macro screen to be recorded as an entry screen. After recording is complete, the macro developer can mark (or unmark) any macro screen as an entry screen, and there can be multiple entry screens.

Entry screens are described in more detail in "Entry screens" on page 105.

If no macro screen is marked as an entry screen, then the macro runtime uses all the macro screens in the macro as candidates for the first macro screen to be processed.

**Subsequent macro screens:** For subsequent macro screens (including the one immediately after the first macro screen), the macro runtime finds the names of the candidate macro screens listed in the <nextscreens> element of the current macro screen.

In the example scenario, Screen1 is the current macro screen, and its <nextscreens> element contains the name of one macro screen, Screen2 (see Table 9 on page 41). Therefore the macro runtime adds Screen2 to the list.

However many macro screen names are listed in the <nextscreens> element, the macro runtime adds all of them to the list of valid next screens.

During macro recording, when the Macro object begins to record a new macro screen, it stores the name of that new macro screen (such as Screen2) in the <nextscreens> element of the macro screen that it has just finished recording (such as Screen1). Therefore each macro screen (except the last) of a recorded macro has the name of one macro screen stored in its <nextscreens> element.

Subsequently a macro developer can add or delete the name of any macro screen in the macro to or from the <nextscreens> element of any macro screen.

The <nextscreens> element is described in more detail in "Valid next screens" on page 103.

**Transient screens:** A transient screen is a screen that can occur at any point in the macro, that occurs unpredictably, and that always needs to be cleared. An example of a transient screen is an error screen that appears in response to invalid input.

The Macro object does not mark any macro screen as a transient screen during macro recording. However, subsequently the macro developer can mark any macro screen as a transient screen.

When macro playback begins, the macro runtime searches the macro, finds each macro screen that is marked as a transient screen, and adds the name of each transient macro screen to the list of valid next screens. These names remain on the list for the duration of the macro playback.

For more information on transient screens see "Transient screens" on page 106.

## Screen recognition (step 1(b))

In this step the macro runtime matches one of the macro screens named in the list of valid next screens to the current application screen.

This process is called screen recognition because the macro runtime recognizes one of the macro screens on the list as corresponding to the application screen that is currently displayed in the session window.

### Overview of evaluation

The macro runtime evaluates the candidate macro screens in the order in which their names appear in the list of valid next screens.

If the macro runtime finds that one of the candidates matches the application screen, then the macro runtime immediately stops evaluating and goes on to the next step of removing the candidate names from the list (step 1(c)). The matching screen becomes the next macro screen to be processed (stage 2).

However, if the macro runtime evaluates each macro screen named in the list without finding a match, then the macro runtime stops evaluating, temporarily, and does nothing further until the session window is updated.

## Re-doing the evaluation

While the macro runtime is working on screen recognition, the host application is working on updating the session window with the new application screen. In the example scenario, the host application is updating the session window so that it displays the Utility Selection Panel (see Table 9 on page 41 and Table 10 on page 41). This process takes some hundreds of milliseconds and may require several packets of data from the host.

This situation explains why the macro runtime temporarily stops working on screen recognition until the screen is updated. If screen recognition has failed, the reason may be that the new application screen is incomplete. Therefore the macro runtime waits.

Each time that the OIA line is updated or the presentation space of the session window is updated, the macro runtime again makes a pass through the list of valid next screens, trying to find a match to the current application screen. If no match occurs then the macro runtime waits again.

The macro runtime may go through several cycles of waiting and re-evaluating before screen recognition succeeds.

Eventually enough of the new application screen arrives so that the macro runtime can match one of the macro screens named in the list to the new application screen.

## Determining whether a macro screen matches the application screen

The macro runtime determines whether a macro screen matches the current application screen by comparing individual descriptors in the macro screen to the current session window.

In the example scenario, the macro runtime find the name Screen2 on the list of valid next screens, retrieves Screen2, looks at its descriptors, and compares the descriptors with the session window.

Each macro screen contains a <description> element that itself contains one or more descriptors. A descriptor is a statement of fact about the session window (application screen in its current state) that can be either true or false. In the example scenario, Screen2 contains three descriptors:

- The input inhibited indicator is cleared (input is not inhibited).
- There are 80 fields in the session window.
- There are 3 input fields in the session window.

When there are several descriptors in a <description> element, as here, the method that the macro runtime uses to evaluate the descriptors (as boolean true or false) and to combine their results into a single result (true or false) depends on some additional configuration information that is not described here.

However, in the example scenario, Screen2 is configured in the default manner, so that the macro runtime evaluates each of the three descriptors in turn. If all three are true, then the macro runtime concludes that the overall result is true, and that Screen2 matches the current application screen.

For more information about evaluating descriptors see “Evaluation of descriptors” on page 53.

### Two recognition features

**Timeout setting for screen recognition:** You can set a timeout value that causes the macro runtime to terminate the macro if screen recognition does not occur before the timer expires (see “Timeout settings for screen recognition” on page 107).

**Recognition limit:** You can set a recognition count that causes the macro runtime to terminate the macro or to jump to a specified macro screen if the macro runtime recognizes a macro screen, such as ScreenA, a number of times equal to the count (see “Recognition limit (General tab of the Screens tab)” on page 108).

## Removing the names of candidate macro screens from the list of valid next screens (step 1(c))

After screen recognition has succeeded, the macro runtime immediately begins its next task, which is cleaning up the list of valid next screens (step 1(c)).

This is a simple step. The macro runtime removes the names of all the candidate macro screens, whether recognized or not, from the list of valid next screens.

If the list contains the names of transient screens, those names remain on the list (see “Transient screens” on page 106).

# Stage 2: Making the successful candidate the new current macro screen

Stage 2 is simple. In stage 2 the macro runtime makes the successful candidate macro screen the new current macro screen.

In the example scenario, the macro runtime makes Screen2 the new current macro screen. The session window displays the new application screen, the Utility Selection Panel (see Table 9 on page 41 and Table 10 on page 41).

The macro runtime immediately begins stage 3.

# Stage 3: Performing the actions in the new current macro screen

In stage 3 the macro runtime performs the actions in the new current macro screen’s <actions> element. If the new current macro screen does not contain an <actions> element or if the <actions> element is empty, then the macro runtime skips this stage.

Each macro screen typically contains an <actions> element that contains one or more actions to be performed. An action is an instruction that causes some type of activity, such as sending a sequence of keys to the session window, displaying a prompt in a popup window for the user, capturing a block of text from the screen, or some other activity.

In the example scenario Screen2 contains only one action:

- Type ’4’ followed by the enter key.

Screen2 does not need an action to position the text cursor in the correct input field because the Utility Selection Panel automatically positions the text cursor there.

If the <actions> element contains multiple actions, the macro run time performs each macro action in turn in the order in which it occurs in the <actions> element.

For more information on actions see Chapter 8, “Macro actions”, on page 69.

### Inserting a delay after an action

Because the macro runtime executes actions much more quickly than a human user does, unforeseen problems can occur during macro playback that cause an action not to perform as expected, because of a dependency on a previous action.

To avoid this type of problem, the macro runtime by default inserts a delay of 300 milliseconds after every action in every macro screen (see “Pause Between Actions (Macro tab)” on page 111). Therefore, by default, after performing each action of any type, the macro runtime waits 300 milliseconds.

You should leave this feature enabled, although you can disable it if you want. You can change the delay from 300 milliseconds to some other value.

If you want to change the duration of the delay for a particular macro screen, you can do so (see “Set Pause Time (General tab of the Screens tab)” on page 111).

Also, for any particular action, you can increase the delay by adding a Pause action after the action (see “Pause action (<pause> element)” on page 86).

## Repeating the processing cycle

After the macro runtime has performed all the actions in the <actions> element of the current macro screen, the macro runtime immediately begins the processing cycle again, starting with step 1(a), and using the candidate macro screens listed in the <nextscreens> element of the new current macro screen.

## Terminating the macro

The macro runtime terminates the macro when it finishes processing a macro screen that is marked as an exit screen.

In the example scenario Screen2 is marked as an exit screen (see Table 10 on page 41).

The exit screen setting (an attribute of the <screen> element) exists for exactly this purpose, to mark macro screens that terminate the macro.

When a macro is recorded, the Macro object by default marks the last macro screen to be recorded as an exit screen. After recording is complete, the macro developer can mark (or unmark) any macro screen as an exit screen, and there can be multiple exit screens.

Exit screens are described in more detail in “Exit screens” on page 106.

# Chapter 7. Screen description and recognition

This chapter discusses:

- The terms descriptor, screen recognition, and screen description
- The Description tab
- How the Macro Facility records a description of an application screen
- How to combine multiple descriptors
- The various types of descriptors

## Terms defined

A descriptor is an XML element that occurs in the <description> element of a macro screen and that states an identifying characteristic of the application screen that the macro screen corresponds to.

For example, a macro screen named ScreenB might contain a String descriptor (<string> element) that states that row 3 of the application screen contains the string `ISPF Primary Option Menu`. During macro playback, when the macro runtime is determining which macro screen to process next, and when ScreenB is a candidate, the macro runtime compares the descriptor in ScreenB with the actual application screen. If the descriptor matches the actual application screen (row 3 of the application screen really does contain the string), then the macro runtime selects ScreenB as the next macro screen to be processed.

Screen recognition is the process that the macro runtime performs when it attempts to match a candidate macro screen to the current application screen.

As you may remember from Chapter 6, "How the macro runtime processes a macro screen", on page 41, when the macro runtime needs to determine the next macro screen to be processed, the macro runtime places the names of candidate macro screens (usually found in the <nextscreens> element of the current macro screen) onto a list of valid next screens. Then, as the host application updates the session window with the new application screen, the macro runtime compares the descriptors of each macro screen on the list with the new application screen. Eventually the application screen is updated to the extent (for example, the string `ISPF Primary Option Menu` appears in row 3) that the macro runtime can match one of the macro screens on the list to the application screen. The matched macro screen becomes the next macro screen to be processed (see "Overview of the entire process (all 3 stages)" on page 42).

Screen description is the process of adding descriptors to the <description> element of a macro screen. You engage in screen description when you go to the Description tab of a macro screen and create or edit a descriptor (such as the String descriptor in the previous example). Likewise, the Macro object during macro recording creates one or more descriptors for each new macro screen that it creates (see "Recorded descriptions" on page 52).

© Copyright IBM Corp. 2003

# Introduction to the Description tab

## Sample Description tab

The Description tab on the Screens tab of the Macro Editor gives you access to the information stored inside the <description> element of a macro screen. Figure 20 shows a sample Description tab:

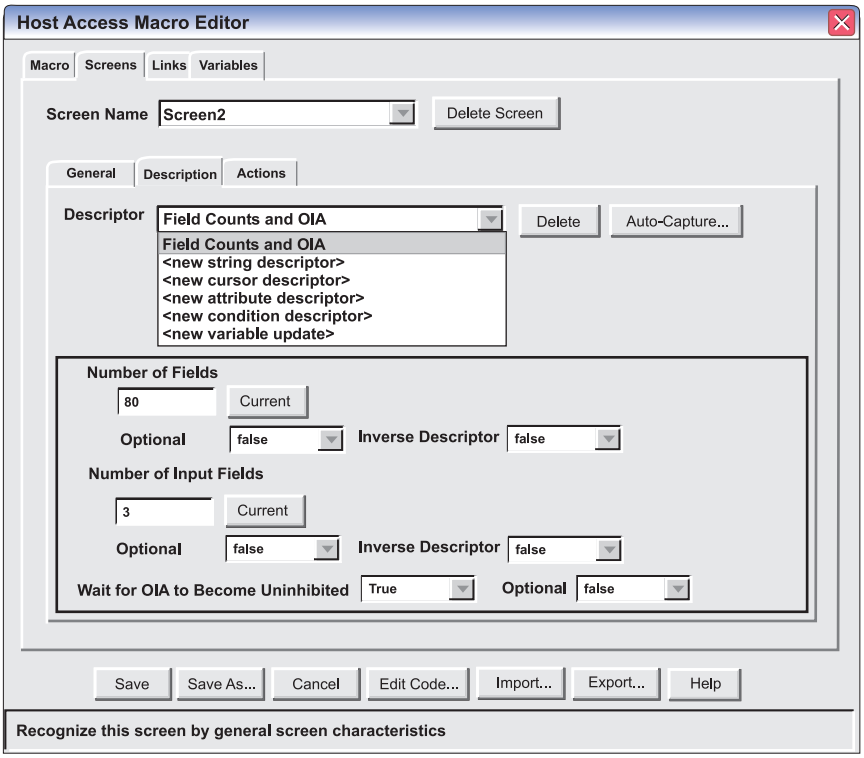


*Figure 20. Description tab*

In the figure above, the Screens tab of the Macro Editor is selected. The name of the currently selected screen, Screen2, is displayed in the Screen Name field at the top of the Screens tab. Below the Screen Name field are the General, Description, and Actions subtabs. The Description tab is selected.

As you look at Description tab in the figure above, you can see that it has an upper area and a lower area.

The upper area contains controls that operate on a single descriptor element considered as a whole. In particular, the Descriptor listbox situated in the upper left corner of the Description tab contains the name of the currently selected descriptor. In the figure above, the currently selected descriptor is a `Field Counts and OIA` descriptor at the top of the list. (Descriptors do not have names. `Field Counts and OIA` is the type of the descriptor.)

The lower area of the Description tab displays the contents of the currently selected descriptor. Because the currently selected descriptor is a Fields Counts and OIA descriptor, the lower area of the Description tab presents the contents appropriate to that type of descriptor. If the user created and selected another type of descriptor, such as a String descriptor, then the lower area would present the contents appropriate to a String descriptor.

Looking more closely at the lower area of the Description tab in Figure 20 on page 50, you can see that the Field Counts and OIA descriptor contains three tests of identity:

- The screen contains 80 fields (the `Number of Fields` field is set to 80).
- The screen contains 3 input fields (the `Number of Input Fields` field is set to 3).
- The screen has the input inhibited indicator cleared (the `Wait for OIA to Become Uninhibited` listbox is set to `true`).

The macro runtime will apply these three tests of identity when it tries to match this macro screen to an application screen.

Caution: Although the Macro Editor presents the Fields Counts and OIA descriptor as a single descriptor containing three tests, in fact the macro language defines these three tests as three separate and independent descriptors. See "Field Counts and OIA descriptor" on page 57.

The lower area of the Description tab in Figure 20 on page 50 also displays, for each of these three tests in the Field Counts and OIA descriptor, two more fields, labeled `Option` and `Inverse Descriptor`. You can ignore these two fields for now. They are described in the section "Default combining method" on page 54.

## Creating a new descriptor

Looking again at the Descriptor listbox in Figure 20 on page 50, you should notice that only the first entry is an actual descriptor. The remaining selections, which are all enclosed in angle brackets and all begin with the word `new`, are for creating new descriptors. Here is the list from Figure 20 on page 50:

```
Fields Counts and OIA
<new string descriptor>
<new cursor descriptor>
<new attribute descriptor>
<new condition descriptor>
<new variable update>
```

*Figure 21. Contents of the Descriptor listbox with one actual descriptor*

For example, if you clicked <new string descriptor>, the Macro object would create a new String descriptor and place it at the start of the list. The lower area of the Description tab would allow you to fill out the various fields that belong to a String descriptor (such as a row and column location and a character string). The Descriptor listbox would then look like this:

```
String descriptor(3, 29)
Fields Counts and OIA
<new string descriptor>
<new cursor descriptor>
<new attribute descriptor>
<new condition descriptor>
<new variable update>
```

*Figure 22. Contents of the Descriptor listbox with two actual descriptors*

In the figure above, the currently selected descriptor is now the `String descriptor` at the top of the list (the `3,29` stands for row 3, column 29). The `Field Counts and OIA` descriptor is now second on the list.

For information on how the macro runtime handles multiple descriptors, as in the figure above, see "Evaluation of descriptors" on page 53.

# Recorded descriptions

## What information is recorded

During macro recording the Macro object adds one or more descriptors to the new <description> element of each new macro screen that it creates.

For the first macro screen of the macro being recorded, the Macro object creates only one descriptor, a Field Counts and OIA descriptor with the following contents:

- The Number of Fields is set to blanks (meaning, ignore the number of fields).
- The Number of Input Fields is set to blanks (meaning, ignore the number of input fields).
- The Wait for OIA to Become Uninhibited field is set to true.

Therefore, when the recorded macro is played back (without having been revised in any way), the macro runtime matches the first macro screen to its corresponding application screen based entirely on whether the input inhibited indicator is cleared.

For every other application screen of the macro after the first application screen, the Macro object likewise creates only one descriptor, a Field Counts and OIA descriptor, but with different contents:

- The Number of Fields is set to the actual number of fields in the application screen (can be 0).
- The Number of Input Fields is set to the actual number of input fields in the application screen (can be 0).
- The Wait for OIA to Become Uninhibited field set to true.

Therefore, when the recorded macro is played back (without having been revised in any way), the macro runtime matches every macro screen after the first one to its corresponding application screen based on whether the input inhibited indicator is cleared, whether the count of fields in the macro screen's description matches the number of fields in the application screen, and whether the count of input fields in the macro screen's description matches the number of input fields in the application screen.

## Why the recorded descriptions work

The recorded descriptions work rather well for at least three reasons.

First, the three parts of the Field Counts and OIA descriptor can be applied unfailingly to every possible application screen. That is, every application screen has some number of fields (perhaps the number is 0), some number of input fields (perhaps 0), and an input inhibited indicator that is either set or cleared.

Second, the combination of a Number of Fields descriptor and a Number of Input Fields descriptor provides a pretty reliable description of an application screen, because application screens typically contain many fields. For example, the Utility Selection Panel shown in Figure 6 on page 15 currently contains 80 fields of all types, 3 of which are input fields. The ISPF Primary Option Menu shown in Figure 5 on page 14 currently contains 116 fields of all types, 3 of which are input fields. When application screens contain many fields, there is less chance of the

macro runtime confusing two application screens with one another because each contains the same number of fields and the same number of input fields.

Third, and perhaps most important, during screen recognition the macro runtime compares the new application screen to a short list (usually a very short list) of macro screens called valid next screens (see "Closer look at stage 1" on page 42). Therefore a single macro screen need not be differentiated from every other macro screen in the macro, only from the other screens in the list of valid next screens. Frequently the list consists of a single macro screen.

### Recorded descriptors provide a framework

Macro recording is a very useful feature because it quickly provides a framework for your macro. However, for some macro screens the recorded description might not be sufficient to allow the macro runtime to reliably distinguish one application screen from another similar application screen. In such cases you should improve the recorded description.

Often the most straightforward way to improve a recorded description is to add a String descriptor. For example, if the macro screen is for the Utility Selection Panel shown in Figure 6 on page 15, then you might add a String descriptor specifying that the application screen contains the string `'Utility Selection Panel'` somewhere in row 3. Of course you are not limited to using a String descriptor. Some situations might require that you use one or more of the other descriptors (such as a Cursor descriptor, Attribute descriptor, or Condition descriptor) to assure that the application screen is correctly recognized.

## Evaluation of descriptors

This section describes in detail how the macro runtime determines whether a macro screen matches an application screen.

### Practical information

Before you read through the following subsections, here are the you should be aware of the following facts:

- In most macro screens the <description> element contains more than one descriptor. (Remember that the Field Counts and OIA descriptor can include up to three independent descriptors. See "Field Counts and OIA descriptor" on page 57.)
- The default settings in the Description tab are that all descriptors are required (the Optional setting of each descriptor is false) and that the default combining rule is used.
- The most common scenario that you will encounter is that all descriptors are required. (That is, if you have defined three descriptors, you want all three of them to be true in order for the macro screen to be recognized.) If you are facing this scenario, then you should use the default settings.
- If you are faced with a scenario that is more complicated than the default scenario, then you should use the **uselogic** method.

### Overview of the process

Here is an overview of the process.

1. The macro runtime evaluates each descriptor individually and arrives at a boolean result for that descriptor, either true or false.

2. The macro runtime then combines the boolean results of the individual descriptors to determine whether the description as a whole is true (the macro screen matches the application screen) or false. To combine the results of the individual descriptors the macro runtime uses either the default combining method or the **uselogic** method.
   a. With the default combining method:
      1) The macro runtime inverts the boolean result of any descriptor that has the Inverse Descriptor option set to true.
      2) The macro runtime combines the boolean results of the individual descriptors using:
         - The setting of the Optional option for each descriptor.
         - The default rule for combining descriptors.
   b. In contrast, with the **uselogic** method:
      1) The macro runtime ignores the settings for Inverse Descriptor and Optional.
      2) The macro runtime combines the results of individual descriptors using a rule that you provide in the **uselogic** attribute.

## Evaluation of individual descriptors

For each individual descriptor in the macro description, the macro runtime evaluates the descriptor and arrives at a boolean result of true or false.

For example, if the descriptor is a String descriptor, then the macro runtime looks in the application screen at the row and column that the descriptor specifies, and compares the string at that location with the string that the descriptor specifies. If the two strings match, then the macro runtime assigns a value of true to the String descriptor. If the two strings do not match then the macro assigns a value of false to the String descriptor.

Usually a macro screen contains more than one descriptor.

However, if a macro screen contains only one descriptor (and assuming that the descriptor does not have the Inverse Descriptor option set to true) then if the single descriptor is true the entire description is true, and the macro runtime recognizes the macro screen as a match for the application screen. In contrast, if the single descriptor is false, then the entire description is false, and the macro screen is not recognized.

## Default combining method

If you have more than one descriptor in a <description> element, then you must use either the default combining method described in this section or the **uselogic** attribute described in "The uselogic attribute" on page 56.

### When to use the default combining method

The default combining method is appropriate for only two scenarios:

- You want the description as a whole to be true only if ALL the individual descriptors are true (this is the most common scenario); or
- You want the description as a whole to be true if AT LEAST ONE of the individual descriptors is true.

You should not use the default method for any other scenario, unless you thoroughly understand how the default combining method works.

The default combining method uses:

- The boolean result for each individual descriptor (see "Evaluation of individual descriptors" on page 54).
- The value of the Inverse Descriptor option in each individual descriptor.
- The value of the Optional option in each individual descriptor.
- The default combining rule.

### Inverse Descriptor

Every descriptor has an Inverse Descriptor option that is set to false (the default) or true. You can see the Inverse Descriptor option as a listbox below and to the right of the Number of Fields input field in Figure 20 on page 50. The macro language uses the **invertmatch** attribute of the descriptor element to store this option.

By default this option is false, so that it has no effect on the evaluation of the descriptor.

If this setting is true, then the macro runtime inverts the boolean result that it obtains from evaluating the descriptor, changing true to false or false to true.

For example, if the macro runtime determines that a String descriptor is true (the string in the descriptor matches the screen in the application window), but the String descriptor has the Inverse Descriptor option set to true, then the macro runtime changes the String descriptor's result from true to false.

### Optional

Every descriptor has an Optional option that is set to either false (the default) or true. You can see this option as a listbox below the Number of Fields input field in Figure 20 on page 50. The macro language uses the **optional** attribute of the descriptor element to store this option

The Optional option states how a individual descriptor's result is to be treated when the macro runtime uses the default combining rule to combine the boolean results of the descriptors. By default this option is set to false, signifying that the descriptor's result is required rather than optional.

### Default combining rule

As stated earlier, the default combining rule is appropriate for only two scenarios:

- You want the description as a whole to be true only if ALL the individual descriptors are true (this is the most common scenario); or
- You want the description as a whole to be true if AT LEAST ONE of the individual descriptors is true.

If you want the description as a whole to be true only if ALL the descriptors are true, then set the Optional setting of all the descriptors in the description to false (the default setting).

In contrast, if you want the description as a whole to be true if AT LEAST ONE of the descriptors is true, then set the Optional setting of all of the descriptors in the description to true.

You should not use the default combining rule in any other scenario where you have multiple descriptors in one macro screen, unless you understand the rule and its implications thoroughly. For more information see "The default combining rule for multiple descriptors in one macro screen" on page 177.

Also, you should not set the Optional settings of multiple descriptors in one macro screen differently (some true, some false) unless you understand the rule and its implications thoroughly.

## The uselogic attribute

The **uselogic** attribute of the <description> element allows you to define more complex logical relations among multiple descriptors than are available with the default combining method described in the previous section.

If you use the **uselogic** attribute, then the macro runtime ignores the Inverse Descriptor settings and the Optional settings in the individual descriptors.

You have to add the **uselogic** attribute to the <description> element manually using the the Code Editor. The Macro Editor does not provide a control for this.

The value of the **uselogic** attribute is a simplified logical expression whose terms are 1-based indexes of the descriptors that follow. Figure 23 shows an example of a <description> element that contains a **uselogic** attribute (some of the attributes of the <string> element are omitted for clarity):

```
<description uselogic="(1 and 2) or (!1 and 3)" />
   <oia status="NOTINHIBITED" optional="false" invertmatch="false"/>
   <string value="&apos;Foreground&apos; row="5" col="8"/>
   <cursor row="18" col="19" optional="false" invertmatch="false"/>
</description>
```

*Figure 23. Example of the uselogic attribute of the <description> element*

In the figure above the value of the **uselogic** attribute is:

```
(1 and 2) or (!1 and 3)
```

This logical expression is not a regular logical expression (as described in "Conditional and logical operators and expressions" on page 36) but rather a simplified style of logical expression used only in the **uselogic** attribute. The rules for this style of logical expression are:

- The numerals 1, 2, 3, and so on stand for the boolean results of, respectively, the first, second, and third descriptors in the <description> element (<oia>, <string>, and <cursor> in the figure above). You can use any numeral for which a corresponding descriptor exists. For example, if a <description> element has seven descriptors, then you can use 7 to refer to the boolean result of the seventh descriptor, 6 to refer to the boolean result of the sixth descriptor, and so on.
- Only the following logical operators are allowed:

*Table 11. Logical operators for the uselogic attribute*

| **Operator:** | **Meaning:** |
|---|---|
| and | Logical AND |
| or | Logical OR (inclusive) |
| ! | Logical NOT (inversion) |

- You can use parentheses () to group terms.
- The following entities are not allowed:
  - Arithmetic operators and expressions

– Conditional operators and expressions
– Variables
– Calls to Java methods

In the example in Figure 23 on page 56 the macro runtime will determine that the description as a whole is true if:

- The result of the first descriptor is true and the result of the second descriptor is true (1 and 2); or
- The result of the first descriptor is false and the result of the third descriptor is true (!1 and 3).

Remember that if you use the **uselogic** attribute, then the macro runtime ignores the Inverse Descriptor settings and the Optional settings in the individual descriptors.

# The descriptors

## Overview

Each type of descriptor is stored as an individual XML element situated within the <description> element of one macro screen.

You do not have to understand all the types of descriptors at first. Instead you should begin by becoming familiar with just two types:

- The Fields Counts and OIA descriptor (actually contains 3 individual descriptors)
- The String descriptor

These types of descriptors are sufficient to reliably describe many and perhaps even most application screens. However, if these types are not sufficient, then you should turn for help to one of the other types of descriptors.

Table 12 lists all the types of descriptors and shows the number of descriptors of each type that are allowed to exist in one macro screen (more specifically, in one <description> element belonging to one <screen> element):

*Table 12. Types of descriptors, how many of each type allowed*

| **Type of descriptor:** | **Number of this type of descriptor allowed per macro screen:** |
|---|---|
| Field Counts and OIA | 1 (required) |
| String descriptor | 0 or more |
| Cursor descriptor | 0 or 1 |
| Attribute descriptor | 0 or more |
| Condition descriptor | 0 or more |

The following subsections describe each type of descriptor in detail.

## Field Counts and OIA descriptor

### Required

The Field Counts and OIA descriptor is required and must be unique. That is, every Description tab (<description> element) must contain one and only one Field Counts and OIA descriptor.

This fact should not cause you any trouble in practice, for the following reasons:

- Although the Field Counts and OIA descriptor itself is required, only one of the three tests that it contains is required. Therefore the actual requirement is that every Description tab must contain one and only one Wait for OIA to Become Uninhibited descriptor.
- The Macro Editor and the Code Editor enforce these rules and will not let you mistakenly include more than one Field Counts and OIA descriptor in a Description tab (or <description> element). For example,
  - The Delete button on the Description tab does not have any effect when you try to delete the Field Counts and OIA descriptor.
  - The Descriptor listbox on the Description tab does not contain a <new> entry for the Field Counts and OIA descriptor.

### Presents three separate and independent descriptors as if one

For user friendliness and for certain design reasons that are not discussed here, the Macro Editor presents the Field Counts and OIA descriptor as one descriptor (see Figure 20 on page 50). However, in fact each of the three parts of the Field Counts and OIA descriptor on the Description tab of the Macro Editor corresponds to a separate and independent descriptor in the underlying XML macro language. Specifically:

- The Number of Fields setting is stored as a <numfields> descriptor.
- The Number of Input Fields setting is stored as a <numinputfields> descriptor.
- The Wait for OIA to Become Uninhibited setting is stored as an <oia> descriptor.

Table 13 lists these three types of descriptors and shows how many of each can occur within a <description> element:

*Table 13.*

| **Type of descriptor:** | **Number of this type of descriptor allowed per macro screen (that is, per <description> element):** |
|---|---|
| <oia> | 1 (required) |
| <numfields> | 1 (optional) |
| <numinputfields> | 1 (optional) |

As the table above shows, only one of each type of these descriptors can occur in a <description> element. The <oia> descriptor is required, but the <numfields> descriptor and the <numinputfields> descriptor are optional. The Macro Editor enforces these rules.

You can reinforce these ideas in your mind by looking at a Field Counts and OIA descriptor first as it appears on the Description tab of the Macro Editor and then in the Code Editor. Figure 20 on page 50 shows a Field Counts and OIA descriptor on the Description tab of the Macro Editor. The settings of the three parts of the Field Counts and OIA descriptor are set as follows:

```
Number of Fields:  80
Number of Input fields:  3
Wait for OIA to Become Uninhibited:  true
```

But if you look at the corresponding <description> element with the Code Editor, you see the following:

```
<description>
   <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
   <numfields number="80" optional="false" invertmatch="false" />
   <numinputfields number="3" optional="false" invertmatch="false" />
</description>
```

*Figure 24. A <description> element with three descriptors*

The XML code fragment above shows that the <description> element contains three separate and independent descriptors, each corresponding to one of the three parts of the Field Counts and OIA descriptor.

Now suppose if you will that you change the Field Counts and OIA descriptor settings to be as follows:

```
Number of Fields:  (blank)
Number of Input fields:  (blank)
Wait for OIA to Become Uninhibited:  true
```

Setting the first two fields to blank tells the Macro Editor that these items are not to be included in the script. If you look again at the corresponding <description> element with Code Editor you now see:

```
<description>
   <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
</description>
```

The XML code fragment above shows that the <description> element now contains only one descriptor, an <oia> descriptor corresponding to the Wait for OIA to Become Uninhibited setting in the Field Counts and OIA descriptor.

## Treatment during screen recognition

During screen recognition, when the macro runtime evaluates individual descriptors and combines the boolean results, the macro runtime treats the <oia> descriptor, the <numfields> descriptor (if it is present), and the <numinputfields> descriptor (if it is present) each as a separate and independent descriptor, one like any other descriptor.

For more information about evaluating multiple descriptors see “Evaluation of descriptors” on page 53

## Wait for OIA to Become Uninhibited descriptor (<oia> element)

Table 14 on page 60 shows:

- The three permissible settings for the Wait for OIA to Become Uninhibited listbox.
- The corresponding values used in the <oia> element.
- How the macro runtime evaluates the setting.

*Table 14. Valid settings for the descriptor Wait for OIA to Become Uninhibited*

| Setting on the Description tab: | Value of the `status` attribute in the <oia> element: | Meaning: |
|---|---|---|
| `true` | NOTINHIBITED | If the input inhibited indicator in the session window is cleared (that is, input is not inhibited) then the macro runtime evaluates the descriptor as true. Otherwise the macro runtime evaluates the descriptor as false. |
| `false` | DONTCARE | The macro runtime always evaluates the descriptor as true. |
| `<Expression>` | 'NOTINHIBITED', 'DONTCARE', or any expression that evaluates to one of these strings. | The macro runtime evaluates the expression and then interprets the resulting string. |

In almost all scenarios you can accept the default setting for this descriptor, which is true (on the Description tab) and NOTINHIBITED (in the macro language). Then, during screen recognition:

- If the input inhibited indicator in the session window is set (that is, input is inhibited), then the macro runtime will evaluate this descriptor as false.
- But if the input inhibited indicator is cleared (that is, input is not inhibited), then the macro runtime will evaluate this descriptor as true.

These are the results that you would want and expect. You typically do not want the macro runtime to recognize the macro screen and immediately start processing its actions while the input inhibited indicator is still set. An important timing issue is involved here, which you should read about separately (see "Screen completion" on page 112). But no matter how you resolve that issue, you should almost always leave this descriptor at the default setting, which is true.

However, if you have a scenario in which you want the macro runtime to ignore the input inhibited condition, then set this descriptor to false on the Description tab (the equivalent setting in the macro language is DONTCARE).

## Number of Fields descriptor (<numfields> element)

The Number of Fields descriptor specifies a particular number of 3270 (or 5250) fields. You can use an integer in the Number of Fields input field, or any entity that evaluates to an integer (such as a variable, an arithmetic expression, or a call to an external Java method).

During screen recognition the macro runtime:

1. Evaluates this descriptor and obtains an integer result.
2. Counts the number of fields in the application screen (in its current state).
3. Compares the two numbers.

If the two numbers are equal then the macro runtime evaluates this descriptor as true. Otherwise the macro runtime evaluates this descriptor as false.

When the macro runtime counts the fields in the session window it counts all types of 3270 (or 5250) fields, including input fields.

If you do not want to use this descriptor then set the Number of Fields input field to blank.

### Number of Input Fields descriptor (<numinputfields> element)

The Number of Input Fields descriptor is very similar to the Number of Fields descriptor described in the previous section. The difference is that the Number of Input Fields descriptor specifies a number of 3270 (or 5250) input fields, whereas the Number of Fields descriptor specifies a number of fields of all types, including input fields.

You can use an integer in the Number of Input Fields field, or any entity that evaluates to an integer (such as a variable, an arithmetic expression, or a call to an external Java method).

During screen recognition the macro runtime:

1. Evaluates this descriptor and obtains an integer result.
2. Counts the number of input fields in the application screen (in its current state).
3. Compares the two numbers.

If the two numbers are equal then the macro runtime evaluates this descriptor as true. Otherwise the macro runtime evaluates this descriptor as false.

If you do not want to use this descriptor then set the Number of Input Fields field to blank.

### Counting fields in the session window during macro development

When you are editing a Field Counts and OIA descriptor and you want to set the Number of Fields field and the Number of Input Fields field to the correct values, you can use the Current button beside each input field to count the fields in the application screen for you.

To use this feature follow these steps:

1. Click on the session window to activate it (see "Using the session window" on page 129).
2. In the session window, go to the application screen corresponding to the macro screen that you are working on.
3. In the Description tab select the Field Counts and OIA descriptor.
4. On the Description tab click the Current button immediately to the right of the Number of Fields input field. The Macro Editor counts the number of fields in the current application screen and then displays the count in the input field.
5. On the Description tab click the Current button immediately to the right of the Number of Input Fields input field. The Macro Editor counts the number of input fields in the current application screen and then displays the count in the input field.

## String descriptor (<string> element)

The String descriptor specifies the following information:

- A sequence of characters (the string).
- A rectangular area of text on the session window.

The macro runtime searches inside the entire rectangular area of text for the string you specify. If the macro runtime finds the string inside the rectangular area of text, then it evaluates the string descriptor as true. If not, then it evaluates the string descriptor as false.

## Specifying the rectangular area

You define the rectangular area of text by specifying the row and column coordinates of opposite corners. The default values for these coordinates are (1,1) and (-1,-1), indicating the entire text area of the session window (for the significance of negative values such as -1,-1, see "Significance of a negative value for a row or column" on page 39). You can use an integer or any entity that evaluates to an integer (such as a variable, an arithmetic expression, or a call to an external Java method).

The rectangular area can be just large enough to contain the string, or much larger than the string. For example, suppose that the application screen that you want to match to the macro screen has the string `'Terminal and user parameters'` in the rectangular area (6,20), (6,37). This rectangular area is exactly large enough to contain the string. If the application screen always has this string at this location, then you might specify the exact rectangular area.

However, suppose that the application screen that you want to match to the macro screen has the same string, `'Terminal and user parameters'`, located somewhere on it, but that you cannot predict which row of the application screen will contain the string. In this case you could specify the rectangular area (1,1), (-1,-1), indicating that the macro runtime should search every row of the application screen for the identifying string.

For the string value you can use a string or any entity that evaluates to a string (such as a variable, an expression, or a call to an external Java method). The string must be in the form required by the macro format that you have chosen, either basic or advanced (see "Error in specifying a string" on page 130).

During screen recognition the macro runtime:

1. Evaluates the row and column values and obtains an integer result for each value.
2. Evaluates the string value and obtains a string result.
3. Looks for the string anywhere within the rectangular block of text in the application screen (in its current state) specified by the row and column values.

If the the macro runtime finds the string within the rectangular block of text then the macro runtime evaluates this descriptor as true. Otherwise the macro runtime evaluates this descriptor as false.

## How the macro runtime searches the rectangular area (Wrap option)

If the Wrap option is set to false (the default setting), then the macro runtime searches each row of the rectangular area separately. This method works well when the entire string is contained within one row. For example, if the string is `Utility Selection Panel` and the rectangular area is (1,1), (24,80), then the macro runtime searches for the string as follows:

1. Get the first row of the rectangular area. Determine whether the string occurs in the this row. If it does not, then search the next row.
2. Get the second row of the rectangular area. Determine whether the string occurs in this row. If it does not, then search the next row.

3. And so on.

In contrast, if the Wrap option is set to true then the macro runtime searches for the string as follows:

1. Get all the lines of the rectangular area and concatenate them all in order.
2. Determine whether the string occurs in the concatenated string.

If the string you are searching for can wrap from one line to the next of the session window, then you should set the Wrap option to true. Do not confuse this option with the Unwrap attribute of the Extract action, which is based on fields rather than blocks of text (see "Unwrap Text option" on page 80).

The following description provides an example in which the Wrap option is set to true.

Figure 25 shows rows 14 through 18 of an application screen:

```
 6  Hardcopy    Initiate hardcopy output
 7  Transfer    Download ISPF Client/Server or Transfer data set
 8  Outlist     Display, delete, or print held job output
 9  Commands    Create/change an application command table
 *  Reserved    This option reserved for future expansion
```

*Figure 25. Rows 14–18 of an application screen*

In the rows above, the first character of each row is a blank space. For example, in row 14, the the first two characters are `' 6'`, that is, a blank space followed by the numeral 6. Suppose that you want to set up a String descriptor that checks for the following rectangular block of text on this application screen:

```
Hardcopy
Transfer
Outlist
Commands
Reserved
```

The steps in setting up the String descriptor for this multi-row block are as follows:

1. Create a new String descriptor.
2. Set the values in the Row and Column fields. The row and column location of the upper left corner of the text rectangle above is (14, 5) and the row and column location of the lower right corner is (18, 12).
3. Set the string value. The string value is:

   ```
   'HardcopyTransferOutlist CommandReserved'
   ```
4. Set the Wrap option to true.
5. Leave all the other options set to the default.

Notice that in step 3 above the five rows are concatenated as a single string, without any filler characters added (such as a newline or space at the end). However, the string does contain a blank space after `'Outlist'` because that blank space does fall within the boundaries of the rectangle.

**Using an extracted string in a String descriptor:** If you use an Extract action to read text from the screen into a string variable (see "Extract action (<extract> element)" on page 77) then in a subsequent screen you can use the string variable in the String input field of a String descriptor.

For example, in ScreenA you might read a company name from the session window into a string variable named $strTmp$, using an Extract action. Then in ScreenB you could use $strTmp$ as the string to be searched for in a String descriptor.

You can do this when extracting multiple lines of text if you have the Wrap option set to true.

### The '*' string in a new String descriptor

When you create a new String descriptor the Macro Editor places the string `'*'` into the String input field as an initial, default value. Just erase this initial string and fill in the string that you want. The asterisk (*) does not mean anything or have any function. The initial string could say 'Default string value' and have the same effect.

### Easy method for filling in the parameters

When you are editing a String descriptor and you want to specify the correct text rectangle and text string, you can use the marking rectangle to set these values. To use this feature follow these steps:

1. In the Descriptions tab, select the String descriptor that you want to edit.
2. Click on the session window to activate it (see "Using the session window" on page 129).
3. In the session window, go to the application screen corresponding to the macro screen that you are working on.
4. In the session window, use the marking rectangle to mark the block of text that you want to use in the String descriptor (see "Using the marking rectangle" on page 129).
   - When you complete the marking rectangle in the session window, the Macro Editor automatically copies the row and column values for the corners of the rectangle into the Start Row, Start Column, End Row, and End Column input fields in the String descriptor window on the Description tab.
5. In the session window, click Edit > Copy.
6. In the String descriptor window, click the String field. Move the text cursor in the String field to the position at which you want to add the string.
7. Click Ctrl-v.
8. If you captured a rectangle containing more than one row of text from the session window, then the string requires further editing. Read the description below of how to edit the string.

When you click Ctrl-v the Macro Editor does the following:

- Copies the text from the rectangular block in the session window into the String field.

However, if you captured more than one row of text (as in the example in "How the macro runtime searches the rectangular area (Wrap option)" on page 62), then the paste operation inserts an extra space after each row of text except the last. Edit the String field to remove these extra spaces. In the example in "How the macro runtime searches the rectangular area (Wrap option)" on page 62, the String field contains the following string before editing:

```
'Hardcopy Transfer Outlist  Command Reserved'
```

Notice that the string is too long (by 4 characters) to fit in the rectangle of the specified size (5 rows, 8 columns), and that an extra blank space has been inserted after each row. You should edit the string to the following value:

```
'HardcopyTransferOutlist CommandReserved'
```

### Multiple String descriptors in the same <description> element

The Macro Editor does not prevent you from creating more than one String descriptor in the same <description> element. You should use as many String descriptors as you need in order to create a reliable description.

You can even define two different strings for the same rectangular block of text in the same <description> element. You might do this if the application screen corresponding to your macro screen displays different strings in the same location at different times. However, if you do define two different strings for the same rectangular block of text, you should be careful to indicate that both of the strings are not required (both are optional).

## Cursor descriptor (<cursor> element)

The Cursor descriptor specifies a row and column location on the application screen, such as row 10 and column 50. For either the row value or the column value you can use an integer or any entity that evaluates to an integer (such as a variable, an arithmetic expression, or a call to an external Java method).

During screen recognition the macro runtime:

1. Evaluates the row value and obtains an integer result.
2. Evaluates the column value and obtains an integer result.
3. Looks at the row and column location of the text cursor in the application screen (in its current state).
4. Compares the row and column location in the descriptor with the row and column location of the text cursor in the application screen.

If the two locations are the same then the macro runtime evaluates this descriptor as true. Otherwise the macro runtime evaluates this descriptor as false.

When you are editing a Cursor descriptor and you want to specify the correct row and column values, you can use the Current button. The Current button is situated immediately to the right of the Row and Column fields in the Cursor descriptor window.

To use this feature follow these steps:

1. Click on the session window to activate it (see “Using the session window” on page 129).
2. In the session window, go to the application screen corresponding to the macro screen that you are working on.
3. In the session window, use the mouse or the keyboard to set the text cursor to the location that you want.
4. In the Cursor descriptor window click Current.

When you click Current the Macro Editor does the following:

- Finds the location of the text cursor in the current session window and sets the Row and Column values to that location.

## Attribute descriptor (<attrib> element)

The attribute descriptor specifies a 3270 or 5250 attribute and a row and column location on the application window.

During screen recognition the macro runtime compares the specified attribute (such as 0x3) to the actual attribute at the row and column specified. If the attributes are the same, then the macro runtime evaluates the descriptor as true. Otherwise the macro runtime evaluates the descriptor as false.

This descriptor can be useful when you are trying to differentiate between two application screens that are very similar except for their attributes.

### Specifying an attribute

Before you specify an attribute, use the Data Plane listbox to select the data plane in which the attribute that you are looking for occurs. If you select `<Expression>` then you must specify an expression (such as a variable named $strDataPlane$) that resolves at runtime to one of the data plane strings in the listbox (`FIELD_PLANE`, `COLOR_PLANE`, or `EXFIELD_PLANE`).

Use the Row and Column input fields to specify the Row and Column location on the application screen of the attribute that you are looking for.

You can use any one of three methods to specify an attribute value:

- Click the session window and move the text cursor to the row and column location of an attribute like the one that you want specify in the Attribute descriptor, then click Current in the Macro Editor; or
- Click Edit Attributes and use the controls on the popup window; or
- Type the value into the Attribute Value input field (for example, `0x3`).

## Condition descriptor (<condition>) element

The Condition descriptor specifies a conditional expression that the macro runtime evaluates during screen recognition, such as `$intNumVisits$ == 0`. For more information on conditional expressions see "Conditional and logical operators and expressions" on page 36.

During screen recognition the macro runtime:

- Evaluates the conditional expression and obtains a boolean result.

If the conditional expression evaluates to true then the macro runtime evaluates this descriptor as true. Otherwise the macro runtime evaluates this descriptor as false.

The Condition descriptor increases the flexibility and power of screen recognition by allowing the macro runtime to determine the next macro screen to be processed based on the value of one or more variables or on the result of a call to a Java method.

## Custom descriptor (<customreco> element)

The Custom descriptor allows you to call custom description code.

To use a Custom descriptor you need the separate Host Access Toolkit product.

To create a Custom descriptor you must use the Code Editor to add an <customreco> element to the <description> element of the macro screen. For more information on this element see "<customreco> element" on page 150.

## Variable update action (<varupdate> element)

The final type of entry that can occur in the Descriptor listbox is the Variable update entry, which is not a descriptor at all, but rather an action that the macro language allows to occur inside a <description> element.

The Variable update action in a <description> element performs the very same type of operation that it performs in an <actions> element, which is to store a specified value into a specified variable.

For information about creating a Variable update action see "Variable update action (<varupdate> element)" on page 97.

### Processing a Variable update action in a description

You should be aware of how the macro runtime processes one or more Variable update actions when they occur in a <description> element:

1. The macro runtime performs all the Variable update actions immediately, as if they were first in sequence.
2. The macro runtime then evaluates the remaining items (descriptors) in the description as usual and arrives at an overall boolean result. The Variable update actions have no effect on the boolean result.

As you might remember, the macro runtime can go through the screen recognition process a number of times before matching a macro screen to an application screen (see "Re-doing the evaluation" on page 46). Therefore, if a <description> element contains one or more Variable update actions, then the macro runtime will perform the Variable update actions each time that it evaluates the <description> element.

For example, suppose that a macro is being played back, that the screen name ScreenB is on the list of valid next screens, and that ScreenB contains a <description> element like the one shown in Figure 26:

```
<description>
   <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
   <varupdate name="$boolUpdate$" value="true" />
   <attrib value="0x4" row="1" col="1" plane="COLOR_PLANE" optional="false"
              invertmatch="false" />
</description>
```

*Figure 26. The <description> element of ScreenB*

Each time that the macro runtime tries to match ScreenB to the current application screen:

1. The macro runtime sees the <varupdate> action and performs it, storing the value `true` into `$boolUpdate$`.
2. The macro runtime evaluates the <oia> descriptor and the <attrib> descriptor and arrives at a boolean result for the entire <description> element.

### Variable update with the uselogic attribute

If you want the macro runtime to perform a Variable update action in a <description> element in some other sequence than first, you can change the order of execution by using the <description> element's **uselogic** attribute instead of the default combining rule (see "The uselogic attribute" on page 56).

When a Variable update action is used in a **uselogic** attribute:

- The macro runtime performs the Variable update action in the same order in which it occurs in the **uselogic** attribute.
- The macro runtime always evaluates the Variable update action to true.

# Chapter 8. Macro actions

## In general

### The actions by function

Here is a list of all the actions, grouped according to function.

- Interaction with the user:
  - Box selection
  - Input (keystrokes and key-activated functions, such as copy to clipboard)
  - Message
  - Mouse click
  - Prompt
- Getting data from the application
  - Extract
  - Transfer (file transfers between host and client, either upload or download)
  - Variable update
- Waits
  - Comm wait
  - Pause
- Programming
  - Conditional
  - Perform (call a Java method)
  - Play macro (chain to another macro)
  - Run program (launch a program on the client operating system)
  - Variable update
- System
  - Print extract
  - Print start
  - Print end
  - Run program (launch a program on the client operating system)
- Debug
  - Trace

### How actions are performed

#### The runtime context

As you may remember from Chapter 6, "How the macro runtime processes a macro screen", on page 41, when the macro runtime has selected a new current macro screen, the macro runtime immediately begins to perform the actions in the <actions> element of that macro screen.

After the macro runtime has performed all the actions, it immediately goes on to the next step of determining the next macro screen to be processed.

© Copyright IBM Corp. 2003

### The macro screen context

Within a single macro screen, the macro runtime performs the actions in the order in which they occur in the <actions> element. This is the same order in which you haved ordered the actions in the Action listbox.

You are not required to create any actions for a macro screen. If there is no <actions> element or if the <actions> element is empty, then the macro runtime will go straight to the next section of macro screen processing, which is selecting the next macro screen to be processed.

## Specifying parameters for actions

In specifying the parameters of an action, remember that, in general, any context that accepts an immediate value of a particular standard data type also accepts any entity of the same data type. For example, if an input field accepts a string value, then it also accepts an expression that evaluates to a string, a value that converts to a string, a string variable, or a call to an imported method that returns a string (see "Equivalents" on page 38).

However, there are a few fields in which you cannot use variables (see "Using the value that the variable holds" on page 124).

# Introduction to the Actions tab

## Sample Actions tab

The Actions tab on the Screens tab of the Macro Editor allows you to create and edit actions. When you create an action in the Actions tab, the Macro Editor inserts the new action into the <actions> element of the currently selected screen. Figure 27 on page 71 shows a sample Actions tab:

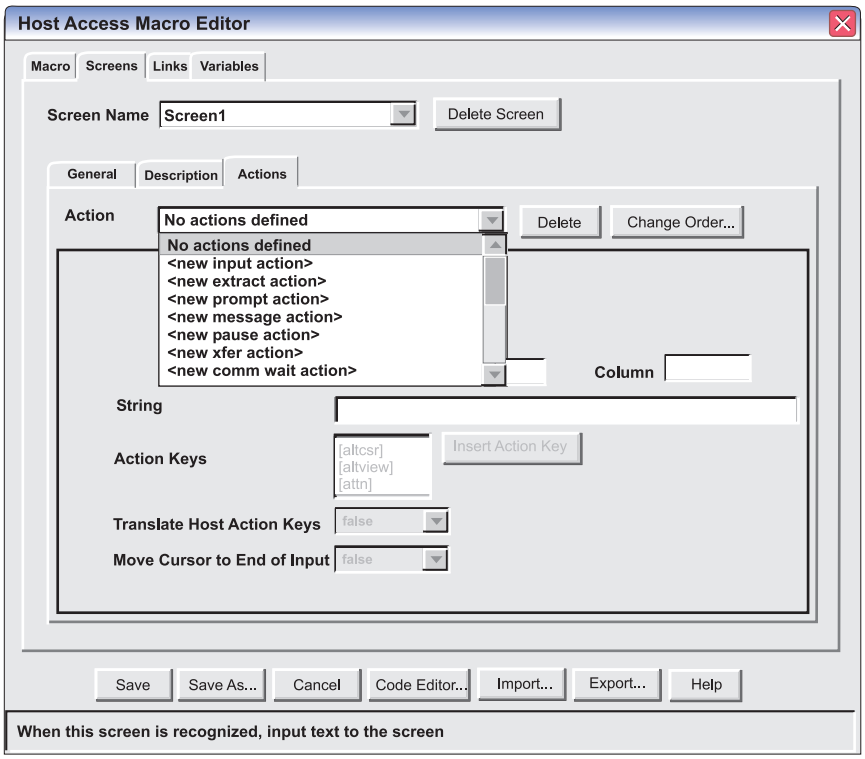


*Figure 27. Actions tab*

In the figure above, the Screens tab of the Macro Editor is selected. The name of the currently selected screen, Screen1, is displayed in the Screen Name field at the top of the Screens tab. Below the Screen Name field are the General, Description, and Actions subtabs. The Actions tab is selected.

As you look at the Actions tab in the figure above, you can see that, like the Description tab, it has an upper area and a lower area.

The upper area contains controls that operate on a single action element considered as a whole. In particular, the Action listbox situated in the upper left corner of the Actions tab contains the name of the currently selected action. In the figure above, there is no currently selected action, because no action has been created yet.

The lower area of the Actions tab displays the contents of the currently selected action, if any. If the currently selected descriptor is an Input action, then the lower area of the Actions tab presents the contents appropriate to that type of action. If the user creates or selects another type of action, such as an Extract action, then the lower area presents the contents appropriate to an Extract action.

## Creating a new action

Looking again at the Actions listbox in Figure 27, you should notice that it does not yet contain any actions. The selections, which are all enclosed in angle brackets and all begin with the word `new`, are for creating new actions. As you can see in Figure 27, the displayable part of the Actions listbox is not tall enough to show the whole list at once. Here is the entire list:

```
<new input action>
<new extract action>
<new prompt action>
<new message action>
<new pause action>
<new xfer action>
<new comm wait action>
<new trace action>
<new mouse click action>
<new box select action>
<new run program>
<new variable update action>
<new play macro action>
<new perform action>
<new conditional action>
<new print start action>
<new print extract action>
<new print end action>
```

*Figure 28. Contents of the list of an Actions listbox with no actions created*

For example, if you clicked <new input action>, the Macro object would create a new Input action and place it at the top of the list. The lower area of the Actions tab would allow you to fill out the various fields that belong to an Input action (such as the input key sequence). The new Input item would be in the selected area of the Actions listbox, and the list part of the listbox would then look like this:

```
Input action1(0,0)
<new input action>
<new extract action>
<new prompt action>
<new message action>
<new pause action>
<new xfer action>
<new comm wait action>
<new trace action>
<new mouse click action>
<new box select action>
<new run program>
<new variable update action>
<new play macro action>
<new perform action>
<new conditional action>
<new print start action>
<new print extract action>
<new print end action>
```

*Figure 29. Contents of the list of an Actions listbox with one actual action*

When the macro runtime processes this macro screen, it will perform the actions in the same order in which they are listed in the Actions listbox. To change the order of the actual actions, click the Change Order button to the right of the Actions listbox.

# The actions

## Box selection action (<boxselection> element)

The Box selection action draws a marking rectangle on the session window, simulating the action in which an actual user clicks on the session window, presses mouse button 1, and drags the mouse to create a marking rectangle.

### Specifying row and column values

In the lower area of the Actions tab, specify the row and column locations of the corners of the marking rectangle that you want to create or edit. Or, if the Box selection action is the currently selected action in the Actions listbox, you can click and drag the mouse over the session window to create the marking rectangle that you want to create. When you have completed the marking rectangle on the session window, the Macro Editor writes the row and column value of the corners of the marking rectangle into the appropriate Row and Column input fields of the Box selection action.

You can type a negative number (such as -1) into the Row (Bottom Corner) input field and into the Column (Bottom Corner) input field to specify a relative offset from the last row and last column of the session window. For example, if the session window is 24 by 80 and you specify corners of (1, 1) and (-4, -10), then the macro runtime will draw a marking rectangle with the coordinates (1, 1) and (21, 71) (see "Significance of a negative value for a row or column" on page 39).

### Erasing the marking rectangle

If you draw a marking rectangle and then click the session window anywhere outside the boundaries of the marking rectangle, the session window erases the marking rectangle. However, when the macro runtime is playing a macro, a Mouse click action does not erase an existing marking rectangle.

To erase a marking rectangle in a macro you have two choices:

- Draw a new marking rectangle in a different location .When you draw a new marking rectangle the old one is erased.
- Use the Code Editor to set the **type** attribute of the <boxselection> element to DESELECT. For the DESELECT option the macro runtime ignores the row and column coordinates and just erases the existing marking rectangle if there is one.

### Example

See "Copy and paste example" on page 85.

## Comm wait action (<commwait> element)

The Comm wait action waits until the communication status of the session changes to some state that you have specified in the action. For example, you might create a Comm wait action to wait until the session was completely connected.

### How the action works

When the macro runtime starts to perform a Comm wait action, it looks at the communication status specified in the Comm wait action and compares it to the actual communication status of the session. If the two statuses match, then the macro runtime concludes that the Comm wait action is completed, and the macro runtime goes on to perform the next action.

However, if the two statuses do not match, then the macro runtime does no further processing, but just waits for the communication status that is specified in the Comm wait action to actually occur in the session.

You can specify in the Comm wait action a timeout value in milliseconds that causes the macro runtime to end the Comm wait action when the timeout value has expired. That is, the macro runtime terminates the action when the timeout value has expired, even if the communication status that the macro runtime has been looking for has not been reached.

After a Comm wait action, you probably want to use some other action, such as an Extract action, to check some characteristic of the application screen that will indicate to you whether the session has actually reached the communication status that you wanted, or whether the Comm wait action ended because of a timeout.

## Specify a communication status that is persistent

As the session connects or disconnects, the communication status typically moves quickly through some states (such as pending active, then active, then ready) until it reaches a particular state at which it remains stable for some time (such as workstation id ready). In most situations you want to specify that persistent, ending state in the Comm wait action. See the next section for a list of persistent states.

(If instead you specified some transitional state such as pending active, then the session might pass through that state and go on to the next state before the macro runtime gets a chance to perform your Comm wait action. Therefore when the macro runtime does perform your Comm wait action it will be waiting interminably for some state that has already occurred.)

## Communication states

You can specify any of the states listed in the Connection Status listbox. Table 15 lists the name and significance of each state:

*Table 15. Communication states*

| **Communication state:** | **Significance:** |
|---|---|
| Connection Initiated | Initial state. Start Communications issued. |
| Connection Pending Active | Request socket connect. |
| Connection Active | Socket connected. Connection with host. |
| Connection Ready | Telnet negotiation has begun. |
| Connection Device Name Ready | Device name negotiated. |
| Connection Workstation ID Ready | Workstation ID negotiated. |
| Connection Pending Inactive | Stop Communications issued. |
| Connection Inactive | Socket closed. No connection with host. |

The stable states (that is, the ones that usually persist for more than a few seconds) are:

- Connection Inactive - Here the session is completely disconnected.
- Connection Workstation ID Ready - Here the session is completely connected.

If you select `<Expression>` in the Connection Status listbox, then you must specify an expression that resolves to one of the keywords that the macro runtime expects to find in the **value** attribute of the <commwait> element (see "<commwait> element" on page 147). For example, you might specify a variable named `$strCommState$`) that resolves to `CONNECTION_READY`.

## Examples

```
<commwait value="CONNECTION_READY" timeout="10000" />
```

*Figure 30. Example of Comm wait action*

## Conditional action (<if> element and <else> element)

The Conditional action contains the following items:

- A conditional expression that the macro runtime evaluates to true or false.
- A list of actions that the macro runtime performs if the condition evaluates to true. (Optional)
- A list of actions that the macro runtime performs if the condition evaluates to false. (Optional)

The Conditional action provides the functions of an if-statement or of an if-else statement.

### Specifying the condition

You should specify in the Condition field the conditional expression that you want the macro runtime to evaluate. The conditional expression can contain logical operators and conditional operators and can contain terms that include arithmetic expressions, immediate values, variables, and calls to Java methods (see "Conditional and logical operators and expressions" on page 36).

### Condition is True (<if> element)

Use the Condition is True tab to specify the actions that you want to be performed if the condition evaluates to true.

The Condition is True tab contains controls that are almost identical to the controls for the Actions tab. Specifically:

- The Action listbox on the Condition is True tab allows you to create and edit actions in the same way that the Action listbox on the Actions tab does.
- The Delete button and the Change Order button on the Condition is True tab allow you to delete or reorder actions in the same way that the Delete button and the Change Order button on the Actions tab do.
- The lower area of the Condition is True tab allows you to edit the values of the currently selected action in the same way that lower area of the Actions tab does.

Use these controls on the Condition is True tab to create and edit the actions that you want the macro runtime to perform if the condition is true.

For example, you might set the Condition field to the name of a variable, such as `$boolResult$`, into which a previous operation has stored its result. If the value of $boolResult$ is true, you might want to a perform a Message action that displays a status message to the user. Therefore, in the Condition is True tab you would create a Message action that contains the status message that you want to display.

During macro playback the macro would evaluate the condition, $boolResult$. If the value is true then the macro runtime would perform the Message action that you had defined in the Condition is True tab. If the value is false then the macro runtime would skip over the Message action and over all the other actions (if any) that you had defined in the Condition is True tab.

## Condition is false (<else> element)

Use the Condition is False tab to specify the actions that you want to be performed if the condition evaluates to false.

Like the Condition is True tab, the Condition is False tab contains controls that are almost identical to the controls for the Actions tab. Use these controls on the Condition is False tab to create and edit the actions that you want the macro runtime to perform if the condition is false.

## Condition action not allowed within a Condition action

The Macro Editor does not allow you to create a Condition action inside the Condition is True tab or inside the Condition is False tab. Therefore you cannot have the equivalent of an if-statement nested inside another if-statement, or of an if-statement nested inside an else-statement. The Code Editor enforces the same rules.

## Example

The following code fragment prompts the user for input. If the input string is the string `true`, the code fragment displays a message window with the message ″You typed TRUE″. If the input string is any other string, the code fragment displays a message window with the message ″You typed FALSE″. This example uses the following actions: Prompt action, Condition action, and Message action.

You can copy this code fragment from this document into the system clipboard, and then from the system clipboard into the Code Editor. Because this code is a fragment, you will have to copy it into a macro screen in an existing macro script. You will also have to create a string variable named $strData$. To create the variable, add the follow lines after the <HAScript> begin tag and before the first <screen> element:

```
<vars>
   <create name="$strData$" type="string" value="" />
</vars>
```

After you save the script in the Macro Editor, you can edit it either with the Macro Editor or with the Code Editor.

You should notice the following facts about this example:

- The example consists of one code fragment containing an <actions> element and the actions inside it.
- The first action is a Prompt action that displays a message window and copies the user's input into the variable $strData$, without writing the input into an input field in the session window.
- The first part of the condition action (the <if> element) contains the condition, which is simply `$strData$`.
- Because $strData$ is a string variable in a boolean context, the macro runtime tries to convert the string to a boolean value (see "Automatic data type conversion" on page 37). If the user's input is the string `'true'` (in upper, lower, or mixed case), then the conversion is successful and the condition contains the boolean value true. If the user's input is any other string, then the conversion fails and the condition contains the boolean value false.
- If the condition is true, then the macro runtime performs the action inside the <if> element, which is a Message action displaying the message You typed TRUE. Then, having exhausted all the actions to be performed when the condition is true (just one action here), the macro runtime skips over the <else> action and continues as usual.

- In contrast, if the condition is false, then the macro runtime skips over the actions in the <if> element and begins performing the actions in the <else> element, which include one Message action that displays the message You typed FALSE. After performing all the actions in the <else> action, then the macro runtime continues as usual.

```
<actions>
   <prompt name="'Type true or false'" description="" row="0" col="0"
               len="80" default="" clearfield="false" encrypted="false"
               movecursor="true" xlatehostkeys="true" assigntovar="$strData$"
               varupdateonly="true" />
   <if condition="$strData$" >
       <message title="" value="'You typed TRUE'" />
   </if>
   <else>
       <message title="" value="'You typed FALSE'" />
   </else>
</actions>
```

*Figure 31. Sample code fragment showing a Condition action*

## Extract action (<extract> element)

The Extract action captures text from the session window and stores the text into a variable. This action is very useful and is the primary method that the Macro object provides for reading application data (short of using programming APIs from the toolkit).

If you have the IBM Host Access Toolkit product, then you can use the Extract action to read data from any of the available data planes. For more information see "Using the Toolkit to capture data from any data plane" on page 82.

### Capturing text

The most common use of the Extract action is to capture text that is being displayed in the session window. This operation does not require the IBM Host Access Toolkit.

Here is an overview of the steps to follow. Each step is described in more detail in the following subsections.

1. Set the Continuous Extract option, if necessary
2. Specify an area on the session window that you want to capture.
3. Specify an extraction name.
4. Specify TEXT_PLANE as the data plane.
5. Specify a variable in which you want the text to be stored.

**Set the Continuous Extract option:** If you want to capture a rectangular block of text, then set the Continuous Extract option to false (this is the default value). For more information, see "Capturing a rectangular area of the session window" on page 79.

In contrast, if you want to capture a continuous sequence of text that wraps from line to line, then set the Continuous Extract option to true. For more information, see "Capturing a sequence of text from the session window" on page 79.

**Specify the area of the session window:** To specify the area of the session window that you want to capture, you can either use the marking rectangle to

gather the row and column coordinates, or else you can type the row and column coordinates of the text area into the Row and Column fields on the Extract action window.

Whichever method you use, the macro runtime will interpret the values differently depending on whether the Continuous Extract option is set to false or true (see "Set the Continuous Extract option" on page 77).

If you are using the marking rectangle (see "Using the marking rectangle" on page 129) then the macro runtime writes the row and column coordinates of the upper left corner of the marking rectangle into the first pair of Row and Column values (labeled Top Corner on the Extract action window) and the row and column coordinates of the lower right corner into the second pair of Row and Column values (labeled Bottom Corner).

If you enter the Row and Column values yourself, then type the first set of row and column coordinates into the first pair of Row and Column values (labeled Top Corner on the Extract action window) and type the second set of coordinates into the second pair of Row and Column values (labeled Bottom Corner). You can use the text cursor on the session window as an aid to determine the coordinates that you want (see "Using the session window's text cursor" on page 129).

In the Row (Bottom Corner) input field you can enter `-1` to signify the last row of the data area on the session window. This feature is helpful if your users work with session windows of different heights (such as 25, 43, 50) and you want to capture data down to the last row. Similarly for the Column (Bottom Corner) input field you can enter `-1` to signify the last column of the data on the session window (see "Significance of a negative value for a row or column" on page 39).

**Specify an extraction name:** You must specify an extraction name, such as `'Extract1'`. However, you will not use this name for any purpose unless you are using the IBM Host Access Toolkit product.

**Specify TEXT_PLANE as the data plane:** In the Data Plane listbox click TEXT_PLANE. This is the default.

**Specify the variable in which you want the text to be stored:** Set the checkbox labeled Assign Text Plane to a Variable and enter the name of the variable into which you want the text to be stored. You have to use this method to store the captured text unless you are using the IBM Host Access Toolkit product.

The text is returned as a string. In most cases you will probably want to store the string in a string variable, so that some other action in your macro can process the string.

However, if you specify a variable of some other standard data type (boolean, integer, double) then the macro runtime will convert the string to the format of the variable, if possible. For example, if the text on the screen is `1024` and the variable is an integer variable then the macro runtime will convert the string `1024` to the integer 1024 and store the value in the integer variable. If the format is not valid for converting the string to the data type of the variable then the macro runtime will terminate the macro with a run-time error. For more information about data conversion see "Automatic data type conversion" on page 37.

### Treatment of nulls and other undisplayable characters

Text captured from the TEXT_PLANE does not contain any nulls (0x00) or other undisplayable characters. Any character cell on the display screen that appears to contain a blank space character will be captured as a blank space character.

### Capturing a rectangular area of the session window

When the Continuous Extract option is false (this is the default value), the macro runtime treats the two pairs of Row and Column values as the upper left and lower right corners (inclusive) of a rectangular block of text. The rectangular block can be as small as one character or as large as the entire application window.

The macro runtime:

- Initializes the result string to an empty string.
- Reads the rectangular block of text row by row, concatenating each row to the result string.
- Stores the result string in the specified variable.

As an example, suppose that the first 40 characters of rows 16, 17, and 18 of the session window are as follows:

```
.8..Outlist.....Display, delete, or prin
.9..Commands....Create/change an applica
.10.Reserved....This option reserved for
```

and suppose that the macro runtime is about to perform an Extract action with the following settings:

- Continuous Extract is false.
- The row and column pairs are (16, 5) (the 'O' of Outlist) and (18, 12) (the 'd' of 'Reserved').
- The extraction name is 'Extract1'.
- The data plane is TEXT_PLANE.
- The string variable $strTmp$ is the variable in which the result string is to be stored.

Because the Continuous Extract option is false, the macro runtime treats the row and column pairs as marking a rectangular block of text, with the upper left corner at row 16 and column 5 and the lower right corner at row 18 and column 12.

The macro runtime initializes the result string to an empty string. Then the macro runtime reads the rectangular block of text one row at a time ('Outlist.', 'Commands', 'Reserved'), concatenating each row to the result string. Finally the macro runtime stores the entire result string into the result variable $strTmp$. The variable $strTmp$ now contains the following string:

```
'Outlist.CommandsReserved'
```

### Capturing a sequence of text from the session window

When the Continuous Extract option is true, the macro runtime treats the two pairs of Row and Column values as the beginning and ending positions (inclusive) of a continuous sequence of text that wraps from line to line if necessary to get from the beginning position to the ending position. The sequence of text can be as small as one character or as large as the entire application window.

The macro runtime:

- Initializes the result string to an empty string.

- Reads the continuous sequence of text from beginning to end, wrapping around from the end of one line to the beginning of the next line if necessary.
- Stores the result string in the specified variable.

For example, suppose that rows 21 and 22 of the session window contain the following text (each row is 80 characters):

```
........Enter / on the data set list command field for the command prompt pop-up
or ISPF line command.....................................................
```

and suppose that the macro runtime is about to perform an Extract action with the following settings:

- Continuous Extract is true.
- The row and column pairs are (21, 9) (the 'E' of 'Enter') and (22, 20) (the 'd' of 'command').
- The extraction name is 'Extract1'.
- The data plane is TEXT_PLANE.
- The string variable $strTmp$ is the variable in which the result string is to be stored.

Because the Continuous Extract option is true, the macro runtime treats the row and column pairs as marking the beginning and end of a sequence of text, with the beginning position at (21, 9) and the ending at (22, 20).

The macro runtime initializes the result string to an empty string. Then the macro runtime reads the sequence of text from beginning to end, wrapping around from the last character of row 21 to the first character of row 22. Finally the macro runtime stores the entire result string into the result variable $strTmp$. The variable $strTmp$ now contains the following string of 92 characters (the following text is hyphenated to fit on the page of this document, but actually represents one string without a hyphen):

```
'Enter / on the data set list command field for the com-
mand prompt pop-up or ISPF line command'
```

In contrast, if the Continuous Extract option is set to false in this example, $strTmp$ would contain a string of 24 characters, `'Enter / on tline command'`.

## Unwrap Text option

This option was originally intended to be used with the IBM Host Access Toolkit product and only when the Continuous Extract option was set to false. However, you can also use it without the toolkit, and you can use it with the Continuous Extract option set either to false or true.

When you set Unwrap Text to true, the macro runtime uses not only the row and column pairs in the Extract window but also the field boundaries in the session window in determining the data to collect. The macro runtime returns an array of strings (if you are using the toolkit) or a single string of concatenated strings (if you are not using the the toolkit).

Do not confuse the Unwrap Text option with the Wrap option of the String descriptor, which is based on a rectangular block of text rather than fields (see "How the macro runtime searches the rectangular area (Wrap option)" on page 62).

**When Unwrap Text is true and Continuous Extract is false:** When Continuous Extract is false, the row and column pairs represent the corners of a rectangular

block of text. When you set Unwrap Text to true, the macro runtime reads each row of the rectangular block of text and processes each field in the row as follows:

- If the field begins outside the row and continues into the row, then the macro runtime ignores the field.
- If the field begins inside the row and ends inside the row, then the macro runtime includes the field's contents in the result.
- If the field begins inside the row and ends outside the row, then the macro runtime includes the contents of the entire field (including the part outside the rectangular block of text) in the result.

The intent of the Unwrap Text option is to capture the entire contents of a field as one string even if the field wraps from one line to the next.

For example, suppose that the session window is 80 characters wide and that rows 9, 10, 11, and 12 of the session window are as follows:

```
.............................................Compress or print data set.......
.............................................................Reset statistics..
.............................................................Catalog or display
information of an entire data set.........................................
```

Suppose also that the following text areas in the lines above are fields:

```
Compress or print data set
Reset statistics
Catalog or display information of an entire data set
```

Finally, suppose that:

- Continuous Extract is false (this is the default setting).
- Unwrap Text is true.
- The marking rectangle has its upper left corner at row 9 and column 63 (the 'n' of 'print') and its lower right corner at row 11 and column 73 (the ' ' after 'or').
- The extraction name is 'Extract1'.
- The data plane is TEXT_PLANE.

If you are using the IBM Host Access toolkit the macro runtime returns the following array of strings as the return value of the toolkit method: `'Reset statistics', 'Catalog or display information of an entire data set'`. The macro runtime:

- Skips `'Compress or print data set'` because the field begins outside the marking rectangle.
- Returns `'Reset statistics'` because the field begins within the marking rectangle.
- Returns `'Catalog or display information of an entire data set'` because the field begins inside the marking rectangle, even though the field wraps to the next line.

If you are not using the toolkit, the macro runtime concatenates the individual strings and stores them as a single string into the variable that you specified in the Extract window. In this example the macro runtime stores the string `'Reset statisticsCatalog or display information of an entire data set'` into the variable.

**When Unwrap Text is true and Continuous Extract is true:** When Continuous Extract is true, the row and column pairs represent the beginning and ending

locations of a continuous sequence of text that wraps from line to line if necessary. When you then set Unwrap Text to true, the macro runtime processes the continuous sequence of text as follows:

- If the field begins outside the sequence and continues into the sequence, then the macro runtime ignores the field.
- If the field begins inside the sequence and ends inside the sequence, then the macro runtime includes the field's contents in the result.
- If the field begins inside the sequence and ends outside the sequence, then the macro runtime includes the contents of the entire field (including the part outside the continuous sequence) in the result.

### Using the Toolkit to capture data from any data plane

You can use the Java APIs from the IBM Host Access Toolkit product to access data from any data plane, including the TEXT_PLANE.

The data planes, together with the type of data associated with each plane, are:

- TEXT_PLANE - The characters displayed on the screen.
- FIELD_PLANE - 3270 or 5250 field attributes
- COLOR_PLANE - 3270 or 5250 color attributes
- EXFIELD_PLANE - 3270 or 5250 extended attributes
- DBCS_PLANE - Double byte character set characters
- GRID_PLANE - Double byte character set grid information

To access the extracted data using the Toolkit, you will need to implement the MacroRuntimeListener class and register yourself with the Macro bean. For every Extract action, the Macro bean will fire data to you in a MacroExtractEvent by calling your macroExtractEvent() method. Use the get methods of the MacroExtractEvent to access the data.

## Input action (<input> element)

The Input action sends a sequence of keystrokes to the session window. The sequence can include keys that display a character (such as a, b, c, #, &, and so on) and also action keys (such as [enterreset], [copy], [paste], and others).

This action simulates keyboard input from an actual user.

### Location at which typing begins

Use the Row and Column fields to specify the row and column location in the session window at which you want the input sequence to begin. For example, if you specify row 23 and column 17 in the Input action, and you specify `Hello world` as the String value of the Input action, then (assuming that the location you have specified lies within an input field) the macro runtime will type the key sequence `Hello world` on the session window starting at row 23 and column 17.

If you specify a row or column location of 0, then the macro runtime will type the key sequence beginning at the actual row and column location of the text cursor on the session window when the Input action is performed. You should not specify a row or column of 0 unless the context is one in which the location of the text cursor does not matter (for example, with a [copy] action key) or unless you can predict where the text cursor will be located (for example, if a Mouse click action has just moved the text cursor to a specific location, or if the application has positioned the text cursor as part of displaying the application screen).

## Input errors

During macro playback, the session window reacts to a key input error in the same way as it would react if an actual user had typed the key.

For example, if an Input action sends a key that displays a character (such as a, b, c, #, & and so on) to the session when the text cursor is not located in a 3270 or 5250 input field, then the session will respond by inhibiting the key input and displaying an error symbol in the Operator Information Area, just as it would with a keystroke typed by an actual user.

## Input string

The String field is an input field in which you specify the key sequence that you want the action to perform.

To specify a key that causes a character to be displayed (such as a, b, c, #, &, and so on), type the key itself.

To specify a key from the Actions Keys listbox, scroll the list to the key you want (such as `[backspace]`) and click Insert Action Key. The name of the key enclosed by square brackets appears at the next input position in the String field. Please notice that the keys in the Action Keys listbox are not listed alphabetically throughout. You might have to keep scrolling down the list to find the key you want.

Another way to specify an action key is just to type the name itself into the String input field, surrounded by square brackets (for example, `[backspace]`).

The following copy/paste keys occur in the Action Keys list for a 3270 Display Session:

```
[copy]                  [mark right]
[copyappend]            [mark up]
[cut]                   [paste]
[mark down]             [pastenext]
[mark left]             [unmark]
```

For other keys see “Mnemonic keywords for the Input action” on page 177.

## Translate Host Action Keys

The Translate Host Action Keys field indicates whether the macro runtime is to interpret action key names (such as [copy], [enterreset], [tab], and so on) in the input sequence as action keys or as literal sequences of characters. The default is true (interpret the action key names as action keys).

For example, suppose that the input key sequence is `'[up][up]Hello world'` and that the text cursor is at row 4, column 10. If the Translate Host Actions Keys value is true, then in performing this input sequence the macro runtime will move the text cursor up two rows and then type `Hello world` beginning at row 2, column 10. In contrast, if the Translate Host Actions Keys value is false, then the macro runtime will type `[up][up]Hello World` beginning at row 4, column 10.

## Move Cursor to End of Input

When the Translate Host Action Keys field is set to true (the default), then the Macro Editor also sets the Move Cursor to End of Input listbox to true and disables it. Even though the listbox is disabled, its value is still set to true.

If you set the Translate Host Action Keys listbox to false, then the Move Cursor to End of Input listbox is enabled and you can set it to false, true, or an expression that is evaluated at runtime.

When the value of this listbox is true (the default), then the macro runtime moves the text cursor in the same way that it would be moved if an actual user were entering keyboard input. For example, if the key is a text character, such as 'a', then the macro runtime types the character on the session window and then moves the text cursor to the first character position after the 'a'. Similarly, if the key is [tab], then the macro runtime moves the text cursor to the next tab location.

In contrast, if the value of the Move Cursor to End of Input listbox is false, then the macro runtime does not move the text cursor at all. The text cursor remains in the same position as it occupied before the macro runtime performed the Input action.

### Example
See "Copy and paste example" on page 85.

## Message action (<message> element)

The Message action displays a popup window that includes a title, a message, and an OK button. The macro runtime does not terminate the action until the user clicks OK.

You can use this message in many scenarios, such as the following:

- To display an instruction, error message, or status message to the user.
- To suspend execution of the macro until the user has performed some action.
- To display values for debugging.

### Displaying the message caption and message text

You should specify in the Message Title input field the caption that you want to be displayed in the caption bar of the message window.

You should specify in the Message Text input field the text that you want to be displayed inside the message window.

Because both input fields expect a string as the input value, you can specify any entity that evaluates to a string (see "Equivalents" on page 38). If you use an arithmetic expression then the expression itself can contain immediate values, variables, arithmetic expressions, and calls to Java methods (see "Arithmetic operators and expressions" on page 35).

You can also use the data type conversion rules (see "Automatic data type conversion" on page 37) and the string concatenation operator (see "String concatenation operator (+)" on page 36). For example, if you want to display the value of an integer variable named `$intResult$`, then you can specify in the Message Text input field the following string:

```
'The result is ' + $intResult$ + '.'
```

If the value of $intResult$ is 204, then the macro runtime displays in the message box the following text:

```
The result is 204.
```

## Mouse click action (<mouseclick> element)

The Mouse click action simulates a user mouse click on the session window. As with a real mouse click, the text cursor jumps to the row and column position where the mouse icon was pointing when the click occurred.

### Specifying row and column

In the lower area of the Actions window, specify the row and column location on the session window where you want the mouse click to occur. Or, you can click on the session window itself, and the Macro Editor updates the values in the Row and Column fields to reflect the new location of the text cursor.

### Copy and paste example

The following example shows how to mark a block of text in the session window, copy it to the system clipboard, and paste it back into the session window at a new location. This example uses the following action elements: Box selection action, Input action, Mouse click action, and Pause action.

You can copy the text of this macro script from this document into the system clipboard, and then from the system clipboard into the Code Editor (see "Copy and paste a script from this guide into the Code Editor" on page 130). After you save this script in the Macro Editor, you can edit it either with the Macro Editor or with the Code Editor.

You should notice the following facts about this example:

- The example consists of one entire macro script named COPY PASTE.
- The following actions occur in the <actions> element:
  - The <boxselection> action draws a marking rectangle.
  - A <pause> action waits one-half second so that when the macro is played back, the user can see what is happening.
  - An <input> action types a [copy] action, which copies the marked area to the clipboard.
  - A <mouseclick> action sets the cursor to the location where the paste will take place.
  - An <input> action types a [paste] key, which pastes the contents of the clipboard to the new location on the session window.
- This macro is written to be run from the ISPF Primary Option Menu (see Figure 5 on page 14). The macro copies the text `Spool Display and Search Facility` from row 18 to the system clipboard, and then pastes the text from the clipboard to the `Option ===>` input field in line 4.
- If this example does not paste properly when you run it, make sure that the target area that you have specified lies within a 3270 or 5250 input field. The Host On-Demand client does not let you paste text into a protected field in the application screen.

```
<HAScript name="COPY PASTE" description=" " timeout="60000" pausetime="300"
            promptall="true" author="" creationdate="" supressclearevents="false"
            usevars="false" >

   <screen name="Screen1" entryscreen="true" exitscreen="true"
            transient="false">
      <description>
         <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
      </description>
      <actions>
         <boxselection type="SELECT" srow="18" scol="19"
                 erow="18" ecol="51" />
         <pause value="500" />
         <input value="[copy]" row="0" col="0" movecursor="true"
                 xlatehostkeys="true" encrypted="false" />
         <mouseclick row="4" col="15" />
         <input value="[paste]" row="0" col="0" movecursor="true"
                 xlatehostkeys="true" encrypted="false" />
      </actions>
      <nextscreens timeout="0" >
      </nextscreens>
   </screen>

</HAScript>
```

*Figure 32. Sample code COPY PASTE*

## Pause action (<pause> element)

The Pause action waits for a specified number of milliseconds and then terminates.

More specifically, the macro runtime finds the <pause> element, reads the duration value, and waits for the specified number of milliseconds. Then the macro runtime goes on to perform the next item.

Uses for this action are:

- Any situation in which you want to insert a wait.
- Waiting for the host to update the session window. For more information see "Screen completion" on page 112.
- To add delay for debugging purposes.

You should type the number of milliseconds in the Duration input field. The default is 10000 milliseconds (10 seconds).

## Perform action (<perform> element)

The Perform action invokes a method belonging to a Java class that you have imported (see "Creating an imported type for a Java class" on page 121).

You can invoke a method in many other contexts besides the Perform action. However, the Perform action is useful in certain scenarios, for example, when you want to invoke a method that does not return a value.

Some of the contexts, other than the Perform action, in which you can invoke a method are as follows:

- You can invoke a method and assign the return value to a variable by using the Update variable action. The variable that receives the return value can be either a variable belonging to a standard type (boolean, integer, string, double) or a variable belonging to an imported type (for example, a variable named $objTmp$ that belongs to the imported type Object, based on the Java class Object).
- You can invoke a method and use the return value as a parameter in a macro action by specifying the method call in the field for the parameter. For example, in the Row parameter of an Extract action you can use a method call that returns an integer value. The macro runtime sees that the parameter is a method call, invokes the method, and uses the integer return value as the value of the Row parameter.
- You can invoke a method as part of any expression by using the method call as a term in the expression. When the macro runtime evaluates the expression, it sees that the term is a method call, invokes the method, and uses the value of the method (for example, a string) as the value of the term.
- You can invoke a method and use the return value as the initial value of a variable that you have just declared.

In general, outside the Perform action, you can invoke a method in any context in which the value returned by the method is valid.

## Invoking the method

Type the method call into the Action to Perform field. You must enclose a method call in dollar signs ($), just as you would a variable (see "Syntax of a method call" on page 126). The macro runtime will invoke the method. See also "How the macro runtime searches for a called method" on page 126.

## Examples

The following examples show how to invoke a method using the Perform action. You should notice the following facts about these examples:

- In Example 1, the Perform action calls the update() method on the variable $importedVar$. Notice that:
  - The entire method call is enclosed in dollar signs ($).
  - In the context of a method call, the variable name itself (`importedVar`) is not enclosed in dollar signs ($).
  - A variable passed as a parameter to a method must be enclosed in dollar signs ($) as usual (`$str$`).
  - A string passed as a parameter to a method must be enclosed in single quotes as usual (`'Application'`).
- In Example 2, the Perform action calls a static method.
- In Example 3, the Perform action calls the close() method belonging to the class to which the variable belongs, such as java.io.FileInputStream.
- In Example 4, the Perform action calls the createZipEntry() method belonging to the class to which the variable belongs, such as java.util.zip.ZipInputStream.
- In Example 5, the Perform action calls the clear() method belonging to the class to which the variable belongs, such as java.util.Hashtable.

```
<actions>
   <!-- Example 1 -->
   <perform value="$importedVar.update( 5, 'Application', $str$)$" />

   <!-- Example 2 -->
   <perform value="$MyClass.myInit('all')$" />

   <!-- Example 3 -->
   <perform value="$fip.close()$" />

   <!-- Example 4 -->
   <perform value="$zis.createZipEntry( $name$ )$" />

   <!-- Example 5 -->
   <perform value="$ht.clear()$" />
</actions>
```

*Figure 33. Examples of the Perform action*

## PlayMacro action (<playmacro> element)

The PlayMacro action launches another macro.

When the macro runtime performs a PlayMacro action, it terminates the current macro (the one in which the PlayMacro action occurs) and begins to process the specified macro screen of the target macro. This process is called chaining. The calling macro is said to ″chain to″ the target macro. There is no return to the calling macro.

You must specify in the PlayMacro action the name of the target macro and, optionally, the name of the macro screen in the target macro that you want the macro runtime to process first.

You can have the macro runtime transfer all of the variables with their contents from the calling macro to the target macro.

### Adding a PlayMacro action

Outside of a Condition element:

- You can add only one PlayMacro action to a macro script, and that PlayMacro action must be the last action in the Actions list (<actions> element) of the macro script.

Inside a Condition element:

- You can add one PlayMacro action to the Condition is True branch (<if> element), and that PlayMacro action must be the last action in the branch (<if> element).
- You can also add one PlayMacro action to the Condition is False branch (<else> element), and that PlayMacro action must be the last action in the branch (<else> element).

You can have as many Condition elements in the macro as you like, with each Condition element containing one PlayMacro action in its Condition is True branch and one PlayMacro action in its Condition is False branch.

The Macro Editor and the Code Editor enforce these rules.

## Target macro file name and starting screen

Use the Macro Name field to specify the name of the target macro. If you are chaining macros in a Server Library, you must specify the name of the macro file rather than the name of the macro.

You cannot call a macro that resides in a different location than the calling macro. Specifically:

- A macro in the Current Session can call only macros that are in the Current Session.
- A macro in the Personal Library can call only macros that are in the Personal Library.
- A macro in a Server Library can call only macros that are in that Server Library.

Use the Start Screen Name listbox to select the macro screen in the target macro that you want the macro runtime to process first:

- If you want to start the target macro at its usual start screen, then select the *DEFAULT* entry in the Start Screen Name listbox, or provide an expression that evaluates to the value *DEFAULT*.
- If you want to start the target macro at some other screen, then select the name of that screen in the Start Screen Name listbox.

## Transferring variables

You can have the macro runtime transfer to the target macro all the variables that belong to the calling macro, including the contents of those variables, by setting the Variable Transfer listbox to Transfer (the default is No Transfer).

This transferring of variables and their contents allows you to use variables to pass parameters from the calling macro to the target macro.

After the target macro gets control, it can read from and write to the transferred variables in the same way that it reads from and writes to variables that it has declared itself.

For example, if MacroA currently has two integer variables named StartRow and StartCol, with values of 12 and 2, and then MacroA launches MacroB with a PlayMacro action, then MacroB will start out having variables StartRow and StartCol with values of 12 and 2.

Even if the transferred variable belongs to an imported type and contains a Java object, the target macro can still refer to the transferred variable and call methods on the Java object, or can write some other object into that transferred variable.

**Requirements for transferring variables:** The target macro must have selected the advanced macro format (see "Choosing a macro format" on page 31).

**Restriction:** Please notice the following restriction on all types of transferred variables:

- You cannot use the transferred variable in the Initial Value field of the Variables tab of the target macro.

**Additional information:** If the target macro creates a variable with the same name and type as a transferred variable, then the macro runtime uses the created variable rather than the transferred variable.

**When the target macro needs to import a type:** In the target macro, if you want to use a transferred variable that belongs to an imported type, then you do not need to import that same type in the target macro. Examples of operations where you do not need to import the type are as follows:

- Using the transferred variable as the value of an attribute.
- Calling a method on the transferred variable.

However, in the target macro, if you want to use the name of an imported type, then you must import that type. Examples of operations where you must import the type:

- Declaring a new variable of the imported type.
- Creating a new instance of the imported type.
- Calling a static method of the imported type.

### Examples

The following example shows a PlayMacro action:

```
<actions>
   <playmacro name="TargetMacro" startscreen="*DEFAULT*"
            transfervars="Transfer" />
</actions>
```

*Figure 34. Example of the PlayMacro action*

## Print actions (<print> element)

The Print actions allow you to print text from the session window of a 3270 Display session. You can print the entire application screen, or you can print a rectangular area of text from the application screen. You can use the same printer setup options and most of the same page setup options that are available for a 3270 Printer session.

You can use the Print actions only with a 3270 Display session.

The Print actions do not create a host print session. Rather, the Print actions print data that is displayed in the 3270 Display session window (screen print).

The Print actions are:

- Print Start
- Print Extract
- Print End

The Print Start action instantiates a print bean object for the current macro and sets the Printer Setup options and the Page Setup options for the bean. The Print Extract action sends text to the print bean. The Print End action terminates the print bean.

Although the Macro Editor presents Print Start, Print Extract, and Print End as separate types of action, in fact the Macro object stores all three using the <print> element.

### Print Start

The Print Start action instantiates a print bean object for the current macro using the Printer Setup options and the Page Setup options that you specify.

Before performing a Print Start action, the macro runtime checks to see if a print bean already exists for the current macro. If so, then the macro runtime terminates the existing print bean and then performs the Print Start action to instantiate a new print bean.

**Printer Setup and Page Setup:** Click Printer Setup to set the printer setup options for the new print bean. You can control the same printer setup options as are available for a 3270 Printer session, including Printer destination (Windows Printer, Other Printer, or File), Printer Definition Tables, and Adobe PDF output for the File destination.

Click Page Setup to set the page setup options for the new print bean. You can control the same page setup options that are available for a 3270 Printer session and that also are appropriate for the 3270 Display datastream (LU2), including Font, treatment of nulls (0x00), and Printer-Font Code Page.

The Printer Setup options and the Page Setup options that you specify for a print bean for the current macro do not affect the printer setup options and page setup options for:

- A print bean in a macro running on another 3270 Display session.
- Any ZipPrint in any 3270 Display session.
- Any File > Print Screen operation in any 3270 Display session.
- A 3270 Printer session.

However, if your print destination is a Windows printer, and you use the Microsoft Windows Print Setup dialog to make configuration changes to that Windows printer, such as an orientation of Landscape rather than Portrait, then those particular configuration changes will affect any Host On-Demand printing activity that uses that Windows printer, including:

- A print bean in a macro running on any 3270 Display session.
- ZipPrint in any 3270 Display session.
- Printing from any 3270 Printer session.

**Assign Return Code to a Variable:** If you want to verify that the Print Start action is successful, then click Assign Return Code to a Variable and select a variable to hold the return code from the Print Start action.

## Print Extract

The Print Extract action copies the text from a rectangular area of the 3270 Display session window that you specify and prints the text using the current print bean.

Before performing a Print Extract action, the macro runtime checks to see if a print bean has been started for the current macro. If not, then the macro runtime performs a Print Start action with the default printer setup options and the default page setup options, and then performs the Print Extract action.

**Specifying the area to be printed:** To specify the area of the session window that you want to print, you can either use the marking rectangle to gather the row and column coordinates, or else you can type the row and column coordinates of the text area into the Row and Column fields on the Extract action window.

If you are using the marking rectangle (see “Using the marking rectangle” on page 129) then the macro runtime writes the row and column coordinates of the upper left corner of the marking rectangle into the first pair of Row and Column

values (labeled Top Corner on the Extract action window) and the row and column coordinates of the lower right corner into the second pair of Row and Column values (labeled Bottom Corner).

If you enter the Row and Column values yourself, then type the first set of row and column coordinates into the first pair of Row and Column values (labeled Top Corner on the Extract action window) and type the second set of coordinates into the second pair of Row and Column values (labeled Bottom Corner). You can use the text cursor on the session window as an aid to determine the coordinates that you want (see "Using the session window's text cursor" on page 129).

In the Row (Bottom Corner) input field you can enter `-1` to signify the last row of the data area on the session window. This feature is helpful if your users work with session windows of different heights (such as 25, 43, 50) and you want to capture data down to the last row. Similarly for the Column (Bottom Corner) input field you can enter `-1` to signify the last column of the data on the session window (see "Significance of a negative value for a row or column" on page 39).

**Assign Return Code to a Variable:** If you want to verify that the Print Extract action is successful, then click Assign Return Code to a Variable and select a variable to hold the return code from the Print Extract action.

### Print End

The Print End action terminates the current print bean if one exists. If a current print bean does not exist then the action has no effect.

**Assign Return Code to a Variable:** If you want to verify that the Print End action is successful, then click Assign Return Code to a Variable and select a variable to hold the return code from the Print End action.

## Prompt action (<prompt> element)

The Prompt action provides a powerful way to send immediate user keyboard input into the 3270 or 5250 application or into a variable.

The Prompt action displays on top of the session window a prompt window that contains a message, an input field, and an OK button. After the user types text into the input field and clicks OK, the Prompt action uses the input in one or both of the following ways:

- The Prompt action types the input into an input field of the session window.
- The Prompt action interprets the input as a string and stores the input into a variable.

A typical use of this action, but by no means the only use, is to permit the user to provide a password. Many scenarios require that a macro log on to a host or start an application that requires a password for access. Because a password is sensitive data and also typically changes from time to time, you probably do not want to code the password as an immediate value into the macro.

With the Prompt action, you can display a message that prompts the user for a password and that lets the user type the password into the input field. After the user clicks OK, the macro runtime types the input into the session window at the row and column location that you specify. The input sequence can include action keys such as [enterreset], so that if the user types `MyPassword[enterreset]` the macro runtime not only can type the password into the password field but also can

type the key that completes the logon or access action. (Or, you can put the action key into an Input action that immediately follows the Prompt action.)

## Displaying the prompt window

**Parts of the prompt window:** You should type the prompt text (such as `'Please type your password:'`) into the Prompt Name field, not into the Prompt Text field. (The Prompt Text field is an optional field than you can use to store a note containing details about the particular Prompt action.)

The macro runtime displays a prompt window with the following characteristics:

- The prompt window appears on top of the session window and is located in the center of the system's desktop window.
- The caption of the prompt window is always Prompt.
- The message that you typed into the Prompt Name field is displayed in the center of the prompt window, followed by an input field.
- A button row across the bottom of the prompt window contains three buttons:
  - The OK button causes the macro runtime to process the contents of the input field.
  - The Cancel button halts the macro.
  - The Help button displays help text explaining how to use the prompt window.

**Default Response:** In the Default Response field, which is optional, you can type the text of a default response that you want to appear in the input field of the prompt window when the prompt window is displayed. If the user does not type any keyboard input into the input field of the prompt window, but rather just clicks OK to indicate that input is complete, then the macro runtime processes the default response that is contained in the input field.

For example, if the user normally uses ApplicationA but sometimes uses ApplicationB, you could type `ApplicationA` into the Default Response field. When the macro runtime performs the Prompt action, the prompt window appears with the text `ApplicationA` already displayed in the input field. The user either can click OK (in which case the macro processes `ApplicationA` as the contents of the input field) or else type ApplicationB into the input field and then click OK (in which case the macro processes `ApplicationB` as the contents of the input field).

**Password Response:** If you set the Password Response listbox to true (the default is false) then when the user types each key into the input field of the prompt window, the macro runtime displays an asterisk (*) instead of the character associated with the key.

For example, with the Password Response listbox set to true (or resolving to true at runtime) then if the user types 'Romeo' the macro runtime displays ***** in the input field.

## Processing the contents of the input field

**Response Length:** The value in the Response Length field specifies not the size of the input field, but the number of characters that the macro runtime allows the user to type into the input field.

For example, if you set the Response Length field to 10, then the macro runtime allows the user to type only 10 characters into the input field.

**Action keys and Translate Host Action Keys:** Both you (in the Default Response input field) and the user (in the input field of the Prompt window) can use action keys (such as [enterreset], [copy], and so on) as you would in the String field of an Input action (see "Input string" on page 83).

The Translate Host Action Keys listbox and its effect are exactly like the Translate Host Action Keys listbox in the Input action (see "Translate Host Action Keys" on page 83). If you set this listbox to true, which is the default value, then the macro runtime interprets an action key string (such as [copy]) as an action key rather than as a literal string.

## Handling the input sequence in the session window

Use the Row and Column fields to specify the row and column position on the session window at which you want the macro runtime to start typing the input sequence. To have the macro runtime start typing the input sequence at the current position of the text cursor, you can set either or both of the Row and Column fields to 0. As with the Input action, the row and column position must lie within a 3270 or 5250 input field at runtime, or else the session window responds by inhibiting the input and displaying an error symbol in the Operator Information Area, just as it responds to keyboard input from an actual user.

You can have the macro runtime clear the contents of the input field before typing begins, by setting the Clear Host Field listbox to true.

The Move Cursor to End of Input field has the same function and effects as the button of the same name in the Input action (see "Move Cursor to End of Input" on page 83).

You can have the macro runtime not display the input sequence in the input field by setting the Don't Write to Screen listbox to true. This field is enabled only when the Assign to a Variable checkbox is selected.

## Assigning the input sequence to a variable

You can have macro runtime store the input sequence into a variable by selecting the Assign to a Variable listbox.

In the popup window for specifying a new variable, you can specify the name of a variable that the current macro inherits from another macro, or you can specify the name of a new variable that you want to create in the current macro. If you want to create a new variable in the current macro, select the Create variable in this macro checkbox and select the type of the new variable.

The macro runtime stores the input sequence as a string, and consequently you might want to specify a string variable as the variable to receive the input. However, if the variable is of some other type than string, then the macro runtime will try to convert the input to the data type of the target variable according to the usual rules (see "Automatic data type conversion" on page 37).

## The promptall attributes

You can have the macro runtime combine the popup windows from all <prompt> elements into one large prompt window and display this large prompt window at the beginning of the macro playback, by setting the **promptall** attribute of the <HAScript> element to true (see "<HAScript> element" on page 154).

The **promptall** attribute in the <actions> element works similarly (see "<actions> element" on page 144).

# Run program action (<runprogram> element)

The Run program action launches a native application and optionally waits for it to terminate. You can provide input parameters for the application and store the return code in a variable.

You can launch any application that can be run by the system runtime.

The Run program action has many uses, such as the following:

- Launching a native application that prepares data that the macro needs, or that uses data that the macro has prepared.
- Launching a native application that prepares the workstation (for example, by making a network connection) for an action that the macro is about to initiate.
- Launching a native application that detects the status of a system condition and then reports the status back to the macro.

## Launching the native application

You should specify in the Program input field the complete path and name of the file that launches the native application, for example:

```
'c:\\Program Files\\MyApp\\bin\\myapp.exe'
```

You should notice in the example above that a single backslash character (\)is represented by two backslash characters (\\). The reason is that in the advanced macro format the backslash is a special character and therefore must be represented by a backslash + the character itself (see "In the advanced macro format, rules for representation of strings, etc." on page 32).

You should specify in the Parameters field any parameters that should be passed to the native application.

## Waiting for the native application to terminate

If you want the macro runtime to wait until the native application has terminated, set the Wait for Program listbox to true. The default is false (the macro runtime does not wait).

## Capturing the return code

You can capture the status code returned by the native application in a variable by selecting the Assign Exit Code to Variable checkbox and specifying the name of a variable.

## Example of launching a native application

The following example launches a native application, waits for it to terminate, and then displays the return code from the application in a message window. This example uses the following action elements: Run program action, Message action.

You can copy the text of this macro script from this document into the system clipboard, and then from the system clipboard into the Code Editor (see "Copy and paste a script from this guide into the Code Editor" on page 130). After you save this script in the Macro Editor, you can edit it either with the Macro Editor or with the Code Editor.

You should notice the following facts about this example:

- The example consists of one entire macro script named RUN PROGRAM.
- The following actions occur in the <actions> element:
  - The <runprogram> element launches a native application, waits for it to return, and stores the return code into $intReturn$.

– A message action displays the value in $intReturn$ in a message window.

```
<HAScript name="g1" description=" " timeout="60000" pausetime="300"
            promptall="true" author="" creationdate="" supressclearevents="false"
            usevars="true" ignorepauseforenhancedtn="false"
            delayifnotenhancedtn="0">
   <vars>
      <create name="$intReturn$" type="integer" value="0" />
   </vars>
   <screen name="Screen1" entryscreen="true" exitscreen="true" transient="false">
      <description>
         <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
      </description>
      <actions>
         <runprogram exe=
                  "'c:\\Program Files\\Windows NT\\Accessories\\Wordpad.exe'"
                  param="'c:\\tm\\new_file.doc'" wait="true"
                  assignexitvalue="$intReturn$" />
         <message title="" value="'Return value is '+$intReturn$" />
      </actions>
      <nextscreens timeout="0" >
      </nextscreens>
   </screen>
</HAScript>
```

*Figure 35. Sample code RUN PROGRAM*

## Trace action (<trace> element)

The Trace action sends a trace message to a trace destination that you specify, such as the Java console.

Use the Trace Handler listbox to specify the trace destination to which you want the trace message sent:

- Select Host On-Demand trace facility to send the trace message to the Host On-Demand Trace Facility.
- Select User trace event to send the trace message to a user trace handler.
- Select Command line to send the trace message to the Java console.

Use the Trace Text input field to specify the string that you want to send to the trace destination.

### Example

The following example shows how to send trace messages to the Java console. This example uses the following action elements: Trace and Variable update.

You can copy the text of this macro script from this document into the system clipboard, and then from the system clipboard into the Code Editor (see "Copy and paste a script from this guide into the Code Editor" on page 130). After you save this script in the Macro Editor, you can edit it either with the Macro Editor or with the Code Editor.

You should notice the following facts about this example:

- The example consists of one entire macro script named TRACE.
- The <create> element creates a string variable named `$strData$` and initializes it to an original value of `'Original value'`.

- The first action is a Trace action with the Trace Text set to 'The value is' + $strData$.
- The second action is a Variable update action that sets the variable $strData$ to a new value of 'Updated value'.
- The third action is another Trace action identical with the first Trace action.

```
<HAScript name="TRACE" description=" " timeout="60000" pausetime="300"
            promptall="true" author="" creationdate="" supressclearevents="false"
             usevars="true" ignorepauseforenhancedtn="false"
             delayifnotenhancedtn="0">
   <vars>
      <create name="$strData$" type="string" value="'Original value'" />
   </vars>
   <screen name="Screen1" entryscreen="true" exitscreen="false" transient="false">
      <description>
          <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
      </description>
      <actions>
         <trace type="SYSOUT" value="'The value is '+$strData$"  />
         <varupdate name="$strData$" value="'Updated value'" />
         <trace type="SYSOUT" value="'The value is '+$strData$"  />
      </actions>
      <nextscreens timeout="0" >
      </nextscreens>
  </screen>
```

*Figure 36. Sample code TRACE*

This script causes the macro runtime to send the following data to the Java console:

```
The value is +{$strData$ = Original value}
The value is +{$strData$ = Updated value}
```

In the trace output above you should notice that instead of just displaying the value of $strData$, the Debug action displays both the variable's name and its value inside curly braces {}.

### User trace event

To take advantage of the User trace event setting, you need the separate Host Access Toolkit product. You should implement the MacroRuntimeListener interface. The macro runtime sends an event to MacroRuntimeListeners for the following types of occurrences:

- Macro error
- Macro state change
- Trace action. The event is a MacroTraceEvent.
- Prompt action.
- Message action.
- Extract action.

## Variable update action (<varupdate> element)

The <varupdate> element stores a value into a variable. You must specify:

- The name of a variable
- The value that you want to store into the variable.

During macro playback the macro runtime performs the Variable update action by storing the specified value into the specified variable.

You can also use the Variable update action in a <description> element (see "Processing a Variable update action in a description" on page 67).

The value can be an arithmetic expression and can contain variables and calls to imported methods. If the value is an expression, then during macro playback the macro runtime evaluates the expression and stores the resulting value into the specified variable.

The Variable update action works like an assignment statement in a programming language. In a Java program you could write assignment statements such as:

```
boolVisitedThisScreen = true;
intVisitCount = intVisitCount + 1;
dblLength = 32.4;
strAddress ="123 Hampton Court";
```

With the Variable update action you type the left side of the equation (the variable) into the Name field on the Variable Update window and type the right side of the equation (the value) into the Value field on the same window. To create the equivalents of the Java assignment statements above, you would write:

*Table 16. Examples of variable names and values*

| **In the Name input field:** | **In the Value input field:** |
|---|---|
| $boolVisitedThisScreen$ | true |
| $intVisitCount$ | $intVisitCount$+1 |
| $dblLength$ | 32.4 |
| $strAddress$ | '123 Hampton Court' |

The value that you provide must belong to the correct data type for the context or be convertible to that type (see "Automatic data type conversion" on page 37).

The great usefulness of the Variable update action is due to the facts that:

- The entity in the Value field can be an expression, and
- Expressions are not evaluated until the action is performed.

For more information on expressions see Chapter 5, "Data types, operators, and expressions", on page 31.

## Variable update action with a field variable

Using a Variable update action to update a field variable is a convenient way of reading the contents of a 3270 or 5250 field in the session window and storing the field's contents as a string into a variable.

A field variable is a type of string variable. A field variable contains a string, just as a string variable does, and you can use a field variable in any context in which a string variable is valid. However, a field variable differs from a string variable in the way in which a string is stored into the field variable. The string that a field variable contains is always a string that the macro runtime has read from a 3270 or 5250 field in the current session window.

When you use the Variable update action to update a string variable, you specify the following information in the Variable update window:

- The name of the field variable, such as $fldTmp$.
- A location string, such as '5,11'. (A location string is a string containing two integers separated by a comma that represent a row and column location on the session window.)

When the macro runtime performs the Variable update action, the macro runtime:

1. Recognizes that the variable is a field variable.
2. Looks at the location string that is to be used to update the field variable.
3. Finds in the current session window the row and column location specified by the location string.
4. Finds in the current session window the 3270 or 5250 field in which the row and column location occurs.
5. Reads the entire contents of the 3270 or 5250 field, including any leading and trailing blanks.
6. Stores the entire contents of the field as a string into the field variable.

You can then use the field variable in any context in which a string is valid. For example, you can concatenate the field variable with another string, as in the following:

```
'The field\'s contents are'+ $fldPrintOption$
```

As an example, suppose that the session window contains a 3270 or 5250 field with the following characteristics:

- It begins at row 5, column 8.
- It ends at row 5, column 32.
- It contains the string `'Print VTOC information'`.

You set up a Variable update action with the following values:

- In the Name field of the Variable update window you type the name of a field variable that you have just created, `$fldData$`.
- In the Value field you type a location string, `'5,11'`. Notice that you have to specify only one row and column location, and that it can be any row and column location that lies within the field.

When the macro runtime performs this Variable update action, the macro runtime reads the entire contents of the field and stores the contents as a string into $fldData$. The field variable $fldData$ now contains the string `'Print VTOC information'`.

**Reading part of a field:** When you are using a field variable in a Variable update action, you can specify a location string that contains two locations. Use this feature if you want to read only part of the contents of a field.

Type the first and second locations into the Value field with a colon (:) between them. For example, if the first location is 5,14 and the second location is 5,17, then you would type `'5,14:5,17'`.

When you specify two locations:

- The first location specifies the first position in the field to read from.
- The second location specifies the last position in the field to read from.

As an example, suppose that the session window contains a 3270 or 5250 field with the following characteristics:

- It begins at row 5, column 8.
- It ends at row 5, column 32.
- It contains the string `'Print VTOC information'`.

and suppose that you set up a Variable update action with the following values:

- In the Name field of the Variable update window you type the name of a field variable that you have just created, `$fldData$`.
- In the Value field you type a location string, `'5,14:5,17'`. Here you are specifying both a beginning location and an ending location within the field.

When the macro runtime performs this Variable update action, the macro runtime reads the string `'VTOC'` from the field (beginning at the position specified by the first location string and continuing until the position specified by the second location string) and stores the string `'VTOC'` into $fldData$.

If the second location lies beyond the end of the field, the macro runtime reads the string beginning at the first location and continuing until the end of the field. The macro runtime then stores this string into the field variable.

## Xfer action (<filexfer> element)

The Xfer action (pronounced ″transfer action″ or ″file transfer action″) transfers a file from the workstation to the host or from the host to the workstation.

### Basic parameters

In the Tranfer Direction listbox you must specify whether you want the file to go from the workstation to the host (Send) or from the host to the workstation (Receive). If you select Expression, then you must specify an expression (for example, a variable named `$strDirection$`) that at runtime resolves to either `Send` or `Receive`.

Table 17 shows the values that you should use in the Host-File Name field and the PC-File Name field:

*Table 17. Host-File Name field and PC-File Name field*

| Transfer Direction: | Host-File Name field: | PC-File Name field: |
|---|---|---|
| Send | The name that you want assigned to the file when it reaches the host. For example, in a 3270 Display session, `'trace1 txt a'` | The name of the file that you want to send to the host. For example, `'e:\\tm\\trace1.txt'`. |
| Receive | The name of the file that you want to receive at the workstation. For example, `'january archive a'` | The name that you want assigned to the file after it reaches the client. For example, `'d:\\MyData\\january.arc'` |

Remember that if you are using the advanced macro format the backslash \ is a special character that must be written `'\\'`(see “In the advanced macro format, rules for representation of strings, etc.” on page 32).

### Advanced parameters

In the Clear Before Transfer field, in most cases you should use `true` for 3270 Display sessions and `false` for 5250 Display sessions.

You should set the Timeout field to the number of milliseconds that you want the macro runtime to wait before terminating the transfer. The default is 10000

milliseconds (10 seconds). This Timeout field saves the user from the situation in which the macro hangs because it is trying to transfer a file over a session that has suddenly been disconnected. You might need to use a greater value for very long files or if your connection is slow.

You should use the Options field for any additional parameters that your host requires. These parameters are different for each type of host system.

You should use the PC Code-page field to select the code page (mapping table) that you want the macro runtime to use in translating characters from the workstation's character set to the host's character set and vice versa. You should select the same code-page number (such as 437) that is specified in the session configuration.

## Parameters for BIDI sessions (Arabic or Hebrew)

There are additional parameters for BIDI sessions (Arabic or Hebrew) that you can set with the Code Editor ("Attributes" on page 153 ).

## Examples

The following example shows an Xfer action:

```
<actions>
   <filexfer direction="send" pcfile="'c:\\myfile.txt'" hostfile="'myfile text A0'"
            clear="true" timeout="10000" pccodepage="437"  />
</actions>
```

*Figure 37. Example of the Xfer action*

# Chapter 9. Screen Recognition, Part 2

## Valid next screens

As you may remember from Chapter 6, “How the macro runtime processes a macro screen”, on page 41, the macro runtime typically finds the names of macro screens that are candidates for becoming the next macro screen to be processed by looking in the <nextscreens> element of the current macro screen. That is, the macro screen contains within itself a list of the macro screens that can validly be processed next. (Entry screens and transient screens are exceptions, see “Entry screens, exit screens, and transient screens” on page 105.)

In the Macro Editor, the Links tab provides the user interface for storing the names of candidate macro screens into the <nextscreens> element of a macro screen. Figure 38 show a sample Links tab:

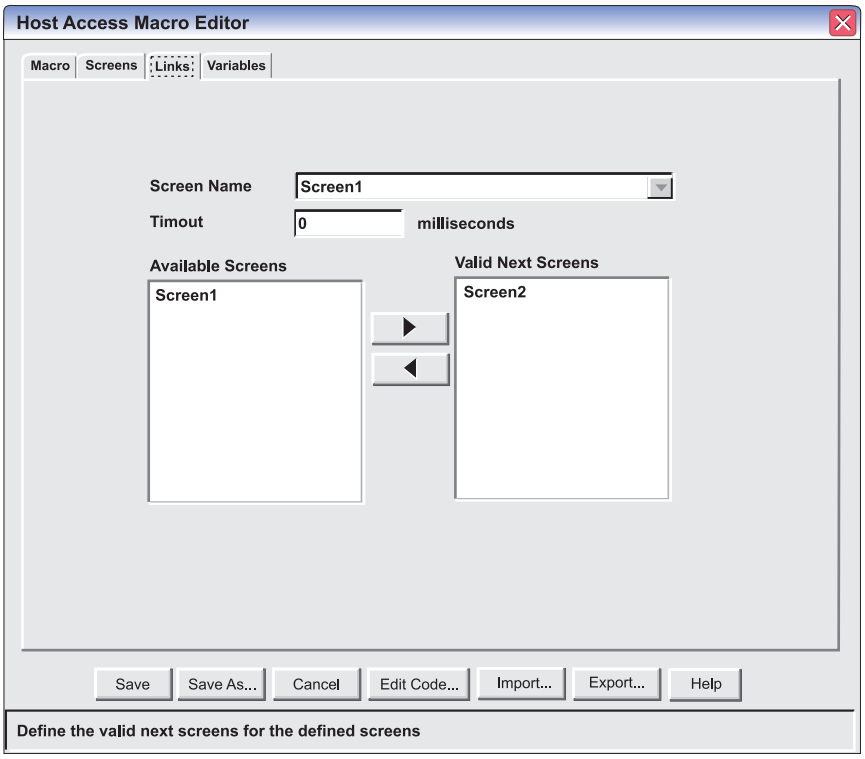


*Figure 38. Sample Links tab*

In the figure above, the Screen Name listbox at the top of the tab contains a list of all the macro screens in the entire macro. The currently selected macro screen is Screen1. On the right, the Valid Next Screens listbox contains a list of candidate macro screens for Screen1 (Do not confuse this listbox, which contains the names in the <nextscreens> element of Screen1, with the list of valid next screens that the macro runtime uses when a macro is played back). On the left, the Available Screens listbox contains a list of the names of all other macro screens .

Although the figure above shows only one screen in the Available Screens list, that is because this figure is from a macro with only two macro screens in it, Screen1

© Copyright IBM Corp. 2003

and Screen2. Instead, imagine a macro of twenty screens, and suppose that you want to add macro screens to the <nextscreens> list of a new macro screen, ScreenR. You would follow these steps:

1. On the Links tab, expand the Screen Name listbox and scroll down until you find ScreenR.
2. Select ScreenR.
3. Because ScreenR is a new screen, there are no macro screen names listed in the Valid Next Screens list on the right.
4. On the left, the Available Next Screens listbox contains the names of all the macro screens in the macro.
5. Select a screen that you want to add to the list for ScreenR. Suppose that you select ScreenS.
6. After selecting ScreenS, click the right arrowhead button between the two listboxes. ScreenS is added to the listbox on the right, and removed from the listbox on the left.
7. In the same way, move the names of any other macro screens that you want to the Valid Next Screens listbox for ScreenR.
8. Suppose that you move a total of three screen names: ScreenS, ScreenG, and ScreenY.

When you are done, ScreenR, the currently selected macro screen, has the names of three macro screens in its list of valid next screens.

In the Code Editor, you would see the names of the valid next macro screens, ScreenS, ScreenG, ScreenY, stored inside ScreenR as shown in Figure 39:

```
<screen name="ScreenR" entryscreen="true" exitscreen="false" transient="false">
   <description>
      ...
   </description>
   <actions>
      ...
   </actions>
   <nextscreens>
      <nextscreen name="ScreenS"/>
      <nextscreen name="ScreenG"/>
      <nextscreen name="ScreenY"/>
   </nextscreens>
</screen>
```

*Figure 39. Macro screen ScreenR with <nextscreens> element*

The figure above shows the <screen> element for ScreenR, with the **name** attribute set to `"ScreenR"`. Inside are the three primary structural elements of a <screen> element: the <description> element, the <actions> element, and the <nextscreens> element. The contents of the <description> element and the <actions> element are not shown but are indicated with an ellipsis (...). The <nextscreens> element contains three <nextscreen> elements, and each <nextscreen> element contains the name of one of the valid next screens: `ScreenS`, `ScreenG`, and `ScreenY`.

For more information about runtime processing see Chapter 6, “How the macro runtime processes a macro screen”, on page 41.

# Entry screens, exit screens, and transient screens

You can use the entry screen, exit screen, and transient screen settings to mark macro screens that you want the macro runtime to treat in a special way. In the Macro Editor, you make these settings on the General tab of the Screens tab. At the top of the tab, under the Screen Name field, are listboxes for Entry Screen, Exit Screen, and Transient Screen. For each of these listboxes, you must specify a boolean value (the default is false) or an expression that evaluates to a boolean value.

In the Code Editor, these settings appear as attributes of the <screen> element. In Figure 39 on page 104, above, you can see these three attributes in the <screen> element for ScreenR: **entryscreen**, **exitscreen**, and **transient**.

## Entry screens

Set Entry Screen to true if you want the macro screen to be considered as one of the first macro screens to be processed when the macro is played back. You might have only one macro screen that you mark as a entry screen, or you might have several.

When the macro playback begins, the macro runtime searches through the macro script and finds all the macro screens that are designated as entry screens. Then the macro runtime adds the names of these entry macro screens to the runtime list of valid next screens. Finally the macro runtime tries in the usual way to match one of the screens on the list to the current session window.

When the macro runtime has matched one of the entry macro screens to the session window, that macro screen becomes the first macro screen to be processed. Before performing the actions in the first macro screen, the macro runtime removes the names of the entry macro screens from the runtime list of valid next screens.

### Macro with several entry screens

One of the situations in which you might have several entry screens in the same macro is when a host application begins with a series of application screens one after another, such as application screen A, followed by application screen B, followed by application screen C. For instance, screen A might be a logon screen, screen B a screen that starts several supporting processes, and screen C the first actual screen of the application.

In this situation, you might want the user to be able to run the macro whether the user was at application screen A, B, or C.

### Entry screen can also be a normal screen

If you mark a screen as an entry screen, it can still participate in the macro as a normal screen and be listed in the <nextscreens> lists of other macro screens.

For example, you might have a host application that has a central application screen with a set of menu selections, so that each time you make a menu selection the application goes through several application screens of processing and then returns to the original central application screen.

In this situation, suppose that macro ScreenA is the macro screen corresponding to the central application screen. Therefore:

- ScreenA could be an entry screen, because the macro could start at the central application screen.

- ScreenA could also appear in the <nextscreens> element of other macro screens, because after each task the application returns to the central application screen.

## Exit screens

Setting Exit Screen to true for a macro screen causes the macro runtime to terminate the macro after it has performed the actions for that macro screen. That is, after the macro runtime performs the actions, and before going on to screen recognition, the macro runtime looks to see if the current macro screen has the exit screen indicator set to true. If so, then the macro runtime terminates the macro. (The macro runtime ignores the <nextscreens> element of an exit screen.)

Therefore you would set Exit Screen to true for a macro screen if you wanted the macro screen to be a termination point for the macro.

You can have any number of exit screens for a macro. Here are some examples of situations in which there could be several exit screens.

- A macro might have one normal termination point and several abnormal termination points, which could be reached if an error occurred.
- A macro might allow you to stop at a certain point in the processing, or to keep going, so that there would be several normal termination points.

## Transient screens

A transient macro screen is used to process an application screen that has the following characteristics:

- The application screen occurs unpredictably during the flow of the application. It might occur at several points or it might not occur at all.
- The only action that needs to occur for the application screen is that it needs to be cleared.

An example of such an application screen is an error screen that the application displays when the user enters invalid data. This error screen appears at unpredictable times (whenever the user enters invalid data) and as a macro developer the only action that you want to take for this error screen is to clear it and to get the macro back on track.

When the macro runtime prepares to play back a macro, at the point where the macro runtime adds the names of entry screens to the runtime list of valid next screens, the macro runtime also adds the names of all macro screens marked as transient screens (if any) to the end of the list.

The names of these transient screens remain on the runtime list of valid next screens throughout the entire macro playback. Whenever the macro runtime adds the names of new candidate macro screens (from the <nextscreens> element of the current macro screen) to the list, the macro runtime adds these new candidate names ahead of the names of the transient screens, so that the names of the transient screens are always at the end of the list.

Whenever the macro runtime performs screen recognition, it evaluates the macro screens of all the names on the list in the usual way. If the macro runtime does not find a match to the application screen among the candidate macro screens whose names are on the list, then the macro runtime goes on down the list trying to match one of the transient macro screens named on the list to the application screen.

If the macro runtime matches one of the transient macro screens to the current application screen, then the macro runtime does not remove any names from the list. Instead, the macro runtime performs the actions in the transient macro screen (which should clear the unexpected application screen) and then goes back to the screen recognition process that it was pursuing when the unexpected application screen occurred.

### Example of handling of transient screen

Suppose that the macro runtime is doing screen recognition and that it has the names of three macro screens on the list of valid next screens: ScreenB and ScreenD, which are the names of candidate screens, and ScreenR, which is the name of a transient screen. The macro runtime performs the following steps:

1. When the session window's presentation space is updated, the macro runtime evaluates the names on the list of valid next screens in the usual way.
2. Suppose that an unexpected application screen has occurred, so that neither ScreenB nor ScreenD matches the current application screen, but that ScreenR does match the current application screen.
3. Because a transient screen has been recognized, the macro runtime does not remove any names from the list of valid next screens.
4. The macro runtime makes ScreenR the current macro screen to be processed.
5. The macro runtime performs the actions in ScreenR. These actions clear the unexpected application screen.
6. The macro runtime ignores the <nextscreens> element, if any, in ScreenR.
7. The macro runtime returns to the previous task of screen recognition in step 1 above. The list of valid next screens has not changed. This time, suppose that an expected application screen is displayed and that the macro runtime finds that ScreenD matches it. Therefore:
   a. The macro runtime makes ScreenD the next macro screen to be processed.
   b. The macro runtime removes the names ScreenB and ScreenD from the list of valid next screens. The name ScreenR remains on the list.
   c. The macro runtime begins processing the actions in ScreenD.

## Timeout settings for screen recognition

This section discusses the scenario in which the macro runtime cannot advance because it cannot match a screen on the list of valid next screens to the current application screen. There are two fields that let you set a timeout value that terminates the macro if screen recognition does not succeed before the timeout expires:

- Timeout Between Screens field on the Macro tab.
- Timeout field on the Links tab.

### Screen recognition

As you know, after the macro runtime has performed all the actions in the <actions> element of a macro screen, then the macro runtime attempts to match one of the screens on the list of valid next screens to the new application screen (see Chapter 6, "How the macro runtime processes a macro screen", on page 41).

Sometimes, unforeseen circumstances make it impossible for the macro runtime to match any of the macro screens on the list of valid next screens to the application screen. For example, a user might type an input sequence that takes him to an application screen unforeseen by the macro developer. Or, a systems programmer

might have changed the application screen so that it no longer matches the description in the <description> element of the corresponding macro screen.

When such a scenario occurs, the result is that the macro appears to hang while the macro runtime continually and unsuccessfully attempts to find a match.

## Timeout Between Screens (Macro tab)

The Timeout Between Screens checkbox and input field are located on the Macro tab and specify a timeout value for screen recognition. By default, if the checkbox is enabled, this value applies to each and every macro screen in the macro. However, you can change the value for a particular macro screen by using the Timeout field on the Links tab (see the next section).

Whenever the macro runtime starts to perform screen recognition, it checks to determine whether the Timeout Between Screens value is set for the entire macro and whether a Timeout value is set for the macro screen. If a timeout value is set, then the macro runtime sets a timer to the number of milliseconds specified by the timeout value. If the timer expires before the macro runtime has completed screen recognition, then the macro runtime terminates the macro and displays a message such as the following:

```
Macro timed out: (Macro=ispf_ex2, Screen=screen_address_type)
```

*Figure 40. Error message for screen recognition timeout*

Please notice that this message displays the name of the macro and the name of the screen that was being processed when the timeout occurred. For example, if the screen specified in this message is ScreenA, then the macro runtime had already performed all the actions in ScreenA and was trying to match a macro screen in the Valid Next Screens list for ScreenA to the application screen.

To use the Timeout Between Screens field, select the checkbox and type a value for the number of milliseconds to wait before terminating the macro. By default the checkbox is checked and the timeout value is set to 60000 milliseconds (60 seconds).

## Timeout (Links tab)

The Timeout input field on the Links tab specifies a timeout value for screen recognition for a particular macro screen. If this value is non-0, then the macro runtime uses this value as a timeout value (in milliseconds) for screen recognition for this macro screen, instead of using the value set in the Timeout Between Screens field on the Macro tab.

If the timer expires before the macro runtime has completed screen recognition, then the macro runtime displays the message in Figure 40.

# Recognition limit (General tab of the Screens tab)

The recognition limit is not an attribute in the begin tag of the <screen> element but rather a separate element (the <recolimit> element) that optionally can occur inside a <screen> element, on the same level as the <description>, <actions>, and <nextscreens> elements.

The Set Recognition Limit checkbox and the Screens Before Error input field are located on the General tab of the Screens tab (see Figure 15 on page 26). By default the Set Recognition limit checkbox is cleared and the input field is disabled. If you select the checkbox, then the Macro Editor sets the default value of the Screens Before Error input field to 100. You can set the value to a larger or smaller quantity.

The recognition limit allows you to take some sort of action if the macro runtime processes a particular macro screen too many times. If the macro runtime does process the same macro screen a large number of times (such as 100), then the reason is probably that an error has occurred in the macro and that the macro is stuck in an endless loop.

When the recognition limit is reached, the macro runtime either terminates the macro with an error message (this is the default action) or starts processing another macro screen that you specify.

You should notice that the recognition limit applies to one particular screen and that by default it is absent. You can specify a recognition limit for any macro screen, and you can specify the same or a different recognition limit value for each macro screen in which you include it.

## Determining when the recognition limit is reached

The macro runtime keeps a recognition count for every macro screen that includes a <recolimit> element. When macro playback begins the recognition count is 0 for all macro screens.

Suppose that a macro includes a macro screen named ScreenB and that ScreenB contains a <recolimit> element with a recognition limit of 100. Each time the macro runtime recognizes ScreenB (that is, each time the macro runtime selects ScreenB as the next macro screen to be processed), the macro runtime performs the following steps:

1. The macro runtime detects the presence of the <recolimit> element inside ScreenB.
2. The macro runtime increments the recognition count for ScreenB.
3. The macro runtime compares the recognition count with the recognition limit.
4. If the recognition count is less than the recognition limit, then the macro runtime starts performing the action elements of ScreenB as usual.
5. However, if the recognition count is greater than or equal to the recognition limit, then the macro runtime performs the action specified by the <recolimit> element. In this case macro runtime does not process any of the action elements in ScreenB.

## Action when the Recognition limit is reached

The default action when the recognition limit is reached is that the macro runtime displays an error message such as the following and then terminates the macro:

```
Recolimit reached, but goto screen not provided, macro terminating.
```

If you want the macro runtime, as a recognition limit action, to go to another macro screen, then you must use the Code Editor to add a **goto** attribute to the <recolimit> element and specify the name of the target macro screen as the value of the attribute (see "<recolimit> element" on page 166).

If you use the **goto** attribute, the macro runtime does not terminate the macro but instead starts processing the macro screen specified in the attribute.

You can use the target macro screen for any purpose. Some possible uses are:

- For debugging.
- To display an informative message to the user before terminating the macro.
- To continue processing the macro.

# Chapter 10. Actions, Part 2: Timing issues

This chapter describes several timing issues involved in processing actions and the resources available for dealing with these issues.

## Pause after an action

This section discusses the scenario in which an action does not perform as expected because a previous action has side effects that have not completed.

There are two settings that let you add a pause after actions during runtime:

- Pause Between Actions on the Macro tab
- Set Pause Time on the General tab of the Screens tab

### Speed of processing actions

Because the macro runtime executes actions much more quickly than a human user does, unforeseen problems can occur during macro playback that cause an action not to perform as expected, because of a dependency on a previous action.

One example is a keystroke that causes the application screen to change. If a subsequent action expects the application screen to have already changed, but in fact the application screen is still in the process of being updated, then the subsequent action can fail.

Timing-dependent errors between actions can occur in many other situations, if the macro runtime performs each action immediately after the preceding action.

### Pause Between Actions (Macro tab)

The Pause Between Actions field on the Macro tab causes the macro runtime to wait a specified number of milliseconds after every action in the entire macro. That is, after the macro runtime performs an action, the macro runtime checks the Pause Between Actions setting to see if it is enabled. If so, then macro runtime waits the specified number of milliseconds, then goes on to perform the next action.

By default this checkbox is enabled and the timeout value is set to 300 milliseconds. Therefore the macro runtime will wait for 300 millliseconds after every action that it performs.

Notice that this wait is added after every action of every macro screen. Therefore this one setting allows you avoid this type of problem without having to change each macro screen that might have a problem.

### Set Pause Time (General tab of the Screens tab)

If you want a longer or shorter pause time between actions for a particular macro screen, or if you have only a few macro screens in which the wait interval between actions is important, then you can use the Set Pause Time setting on the General tab of the Screens tab.

By default this checkbox is disabled.

© Copyright IBM Corp. 2003

If you enable this setting, then the macro runtime waits for the specified number of milliseconds after each action in this particular macro screen.

For example, if for ScreenA you select the Set Pause Time checkbox and set the value to 500 milliseconds, then the macro runtime waits 500 milliseconds after each action in ScreenA.

When the macro runtime processes a macro screen with Set Pause Time enabled, it ignores the setting of the Pause Between Actions option on the macro tab, and uses only the value in the Set Pause Time setting.

### Adding a pause after a particular action

If you need a longer pause after one particular action in a macro screen, you can add a Pause action after the action. The wait that you specify in the Pause action is in addition to any wait that occurs because of a Pause Between Actions or a Set Pause Time.

## Screen completion

### Recognizing the next macro screen too soon

Suppose that you have a macro screen, ScreenB, with the following bug: the macro runtime starts processing the actions in ScreenB before the host has completely finished displaying the new application screen. Although this timing peculiarity might not pose a problem for you in most situations, suppose that in this instance the first action in ScreenB is an Extract action that causes the macro runtime to read data from rows 15 and 16 of the application screen. Unfortunately the macro runtime performs this action before the host has had time to write all the new data into rows 15–16.

Analyzing this problem, you verify that:

- The session is a 3270 Display session using the default connectivity, TN3270.
- The following sequence of actions occurs:
  1. In processing the previous macro screen, the macro runtime performs an Input action that causes an enter key to be sent to the host.
  2. The host receives the enter key and sends the first block of commands and data for the new application screen.
  3. The client receives the first block and processes it, thereby updating some parts but not all of the host application screen. In particular, rows 15 and 16 of the application screen have not yet been updated.
  4. Meanwhile the macro runtime has started trying to recognize a valid next macro screen that matches the new application screen.
  5. As a result of the changes in the application screen from the first block of commands and data, the macro runtime recognizes macro ScreenB as the next macro screen to be processed.
  6. The macro runtime performs the first action element in ScreenB, which is an Extract action that reads data from rows 15 and 16 of the application screen.
  7. The client receives a second block of commands and data from the host and processes it, thereby updating other parts of the application screen, including rows 15 and 16.

In short, as a result of this timing problem the macro runtime has read rows 15 and 16 of the new application screen before the host could finish update them.

## The ordinary TN3270 protocol

The reason for this problem is that the unenhanced TN3270 protocol does not include a way for a host to inform a client that the host application screen is complete. (TN3270 implements a screen-oriented protocol, 3270 Data Stream, over a character-oriented connection, Telnet). Therefore, the host cannot send several blocks of data to the client and then say, "OK, the application screen is now complete – you can let the user enter data now." Instead, each block arrives without any indication about whether it is the last block for this application screen. From the client's point of view, something like the following events occur:

1. A block of commands and data arrives. The client sets the input inhibit indicator, processes the block, and displays the new data on the specified parts of the session window. The client then clears the input inhibit indicator and waits.
2. 30 milliseconds pass.
3. Another block of commands and data arrives. The client processes the block as in step 1 above. This block causes a different part of the screen to be updated. The client waits.
4. 50 milliseconds pass.

This process continues until the host has completely displayed a new host application data screen. The client still waits, not knowing that the host application screen is complete. (For more information, see Chapter 6, "How the macro runtime processes a macro screen", on page 41).

This process does not present problems for a human operator, for various reasons that are not important here.

However, this process does present problems for the macro runtime during screen recognition. Recall that during screen recognition the macro runtime tries to match the application screen to one of the valid next macro screens every time the screen is updated and every time an OIA event occurs (see "Re-doing the evaluation" on page 46). Therefore the macro runtime might find a match before the screen is completely updated. For example, a String descriptor might state that recognition occurs if row 3 of the application screen contains the characters "ISPF Primary Option Menu". When the host has updated row 3 to contain these characters, then the macro runtime determines that a match has occurred, regardless of whether the host has finished updating the remainder of the application screen.

## Solutions

There are three approaches to solving this problem:

- Add more descriptors to the description.
- Insert a delay after the Input action that sends an enter key (see step 1 in "Recognizing the next macro screen too soon" on page 112).
- Use the contention-resolution feature of TN3270E.

The following subsections describe these solutions.

### Add more descriptors

This approach works sometimes but can be awkward and unreliable. You add enough descriptors to the description part of ScreenB so that the macro runtime will not recognize the ScreenB until the critical portion of the application screen has been updated.

### Insert a delay after the input action

Inserting a delay is the best solution if the session is an ordinary TN3270 session or if the session is a TN3270E session without contention-resolution. That is, after the Input action (in ScreenA in our example) that causes the host to send a new application screen, insert a pause of several hundred milliseconds or longer. This delay allows enough time for the host to update the application screen before the macro runtime starts processing the actions in the next macro screen (ScreenB).

In this scenario there are several ways to insert a pause after the Input action:

- Increase the Pause Between Actions delay. However, the Pause Between Actions delay is inserted after every action in every macro screen of the macro.
- Increase the Set Pause Time for ScreenA. This method is a good one. You are increasing the pause time after every action in ScreenA, so that only ScreenA is affected.
- Add a Pause action to ScreenA immediately after the Input action. This method is also good. You are inserting a pause exactly where it is needed.
- Add a Pause action as the first action of ScreenB. You might prefer this method in certain scenarios. However, using this method, if there are several macro screens that can occur after ScreenA (such as ScreenB, ScreenC, ScreenD), and if the screen completion problem occurs for each of these following macro screens, then you must to insert a Pause as the first action for each of these following macro screens. It is easier to use the method in the previous bullet and insert a Pause Action in one macro screen, ScreenA.

If your macro has to run both on ordinary TN3270 sessions and also on TN3270E sessions with contention-resolution enabled, the XML macro language has several attributes that can help you. See “Attributes that deal with screen completion”.

### Use the contention-resolution feature of TN3270E

TN3270E (Enhanced) is an enhanced form of the TN3270 protocol that allows users to specify an LU or LU pool to which the session will connect and that also supports the Network Virtual Terminal (NVT) protocol for connecting to servers in ASCII mode (for example, in order to log on to a firewall).

Contention-resolution mode is an optional feature of TN3270E, supported by some but not all TN3270E servers, that solves the client’s problem of not knowing when the host has finished updating the application screen. If the client is running a TN3270E session and is connected to a server that supports contention-resolution, then the macro runtime does not recognize a new macro screen until the host has finished updating the application screen.

In Host On-Demand you can set a 3270 Display session to use TN3270E rather than TN3270 by clicking the appropriate radio button on the Connection configuration window of the 3270 Display session configuration panel.

This panel does not contain an option for setting contention-resolution support, because Host On-Demand detects contention-resolution mode automatically, if the host supports it, when the TN3270E session is started.

## Attributes that deal with screen completion

Host On-Demand has three element attributes that address problems that the macro developer encounters when trying to support a single version of a macro to run on both the following environments:

- A non-contention-resolution environment (the macro is being run by clients connected to a TN3270 server or to a TN3270E server without contention resolution; consequently some macro screens might require a Pause action to allow time for the host to update the application screen).
- A content-resolution enviroment (the macro is being run by clients connected to a TN3270E server with contention resolution; consequently no macro screen requires a Pause action to allow time for for the host to update the application screen).

You will have to add these attributes using the Code Editor.

### ignorepauseforenhancedtn=true/false

The **ignorepauseforenhancedtn** parameter of the <HAScript> element, when set to true, causes the macro runtime to skip Pause actions (<pause> elements) during macro playback if the session is running in a contention-resolution environment. You can use this attribute if you developed a macro to run in a non-contention-resolution environment (you inserted Pause actions) and you now want the macro to also run in a contention-resolution environment without unnecessary delays (you want the Pause actions to be ignored).

With this attribute set to true, the macro runtime processes Pause actions (waits the specified number of milliseconds) in a non-contention-resolution environment but ignores Pause actions in a contention-resolution environment.

Notice, however, that setting this attribute to true causes the macro runtime to skip all Pause actions (<pause> elements) in the macro, not just the pauses that have been inserted in order to time for the application screen to be updated. The next subsection addresses this secondary problem.

### ignorepauseoverrideforenhancedtn=true/false

The **ignorepauseoverrideforenhancedtn** parameter of the <pause> element, when set to true in a particular <pause> element, causes the macro runtime to process that <pause> element (wait for the specified number of milliseconds) even if the **ignorepauseforenhancedtn** attribute is set to true in the <HAScript> element.

Set this attribute to true in a <pause> element if you want the <pause> element always to be performed, not skipped, even in a contention-resolution environment with the **ignorepauseforenhancedtn** attribute set to true in the <HAScript> element.

### delayifnotenhancedtn=(milliseconds)

The **delayifnotenhancedtn** parameter of the <HAScript> element, when set to a non-zero value, causes the macro runtime to automatically pause the specified number of milliseconds whenever the macro runtime receives a notification that the OIA (Operator Information Area) has changed.

You can use this attribute if you developed a macro in a contention-resolution environment (you did not need to insert Pause actions) but you now want the macro to run also in a non-contention-resolution environment (some macro screens might need a Pause action to allow time for the application screen to be completed).

With this attribute set to true, then when the macro is run in a non-contention-resolution environment the macro runtime inserts a pause for the specified number of milliseconds each time it receives a notification that the OIA has changed. For example, if you specify a pause of 200 milliseconds then the macro runtime waits for 200 milliseconds every time the OIA changes.

The cumlative effect of the macro runtime pausing briefly after each notification of a change to the OIA is that the application screen is completed before the macro runtime begins processing the actions of the new macro screen. The macro runtime inserts these extra pauses only when it detects that the session is running in a non-contention-resolution environment.

A limitation of this attribute is that the macro runtime adds these extra pauses during every screen, not just during screens in which screen update is a problem. However, the additional time spent waiting is small. And more importantly, this attribute lets you quickly adapt the macro to a non-contention resolution environment, without having to test individual screens and insert a pause action in each screen with a screen update problem.

# Chapter 11. Variables and imported Java classes

## Introduction to variables and imported types

Variables help you to add programming intelligence to macros. With a variable you can store a value, save a result, keep a count, save a text string, remember an outcome, or do any number of other programming essentials.

You can create a variable that belongs to any of the standard data types (string, integer, double, boolean, and field).

You can also create a variable that belongs to an imported type representing a Java class. You can then create an instance of the class and call a method on the instance. This capability opens the door to the abundant variety of functionality available through Java class libraries, including libraries in the Java Runtime Environment (JRE) libraries, libraries in the Host Access Toolkit product, classes or libraries that you yourself implement, or Java classes and libraries from other sources.

### Advanced macro format required

Using variables requires that you use the advanced macro format for your macro (see "Choosing a macro format" on page 31). Therefore, if you want to add variables to a macro that is in the basic macro format, you must decide whether to convert the macro to the advanced macro format. If you have an old macro in the basic macro format that many users rely on and that works perfectly, you might want to leave the macro as it is.

However, remember that all recorded macros are recorded in the basic macro format. So, if you have recently recorded a macro and are beginning to develop it further, then you might simply not have gotten around to switching to the advanced macro format.

The Macro Editor addresses both these situations by popping up a window with the following message when you start to define a variable in a macro that is still in the basic macro format:

```
You are attempting to use an advanced macro feature. If you choose to continue,
your macro will automatically be converted to advanced macro format. Would you
like to continue?
```

*Figure 41. Reminder message*

Click Yes if you are building a macro in which you plan to use variables, or No if you have a macro in the basic macro format that you do not want to convert.

### Scope of variables

The scope of every variable is global with respect to the macro in which the variable is created. That is, every variable in a macro is accessible from any macro screen in the macro. All that an action or a descriptor in a macro screen has to do to access the variable is just to use the variable name.

© Copyright IBM Corp. 2003

For example, suppose that you have a variable named $intPartsComplete$ that you initialize to 0. You might use the variable in the following ways as the macro proceeds:

1. ScreenC completes Part 1 of a task and increments $intPartsComplete$ using a Variable update action.
2. ScreenG completes Part 2 of a task and increments $intPartsComplete$ using a Variable update action.
3. ScreenM has a Conditional action that tests whether 1 or 2 parts have been completed so far. Depending on the result, the macro expects either ScreenR or ScreenS as the next macro screen to be processed.
4. ScreenS completes Part 3 of a task and increments $intPartsComplete$ using a Variable update action.
5. ScreenZ displays the value of $intPartsComplete$ using a Message action.

In the example above, actions in several different macro screens were able to read from or write to the variable $intPartsComplete$.

## Introduction to the Variables tab

Because a variable belongs to the entire macro, and not to any one screen, it seems appropriate that there is a separate high-level tab for Variables. The Variables tab allows you to:

- Create a variable
- Remove a variable
- Import a Java class as a new variable type

To create a variable belonging to a standard data type, use the Variables tab in the Macro Editor. Figure 42 shows a sample Variables tab:

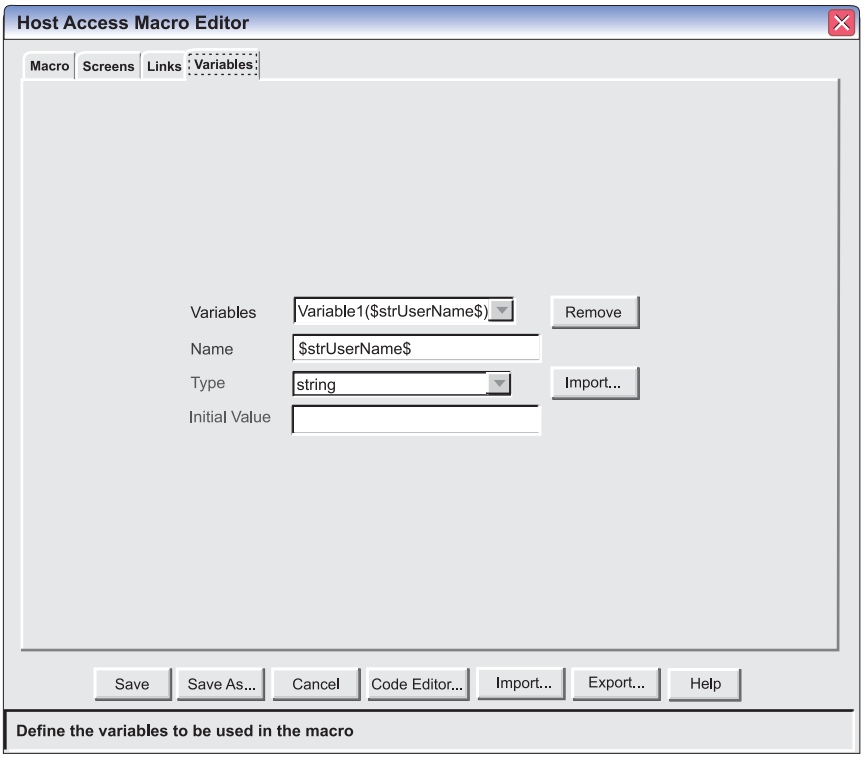


*Figure 42. Variables tab*

In the figure above, the Variables tab of the Macro Editor is selected. The name of the currently selected variable, $strUserName$, is displayed in the Variables listbox. Three other fields contain information that the macro runtime needs to create this variable: the Name input field, the Type listbox, and the Initial Value input field.

The Variables listbox contains the names of all the variables that have been created for this macro. It allows you to select a variable to edit or to remove, and it also contains a <new variable> entry for creating new variables.

Notice that the entry of the currently selected variable is contained in parentheses after another string:

```
Variable1($strUserName$)
```

The string Variable1 is a setting that shows how many variables you have created. It is not saved in the macro script. The real name of the variable is $strUserName$, and you should use this name alone throughout the macro wherever you use the variable.

You have probably noticed that the variable name $strUserName$ is enclosed in dollar signs ($). This is a requirement. You must enclose the variable name in dollar signs ($) wherever you use it in the macro.

The Name input field displays the name of the currently selected variable, $strUserName$. You can change the name of the variable by typing over the old name. Mostly you should use this field only for assigning a name to a newly created variable. Although you can come back later at any time and change the name of this variable (for example to $strUserFirstName$), remember that you might have already used the variable's old name elsewhere in the macro, in some action or descriptor. If you change the name here in the Variables tab, then you must go back to every place in the macro where you have you used the variable and change the old variable name to the new variable name.

You can choose any variable name you like, although there are a few restrictions on the characters you can choose (see "Variable names and type names" on page 123). You do not have to choose names that begin with an abbreviated form of the data type (such as the str in the string variable $strUserName$), as this book does.

The Type listbox lists the available types for variables and lets you select the type that you want to use for a new variable. The standard types are string, integer, double, boolean, and field. Also, whenever you import a Java class, such as java.util.Hashtable, as an imported type the Type listbox picks up this imported type and adds it to the list of available types, as shown in Figure 43:

```
string
integer
double
boolean
field
java.util.Hashtable
```

*Figure 43. Contents of the Type listbox after an imported type has been declared*

You should use this listbox for assigning a type to a newly created variable. You can come back later and change the variable's type to another type, but, as with

variable names, remember that you might have already used the variable throughout the macro in contexts that require the type that you initially selected. If so, you must go to each of those places and make sure that the context in which you are using the variable is appropriate for its new type.

The Initial Value input field allows you to specify an initial value for the variable. The Macro Editor provides the following default values, depending on the type:

*Table 18. Default initial values for variables*

| **Type of variable:** | **Default initial value:** |
|---|---|
| string | No string |
| integer | 0 |
| double | 0.0 |
| boolean | false |
| field | (No initial value) |
| (imported type) | null |

To specify a new initial value just type over the default value.

The Remove button removes the currently selected variable.

The Import button and the Import popup window are discussed in "Creating an imported type for a Java class" on page 121.

## Creating a new variable

To create a new variable in the Macro Editor, first click the `<new variable>` entry at the end of the Variable listbox. The Macro Editor creates a new variable and assigns to it some initial characteristics that you should modify to fit your needs. The initial values are:

1. An initial name (such as `$a1$`).
2. An initial type (`string`).
3. An initial value, which depends on the type (see Table 18).

Now you should set the values that you want for the new variable. For example, if you are creating an integer variable that is for counting screens and that should have an initial value of 1, then you might set the initial values as follows:

1. In the Name input field, type the name `$intScreenCount$`.
2. In the Type listbox, select the `integer` data type.
3. In the Initial Value field, type 1.

Besides the Variables tab, the Macro Editor provides access, in several convenient locations, to a popup window for creating new variables. For example, in the Variable update action, the Name listbox contains not only all the names of variables that you have already created but also a `<New Variable>` entry. Click this entry to bring up the popup window for creating a new variable. Variables created using this popup window are equivalent to variables created in the Variables tab.

In the Code Editor, you create a new variable using a <create> element. There is a containing element called <vars> that contains all the variables in the macro script, and there is a <create> element for each variable. Figure 44 on page 121 shows a <vars> element that contains five <create> elements:

```
<vars>
   <create name="$strAccountName$" type="string" value="" />
   <create name="$intAmount$" type="integer" value="0" />
   <create name="$dblDistance$" type="double" value="0.0" />
   <create name="$boolSignedUp$" type="boolean" value="false" />
   <create name="$fldFunction$" type="field" />
</vars>
```

*Figure 44. Sample <vars> element*

In the figure above the <vars> element creates one variable from each of the standard data types (string, integer, double, boolean, and field). You should notice that the attributes of each <create> element match the fields on the Variables tab: the **name** attribute contains the variable name, the **type** attribute contains the type, and the **value** field contains the initial value.

You must put all variable creations (<create> elements) inside the <vars> element. The <vars> element itself must appear after the <import> element, if any (see the next section), and before the first macro screen (<screen> element).

## Creating an imported type for a Java class

The way that a Host On-Demand macro imports a Java class is through an imported type. That is, you must first create an imported type and associate it with a particular Java class. You have to do this only once per Java class per macro. Follow these steps to create an imported type:

1. On the Variables tab, click the Import button. The Import popup window appears.
2. In the Imported Types listbox, select the entry <new imported type>.
3. Type the Class name for the type, such as java.util.Hashtable. You must type the fully qualified class name, including the package name if any.
4. Type a Short Name, such as Hashtable. If you do not specify a short name then the Macro Editor uses the fully qualified class name as the short name. If you do specify a short name then you can use either the short name or the fully qualified class name when you refer to the imported type.
5. Click OK.

To create a variable belonging to this imported type, create the variable in the normal way, but select the imported type as the type of the variable. Follow these steps to create a variable of the imported type:

1. In the Variables listbox, click the <new variable> entry at the end. The Macro Editor displays the default initial values in the usual way, including a name (such as $a1$), a type (string), and an initial value (blank).
2. In the Name input field, type the name that you want, such as $ht$.
3. In the Type listbox, select the imported type, such as Hashtable (if you specified a short name when you imported the type) or java.util.Hashtable (if you accepted the default short name, which is the same as the fully qualified class name).
4. In the Initial Value field, you can either leave the field blank (which results in an initial value of null) or specify a method that returns an instance of the class, such as $new Hashtable()$ (using the short name) or $new java.util.Hashtable()$ (using the fully qualified class name).

Notice that the constructors are enclosed in dollar signs ($). You must use dollar signs around every call to a Java method, just as you must use dollar signs around

the name of a variable. (The reason is that the enclosing dollar signs tell the macro runtime that it needs to evaluate the item.)

Going back to the Import popup window, the Imported Types listbox allows you to create new types and to edit or delete the types that you have already created. To create a new type, click the `<new imported type>` entry at the end of the list. To edit a type, select the type in the Imported Types listbox and modify the values in the Class and Short Name input fields. To remove a type, select the type and click Remove.

When you specify a short name, you can use any name, with certain restrictions (see "Variable names and type names" on page 123).

In the Code Editor, you create an imported type using a <type> element. There is a containing element called <import> that contains all the imported types in the macro script, and there is a <type> element for each imported type. Figure 45 shows an <import> element that declares an imported type, followed by a <vars> element that creates and initializes a variable belonging to the imported type:

```
<import>
   <type class="java.util.Hashtable" name="Hashtable" />
</import>

<vars>
   <create name=$ht$ type="Hashtable" value="$new Hashtable(40)$" />
 </vars>
```

*Figure 45. Imported type and variable of that type*

In the figure above the <import> element contains one <type> element, which has a **class** attribute (containing the fully qualified class name, `java.util.Hashtable`) and a **name** attribute (containing the short name, `Hashtable`). The <vars> element contains one <create> element, which as usual specifies a name (`$ht$`), a type (`Hashtable`), and an initial value (which here is not `null` but rather is a call to a constructor that returns an instance of the class, `$new Hashtable(40)$`).

If you are using the Code Editor, you must put all imported types (<type> elements) inside the <import> element. The <import> element itself must appear after the <HAScript> element (see "<HAScript> element" on page 154) and before the first macro screen (<screen> element).

# Issues you should be aware of

## Deploying Java libraries or classes

When the macro runtime finds a call to a Java method, the macro runtime searches the classpath for the class and the method invoked.

If the class belongs to the Java API, then it is already in the classpath (because Host On-Demand requires Java to run) and you do not have to take any action to deploy it.

All other Java classes must be deployed by you to a location where the macro can find them. Depending on the environment, you can deploy the Java classes as class files or as libraries containing Java classes.

For more information on deploying Java libraries and classes, see "Deploying customer-supplied Java archives and classes" in *Planning, Installing, and Configuring Host On-Demand*.

## Variable names and type names

The rules for variable names are as follows:

- A variable name can contain only the alphanumeric characters, underscore (_), or hyphen (-).
- Case is significant (for example, `strTmp` and `strtmp` are two different names).
- A variable name cannot be the same as the short name or the fully qualified class name of an imported type.

The rules for type names are as follows:

- A type name can contain only the alphanumeric characters, underscore (_), hyphen (-), or period (.).
- Case is significant.

## Transferring variables from one macro to another

The PlayMacro action, in which one macro "chains to" another macro (a call without return), allows you to transfer all the variables and their values belonging to the calling macro to the target macro. The target macro has access both to its own variables and to the transferred variables (see "PlayMacro action (<playmacro> element)" on page 88).

## Field variables

A field variable is a type of string variable. It holds a string, just as a string variable does, and you can use it in any context in which a string variable is valid.

However, a field variable differs from a string variable in the way in which a string is stored into the field variable. The string that a field variable contains is always a string that the macro runtime reads from a 3270 or 5250 field in the current session window. To get the macro runtime to read this string from the 3270 or 5250 field, you have to create a Variable update action that specifies:

1. The name of the field variable (such as $fldFilename$).
2. A location string (a string containing a pair of integers separated by a comma, such as `'5,11'`).

When the macro runtime performs the Variable update action it takes the following steps:

1. Looks in the session window at the row and column value specified by the location string.
2. Finds the 3270 or 5250 field in which the row and column value is located.
3. Reads the entire contents of the field.
4. Stores the entire contents of the field as a string into the field variable.

For more information, see "Variable update action with a field variable" on page 98.

# Using variables

## When variables are initialized

The macro runtime assigns initial values to variables at the start of the macro playback, before processing any macro screen.

## Using variables belonging to a standard type

### Using the value that the variable holds

A variable that belongs to a standard type (string, integer, double, boolean) can be used in much the same way as an immediate value of the same type (such as `'Elm Street', 10, 4.6e-2, true`):

- Except for the restrictions listed later in this subsection, a variable of standard type can be used in any input field (in the Macro Editor) or attribute (in the Code Editor) in which an immediate value of the same data type can be used. For example, if an input field (such as the Message Text field on the Message action window) requires a string value, then the field likewise accepts a string variable. See "Equivalents" on page 38.
- Variables can be used with operators and expressions in the same ways that immediate values of the same types are used. See "Operators and expressions" on page 35.
- The value of a variable occurring in a context different from the type of the variable is converted, if possible, to a value of the correct type, in the same way that an immediate value of the same type is converted. See "Automatic data type conversion" on page 37.

However, you cannot use a variable in certain contexts. In the Macro Editor, you cannot use a variable in the following contexts:

- Any field on the General tab.
- The Screen Name field on the Screens tab.
- The value of any field in the PlayMacro action window.

In the Code Editor, you cannot use a variable in the following contexts:

- The name of an attribute of any element.
- The value of any attribute of an <HAScript> element.
- The value of the **name** attribute of a <screen> element.
- The value of the **uselogic** attribute of the <description> element.
- The name of a macro screen in a <nextscreen> element.
- The value of any attribute of a <playmacro> element.

### Writing a value into a variable belonging to a standard type

You can write a value into a variable belonging to a standard type in the following ways:

- Assign an initial value when you create the variable.
- Use a Variable update action to assign a value to the variable.
- Use the Prompt action to get user input and assign it to the variable.
- Use the Extract action to read data from the session window and assign it to the variable.
- Use an action that writes a return code value into a variable (such as the Run program action and the Print actions).

**Restrictions:** You cannot assign one of the following values to a variable of standard type:

- The value `null`. (Exception: If you assign the value `null` to a string variable, it is converted to the string 'null').
- A call to a void method.
- A call to a method that returns an array.

**Writing a Java object into a variable of standard type:** If you write a Java object into a variable of standard type, then the macro runtime calls the toString() method of the imported type and then attempts to assign the resulting string to the variable.

# Using variables belonging to an imported type

## Using the value that the variable holds

You can use the value contained in a variable belonging to an imported type in the following ways:

- You can assign the variable to another variable of the same type using the Variable update action.
- You can call a Java method on the variable (see "Calling Java methods" on page 126). If the Java method returns a value belonging to a standard type (string, integer, double, boolean), then you can use the result as you would use any value of that type.

## Restrictions

You cannot assign the following types of data to a variable of imported type:

- A value or variable belonging to a standard type (string, integer, double, boolean, field).
- A instance of, or a variable belonging to, a different imported type (unless it is a superclass of the imported type).
- An array of instances of objects returned by a method called on a variable of imported type.

If your macro attempts to assign one of these invalid types of values to a variable of imported type then the Macro runtime generates a runtime error and halts the macro

## Writing into the variable belonging to an imported type

You can write a value into a variable of imported type in the following ways:

- You can assign a value to the variable when you create it.
- You can assign a value to the variable using the Variable update action.

You can assign the following types of values to a variable belonging to an imported type:

- An instance of the same type. This instance can be either in a variable of the same type, or from a call to a method that returns an instance of the same type.
- The value null. To signify the value null, you can use one of the following:
  - The keyword `null`.
  - A blank input field (if you are using the Macro Editor), such as the Initial Value field on the Variables tab, or the Value field on the Variable update window.
  - An empty attribute (if you are using the Code Editor), as in the `value` attribute of the following <create> element:

```
<create name=$ht$ type="Hashtable" value="" />
```

### Comparing variables of the same imported type

In any conditional expression (for example, in the Condition field of a conditional action) in which you are comparing two variables of the same imported type, you should implement a comparison method (such as equals()) in the underlying class rather than using the variables themselves. For example,

```
$htUserData.equals($htPortData$)$
```

If instead, you compare the variables themselves (for example `$htUserData$ == $htPortData$`), then:

1. The macro runtime, for each variable, calls the toString() method of the underlying Java class and gets a string result
2. The macro runtime compares the two string results and gets a boolean result.
3. The macro runtime sets the result of the condition to the boolean result obtained in step 2.

This will probably not yield the outcome that you expect from comparing the two variables.

## Calling Java methods

### Where method calls can be used

You can call a method in any context in which the value returned by the method is valid. For example, in an Input action you can set the Row value to the integer value returned by a method, such as:

```
$importedVar.calculateRow()$
```

Also, you can use the Perform action to call a method when you do not need the return variable of the method or when the method has no return value (void) (see “Perform action (<perform> element)” on page 86).

### Syntax of a method call

To call a method belonging to an imported class, use the same syntax that you would use in Java. However, in addition, you must also enclose a method call in dollar signs ($), just as you would a variable. Examples:

```
$new FileInputStream('filename')$
$fis.read()$
```

An immediate string value (such as `'Elm Street'`) passed as a parameter to a method must be enclosed in single quotes, as usual.

### How the macro runtime searches for a called method

When you add a method call (such as `$prp.get( "Group Name" )$`) to a macro script, the Macro Editor does not verify that a called method or constructor exists in the class to which the variable belongs. That check is done by the macro runtime when the call occurs.

The method must be a *public* method of the underlying Java class.

When the macro runtime searches in the Java class for a method to match the method that you have called, the macro runtime maps macro data types (boolean, integer, string, field, double, imported type) to Java data types as shown in

Table 19:

*Table 19.*

| If the method parameter belongs to this macro data type: | Then the macro runtime looks for a Java method with a parameter of this Java data type: |
|---|---|
| boolean | boolean |
| integer | int |
| string | String |
| field | String |
| double | double |
| imported type | underlying class of the imported type |

The macro runtime searches for a called method as follows:

1. The macro runtime searches for the class specified in the imported type definition (such as `java.util.Properties`).
2. The macro runtime searches in the class for a method with the same method signature (name, number of parameters, and types of parameters) as the called method.
3. If the search succeeds, then the macro runtime calls the method.
4. If the search fails, then the macro runtime searches in the class for a method with the same name and number of parameters (disregarding the types of the parameters) as the called method.
   a. If the macro runtime finds such a method, it calls the method with the specified parameters.
   b. If the call returns without an error, the macro runtime assumes that it has called the right method.
   c. If the call returns with an error, the macro runtime searches for another method.
   d. The search continues until all methods with the same name and number of parameters have been tried. If none was successful, then the macro runtime generates a runtime error.

## Converting numbers to and from the local national language format

Different locales represent numbers in different ways. For example, depending on the locale, a decimal number such as 1234.56 is represented as 1,234.56, 1234.56, or 1234,56. Similarly, depending on the locale, a negative number such as –78 is represented as –78 or 78–.

To allow macros to represent numeric strings in locale-specific formats, Host On-Demand provides two conversion methods:

- $FormatStringToNumber(value)$ converts a string in the local format to a number.
- $FormatNumberToString(value)$ converts a number to a string in the local format.

In using these methods you must follow the same rules as with any other method, except that you do not have to import a Java class for these methods. Either of

these two methods can call the other as its input parameter. The output of these methods is converted according to the format of the system locale for the current Host On-Demand session.

## Examples

The following example shows an <input> element that converts the value 3.24 to a string in the local format and sends that string as the input sequence to be typed into the session window at row 1 and column1:

```
<input value="$FormatNumberToString(3.24)$" row="1" col="1"
            movecursor="true" "xlatehostkeys=true" />
```

The following example shows a fragment in which a string variable $num$, which contains a string representation of a number in the local format, is converted to a number, then the number is multiplied by 1000, and the numeric result is converted to a string in the local format:

```
$FormatNumberToString(1000 * $FormatStringToNumber($num$)$)$
```

The following example contains two elements:

- An <extract> element reads a string representation of a number, which may be positive (such as '78') or negative (such as '–78' or '78–', depending on the locale) from the session window and assigns it to a string variable.
- An <if> element converts the string to a number and then tests whether the number is negative.

```
<extract name="'Extract'" planetype="TEXT_PLANE" srow="1" scol="1"
         erow="1" ecol="10" unwrap="false" assigntovar="$value$" />
<if condition="$FormatStringToNumber($value$)$ < 0 "
   ...
</if>
```

# Chapter 12. The graphical user interface

## Updating fields in the Macro Editor

### Using the session window

Even though the Macro Editor window appears on top of the session window, you can still use the session window.

Drag the Macro Editor window to one side of the screen so that you can see the area on the session window that you want to work with. Then click on the session window to make it the current window. (The Macro Editor might still overlap part of the session window.)

### Using the marking rectangle

There are several situations in which you can use the marking rectangle to mark an area of the session window, including:

- Marking a rectangular block of text for a String descriptor.
- Marking the area to be captured by an Extract action.
- Marking the area to be marked by a Box selection action.
- Marking the area to be printed by a Print Extract action.

To mark an area with the marking rectangle, follow these steps:

1. Drag the Macro Editor window to one side of the screen so that you can see the area on the session window that you want to work with. Then click on the session window.
2. Click the mouse on one corner of the area of the session window that you want to mark. You should see the text cursor jump to that row and column position.
3. Hold down the left mouse button and move the mouse. You should see a yellow marking rectangle that changes shape as you move the mouse.
4. Adjust the marking rectangle to surround the area of text that you want to capture, then release the left mouse button.
5. The yellow marking rectangle snaps into place at the nearest character row and column boundaries.
6. The yellow marking rectangle remains visible until you click again on the session window.
7. If you want to mark a different area, start over with step 2 above.

### Using the session window's text cursor

Some actions and descriptors require you to enter a pair of row and column coordinates. Some examples are:

- The Row and Column input fields of the Extract action.

To determine a row and column location on the session window using the text cursor:

1. Drag the Macro Editor window to one side of the screen so that you can see the area on the session window that you want to work with. Then click on the session window to make it the current window. (The Macro Editor might still overlap part of the session window.)

© Copyright IBM Corp. 2003

2. Use the arrow keys to move the text cursor to the row and column location that you are interested in.
3. The row and column numbers are displayed in the lower right hand corner of the session window, in the format row/column (for example, 04/17).
4. Click the Macro Editor window to make it the current window.
5. Enter the row value (such as 4) in the Row input field and the column value (such as 17) in the Column input field.

### Error in specifying a string

In input fields that require a string, you must specify the string in the manner required by the format that you have selected for the macro, either the basic macro format or the advanced macro format (see "Representation of strings and special characters, treatment of operator characters" on page 31).

For example, if you have selected the advanced macro format, but you specify a string that is not surrounded with single quotes (such as `Terminal parameters`), then the Macro Editiro will display an error message like the following:

```
String -- Invalid expression -- Resetting to previous value.
```

To avoid getting this error message, specify the string surrounded with single quotes (such as `'Terminal parameters'`).

In contrast, if you have selected the basic macro format, but you specify a string that is surrounded with single quotes, then you will not get an error message, but the Macro Editor will treat the single quotes as part of the string.

## Using the Code Editor

### Copy and paste a script from this guide into the Code Editor

This section tells you how to copy a macro script from this document to the Code Editor. This text assumes that you are copying an entire macro script, beginning with <HAScript> and ending with </HAScript>. Follow these steps.

1. Start a 3270 Display session and let it connect.
2. Record a simple macro to use as a holder for the script:
   a. Click Record Macro
   b. When the Record Macro window appears:
      1) Click New
      2) Type a name in the Name field, such as `sample1`.
      3) Under Save To click Personal Library
      4) Click OK
   c. The status line on the 3270 Display session window should say, "Recording macro".
   d. Click Stop playing or recording macro.
3. Edit the macro script that you just recorded.
   a. The name of the file for the macro that you just recorded should appear in the window on the left side of the Macro Manager toolbar, such as `sample1.mac`.
   b. Click Edit current macro properties to start the Macro Editor.
   c. When the Macro Editor appears then follow these steps:
      1) Click Code Editor to start the Code Editor.

2) Use the mouse to mark the lines of code that you want to delete.
   a) Which lines to mark for deletion delete depends on the contents of the text that you want to paste into the Code Editor.
   b) However, this example assumes that you want to paste a complete macro script into the Code Editor.
   c) Therefore, in this example you should use the mouse to mark all the lines in the Code Editor for deletion.
3) Type the Delete key to delete the marked area.
4) Copy the entire text of a macro script from this document to the system clipboard, using whichever method you are accustomed to.
5) Make the Code Editor the active window.
6) Use Ctrl-v to paste the macro script into the Code Editor
7) Click OK to close the Code Editor.

d. Click Save to save the macro script and close the Macro Editor

4. The name of the file for the macro that you just edited should appear in the window on the left side of the Macro Manager toolbar, such as `sample1.mac`.
5. Click Play Macro to run the macro.

If you wish to edit this macro, then you can do so either with the Macro Editor or the Code Editor.

# Part 3. The macro language

© Copyright IBM Corp. 2003

# Chapter 13. Features of the macro language

## Use of XML

### XML syntax in the Host On-Demand macro language

A Host On-Demand macro is stored in an XML script using the XML elements of the Host On-Demand macro language. This section describes some of the conventions of XML and gives examples from the Host On-Demand macro language:

- XML code is made up of elements. The Host On-Demand macro language contains about 35 XML elements.
- Element names in the macro language are not case-sensitive, except in the sense that you must write an element in the same combination of upper and lower case in both the begin tag and the end tag. All of the following are correct (the ellipsis "..." is not part of the XML text but is meant to indicate that the element contains other elements):

  ```
  <screen>   ...      </screen>
  <Screen>   ...      </Screen>
  <scrEen>   ...      </scrEen>
  ```

  However, customarily the master element is spelled `HAScript` and the other elements are spelled with all lower case.
- Each XML element has a begin tag and an end tag, as shown in the examples below from the Host On-Demand macro language.:

  ```
  <HAScript> ... </HAScript>
  <import> ... </import>
  <vars> ... </vars>
  <screen> ... </screen>
  ```
- Optionally you can combine the begin tag and end tag of an XML element into one tag. This option is useful when the XML element includes attributes but not other elements. For example,

  ```
  <oia ... />
  <numfields ...  />
  ```
- An element can contain attributes of the form *attribute_name="attribute_value"*. For example:

  ```
  <oia status="NOTINHIBITED" optional="false" invertmatch="false"/>
  <numfields number="80" optional="false" invertmatch="false"/>
  ```

  You can use a pair of empty double quote characters (that is, two double quote characters with nothing in between) to specify that the attribute is not set to a value.

  ```
  <HAScript name="ispf_ex1" description="" timeout="60000" ... author="" ...>
     ...
  </HAScript>
  ```
- An element can include other entire elements between its begin tag and end tag, in much the same way that HTML does. In the example below a <description> element contains two elements: an <oia> element and a <numfields> element.

  ```
  <description>
     <oia status="NOTINHIBITED" optional="false" invertmatch="false">
     <numfields number="80" optional="false" invertmatch="false"/>
  </description>
  ```

© Copyright IBM Corp. 2003

### Code Editor

You can edit the XML text of a macro script directly with the Code Editor (see "Code Editor" on page 9).

You can cut and paste text between the Code Editor and the system clipboard. This is a very important feature, because it allows you to transfer text between the Code Editor and other XML editors or text editors.

## Hierarchy of the elements

Figure 46 lists the begin tags of all the XML elements in the Host On-Demand macro language. This list is not valid in terms of XML syntax and does not indicate where more than one element of the same type can occur. However, the indentation in this list does shows which XML elements occur inside other XML elements. For example, the first element in the list, the <HAScript> element, which is not indented at all, is the master element and contains all the other elements. The second element, the <import> element, occurs inside an <HAScript> element and contains a <type> element. And so on.

```
<HAScript>                Encloses all the other elements in the script.
   <import>               Container for <type> elements.
      <type>              Declares an imported data type (Java class).
   <vars>                 Container for <create> elements.
      <create>            Creates and initializes a variable.
   <screen>               Screen element, contains info about one macro screen.
      <description>       Container for descriptors.
         <attrib>         Describes a particular field attribute.
         <cursor>         Describes the location of the cursor.
         <customreco>     Refers to a custom recognition element.
         <numfields>      Describes the number of fields in the screen.
         <numinputfields> Describes the number of input fields in the screen.
         <string>         Describes a character string on the screen.
         <varupdate>      Assigns a value to a variable.
      <actions>           Container for actions.
         <boxselection>   Draws a selection box on the host application screen.
         <commwait>       Waits for the specified communication status to occur.
         <custom>         Calls a custom action.
         <extract>        Copies data from the host application screen.
         <else>           Allows you to insert an else-condition.
         <filexfer>       Uploads or downloads a file.
         <if>             Allows you to insert an if-condition.
         <input>          Sends keystrokes to the host application.
         <message>        Displays a message to the user.
         <mouseclick>     Simulates a mouse click.
         <pause>          Waits for the specified amount of time.
         <perform>        Calls a Java method that you provide.
         <playmacro>      Calls another macro.
         <prompt>         Prompts the user for information.
         <trace>          Writes out a trace record.
         <varupdate>      Assigns a value to a variable.
      <nextscreens>          Container for <nextscreen> elements.
         <nextscreen>        Contains the name of a valid next macro screen.
      <recolimit>            Takes action if recognition limit is reached.
```

*Figure 46. Hierarchy of elements in the Host On-Demand macro language*

The hierarchy of the elements and the corresponding structure of the macro script are discussed in numerous places in this document. In particular, see the following sections:

- For the <HAScript> element see "Conceptual view of a macro script" on page 20.
- For the <screen> element see "Conceptual view of a macro screen" on page 25.

For descriptions of individual elements see Chapter 14, "Macro language elements" , on page 143.

## Inserting comments into a macro script

You can insert a comment anywhere inside an <HAScript> element by using XML-style comment brackets `<!-- -->` around the text of your comment.

Comments are useful for:

- Organizing a macro script by providing descriptive text.
- Documenting a macro script by explaining complexities.
- Debugging a macro script by commenting out executable elements in order to determine which remaining element is causing a problem.

### Format of comments

When you save a macro script, the Code Editor re-formats your comments if necessary to make them conform to the following format:

- Each comment starts on a new line.
- Each comment is indented the same number of spaces as the element following it.

No matter where you place a comment, the Code Editor will arrange it according to this scheme ( see "Examples of comments").

### Comment errors

The Code Editor will display an error message in the following situations:

- Nested comments
- A comment that comments out part of an executable element.

Also, you cannot use comment brackets `<!-- -->` outside the <HAScript> element. If you do so then the Code Editor will discard those comment brackets and the surrounded text when it saves the script.

### Examples of comments

Here are some examples of the use of comment brackets `<!-- -->` to insert comments:

```
<!--
A multi-line comment that comments on
the following <screen> element
-->
<screen name="Screen1" entryscreen="true" exitscreen="false" transient="false">

<!-- A comment on the following <description> element -->
<description>
   <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
</description>

<! A comment on the following <actions> element -->
<actions>
   <mouseclick row="4" col="16" />
   <input value="3[enter]" row="0" col="0" movecursor="true"
             xlatehostkeys="true" />
```

```
</actions>
<!--
BEGIN
An accidental comment that surrounds part of
a <nextscreens> element,  thereby corrupting
the macro script.
You will get an error when you try to save
this macro script
<nextscreens timeout="0" >
   <nextscreen name="Screen2" />
END of accidental comment
-->
</nextscreens>
</screen>
```

## Debugging macro scripts with the <trace> element

When you are debugging, you can use the <trace> element to send text and values to a trace output destination. In particular, if you include the name of a variable in the output, then the macro runtime will display both the name and the value of the variable in the output, enclosed in curly braces {}. Here is an example:

```
<vars>
<create name="$var1$" type="string" value="'original'" />
</vars>
.
.
<actions>
<trace type="SYSOUT" value="'Before update: '+$var1$" />
<varupdate name="$var1$" value="'updated'" />
<trace type="SYSOUT" value="'After update: '+$var1$" />
</actions>
```

*Figure 47. Example of using the <trace> element*

The code shown in the figure above prints the following text to the Java console:

```
Before update: +{$var1$ = original}
After update: +{$var1$ = updated}
```

*Figure 48. Output from example of using the <trace> element*

Notice that the <trace> action displays each variable in curly brackets {} that contain both the variable name and the contents of the variable.

## Using the Host Access Toolkit product with macros

The separate Host Access Toolkit product includes classes that allow you to dynamically create macro variables, perform macro actions, and run macros. This section contains an example of using the Host Access Toolkit product.

Figure 49 on page 139 shows the first version of a macro that prompts for the user's ID and password, logs on to a host, and says Welcome! This version of the macro does not use the Host Access Toolkit:

```
<HAScript name="Logon" description="" timeout="60000" pausetime="300"
          promptall="true" author="" creationdate="" supressclearevents="false"
          usevars="true" >

  <screen name="Screen1" entryscreen="true" exitscreen="false" transient="false">

    <description>
      <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
    </description>

    <actions>
      <prompt name="'UserID:'" description="" row="20" col="16" len="8"
              default="" clearfield="false" encrypted="false" movecursor="true"
              xlatehostkeys="true" assigntovar="" varupdateonly="false" />
      <input value="'[tab]'" row="0" col="0" movecursor="true"
              xlatehostkeys="true" encrypted="false" />
      <prompt name="'Password:'" description="" row="21" col="16" len="8"
              default="" clearfield="false" encrypted="true" movecursor="true"
              xlatehostkeys="true" assigntovar="" varupdateonly="false" />
      <input value="'[enter]'" row="0" col="0" movecursor="true"
              xlatehostkeys="true" encrypted="false" />
    </actions>
    <nextscreens timeout="0" >
      <nextscreen name="Screen2" />
    </nextscreens>
  </screen>

  <screen name="Screen2" entryscreen="false" exitscreen="true" transient="false">
    <description>
      <oia status="NOTINHIBITED" optional="false" invertmatch="false" />
      <numfields number="7" optional="false" invertmatch="false" />
      <numinputfields number="1" optional="false" invertmatch="false" />
    </description>
    <actions>
      <message title="" value="'Welcome!'" />
    </actions>
    <nextscreens timeout="0" >
    </nextscreens>
  </screen>

</HAScript>
```

*Figure 49. Sample macro that prompts for user's ID and password*

Assume that you want to use this macro in a Host Access Beans program and that you want to store the user ID into a variable and save it for later use (for example, in the Welcome message). You can do this directly by modifying the macro, but one reason for writing a program for this is to avoid having to maintain many different macros for different situations. You could instead have a basic version of the macro and use a program to modify it depending on the situation. The following is an example of how you can do this in Java:

```
// Assume macro is an instantiated Macro with the appropriate listeners set up.
// (See the Javadoc for the Macro bean and the Macro variables demo program,
// MacroVariablesDemo.java, in the Host Access Toolkit samples directory
// for details.)
// Assume macroString is a String containing the previous macro script

macro.setMacro(macroString);
MacroScreens ms = macro.getParsedMacro();
ms.createVariableString("$userid$", null);   //creates a variable $userid$ with
                                             //initial value of ""
MacroScreen mscrn = ms.get(0);   //get the first screen
MacroActions mas = mscrn.getActions();   //get the actions from the first screen
MacroActionPrompt map = (MacroActionPrompt)mas.get(0);   //get the first prompt action
map.setAssignToVar("$userid$"); //assign the prompt response to the variable $userid$
MacroScreen mscrn2 = ms.get(1); //get the second screen
MacroActions mas2 = mscrn2.getActions(); //get the actions from the second screen
MacroActionMessage mam = (MacroActionMessage)mas2.get(0); //get the message action
mam.setMessage("'Welcome ' + $userid$ + '!'"); //change the message to now be a
                                               //personalized message using $userid$
macro.setParsedMacro(ms); //reset the macro with the updated MacroScreens
macro.play(); //play the macro with the changes for variables
```

*Figure 50. Java code to modify a Variable update action and a Prompt action*

Suppose that you now want to add a second message to the actions for Screen2. In this message, you want to display the time and date, which you extract from the screen. You would add the following lines before `macro.setParsedMacro(ms)`:

```
//create a variable $datetimestamp$ with initial value ""
ms.createVariableString("$datetimestamp$", null);

//create new extract to get date and time from second row of screen
MacroActionExtract mae = new MacroActionExtract(2, 35, 2, 71, "'datetimeextract'");

//assign the date and time string to $datetimestamp$
mae.setAssignToVar("$datetimestamp$");

//add the extract after the first message
mas2.add(mae);

//create a new message to display the date and timestamp
MacroActionMessage mam2 = new MacroActionMessage(
   "'You have logged on at ' + $datetimestamp$", "'Date Time Stamp'");

//add the message after the extract
mas2.add(mam2);
```

*Figure 51. Adding a second message*

Note that at the point when the attribute containing the variable(s) is associated with the MacroScreens, you must have already created the variable (through one of the createVariable() methods). For example, this code sequence would also be valid:

```
MacroActionExtract mae = new MacroActionExtract(2, 35, 2, 71, "'datetimeextract'");
mae.setAssignToVar("$datetimestamp$");
ms.createVariableString("$datetimestamp$", null);
mas2.add(mae);
MacroActionMessage mam2 = new MacroActionMessage("'You have logged on at ' +
   $datetimestamp$", "'Date Time Stamp'");
mas2.add(mam2);
```

*Figure 52. Alternate code sequence*

The above sequence is valid because $datetimestamp$ is created before the MacroActionExtract is added to the MacroActions (which are already associated with the MacroScreens because they were pulled from the MacroScreens originally). If the createVariable() method was called at the end of the sequence above, you would have an invalid sequence because the variable $datetimestamp$ would not have been available at the time that the MacroActionExtract and MacroActionMessage were added to the MacroActions and associated with the MacroScreens.

The default value of the MacroScreens method isUseVars() is false. However, if you call one of the createVariable() methods on your MacroScreens, isUseVars() will return true automatically. If you don't create any variables, but want to have your attributes scanned for variables and arithmetic anyway (e.g. you might be writing a chained child macro that has no variables of its own but is anticipating some from the parent), you must call setUseVars(true) on your MacroScreens.

Attributes that can now take variables or expressions as arguments have setAttribute(String) and either getAttributeRaw() or isAttributeRaw() methods available. If you wanted to use an expression now to represent the row attribute for a MacroActionInput, you could call setRow("$rowvar$ + 1"). Subsequently calling getRow() would return the evaluated value of this expression (an integer), whereas calling getRowRaw() would return "$rowvar$ + 1." Note that if you do the following you will get a NumberFormatException:

```
MacroActionInput mai = new MacroActionInput();
mai.setRow("$rowvar$ + 1");
int row = mai.getRow();
```

*Figure 53. Code that causes a NumberFormatException*

This is because `mai` has not yet been associated with any MacroScreens with isUseVars() returning true. Therefore, "$rowvar$ + 1" is being treated as a string rather than a variable plus one. Note also that if you had call the setAttribute() methods to set up variables and expressions after the object containing these attributes have been associated with the MacroScreens, you will likely experience a savings in processing time as the attributes would otherwise need to be reparsed for variables/expressions at the point when they are added to the MacroScreens.

The VariableException class is available for catching exceptions such as illegal expressions (e.g., "45 *") or illegal arithmetic operands (e.g., "'3a' * 2").

A sample program that uses programmed macros, MacroVariablesDemo.java, can be found in the Host Access Toolkit samples directory.

# Chapter 14. Macro language elements

## Specifying the attributes

### XML requirements

In the macro language the value of every attribute must be enclosed in double quotes. For example, in the following <mouseclick> element the values of the **row** and **col** attributes are enclosed in double quotes:

```
<mouseclick row="4" col="51" />
```

### Advanced format in attribute values

As you may remember, even if a macro is in the advanced format, not all input fields in the Macro Editor expect a string to be placed in single quotes (') (see "Representation of strings and special characters, treatment of operator characters" on page 31). Specifically, the advanced format affects input fields only on the following tabs of the Macro Editor:

- Description tab of the Screens tab
- Actions tab of the Screens tab
- Variables tab

Similarly, in the macro language, when you provide a string value for an attribute that corresponds to one of these input fields that is affected by the advanced format, you must enter the string in the advanced format. For example, in the <message> element the strings for both attributes must be written enclosed in single quotes, if the macro is in the advanced format:

```
<message title="'Instructions'"  value="'Check the java console'" />
```

However, if an attribute does not correspond to one of the input fields affected by the advanced format, then you should not write the value enclosed in single quotes, even if the macro is in the advanced formt. For example, the **name** attribute of the <screen> element should never be enclosed in single quotes:

```
<screen name="Screen1" entryscreen="true" exitscreen="true" transient="false" >
   ...
</screen>
```

In the descriptions in this chapter of macro language elements, this book indicates such attributes (attributes that are unaffected by the advanced format) by not specifying a data type. For example, the description of the **name** attribute of the <screen> element is ″Required″ rather than as ″Required string″.

### Typed data

Most attributes require a particular type of data: boolean, integer, string, double, or imported. For these attributes, the same rules apply as in the Macro Editor:

- The consequences of selecting the basic macro format or advanced macro format (see "Choosing a macro format" on page 31).
- The rules for representing strings and special characters, and for treating operator characters (see "Representation of strings and special characters, treatment of operator characters" on page 31).
- The rules for equivalent entities (see "Equivalents" on page 38).

© Copyright IBM Corp. 2003

- The rules for data type conversion (see "Automatic data type conversion" on page 37).
- The rules for arithmetic operators and expressions (see "Arithmetic operators and expressions" on page 35).
- The rules for the string concatenation operator (see "String concatenation operator (+)" on page 36).
- The rules for conditional and logical operators and expressions (see "Conditional and logical operators and expressions" on page 36).
- The rules for representing variables (see "Introduction to the Variables tab" on page 118).
- The rules for calling methods on imported variables (see "Calling Java methods" on page 126).

# <actions> element

## General

The <actions> element, the <description> element, and <nextscreens> element are the three primary structural elements that occur inside the <screen> element (see "Conceptual view of a macro screen" on page 25).

The <actions> element contains elements called actions (such as simulating a keystroke, capturing data, and others) that the macro runtime performs during macro playback (see Chapter 8, "Macro actions", on page 69).

## Attributes

**promptall**

Optional boolean (the default is false). If this attribute is set to true then the macro runtime, before performing any of the actions inside the <actions> element, collects user input for any <prompt> elements inside the element. More specifically:

1. The macro runtime searches the <actions> element to find any <prompt> elements that occur within it.
2. The macro runtime displays the prompts for all the <prompt> elements immediately (all the prompts are combined into one popup).
3. The macro runtime collects the user input for all the popup windows.
4. The macro runtime now performs all the elements in the <actions> element as usual, in sequence.
5. When the macro runtime comes to a <prompt> action, it does not display the popup window for user input, but instead performs the <prompt> action using the input from step 3 above.

The promptall attribute of the <HAScript> element performs the same function for all the <prompt> elements in one macro (see "<HAScript> element" on page 154).

## XML samples

```
<actions promptall="true">
   ...
</actions>
```

*Figure 54. Examples for the <actions> element*

# <attrib> element

## General

The <attrib> element is a descriptor that states the row and column location and the value of a 3270 or 5250 attribute (see "Attribute descriptor (<attrib> element)" on page 65).

## Attributes

**plane** Required. The data plane in which the attribute resides. The valid values are:

- FIELD_PLANE
- COLOR_PLANE
- DBCS_PLANE
- GRID_PLANE
- EXFIELD_PLANE
- Any expression that evaluates to one of the above.

**value** Required. A hexadecimal value in the format 0x37. The value of the attribute.

**row** Required integer. The row location of the attribute in the data plane.

**col** Required integer. The column location of the attribute in the data plane.

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean. See "Inverse Descriptor" on page 55.

## XML samples

```
<attrib value="0x3" row="4" col="14" plane="COLOR_PLANE"
            optional="false" invertmatch="false" />
```

*Figure 55. Examples for the <attribute> element*

# <boxselection> element

## General

The <boxselection> element draws a marking rectangle on the session window, simulating the action in which a user clicks on the session window, holds down mouse button 1, and drags the mouse to create a marking rectangle (see "Box selection action (<boxselection> element)" on page 73).

## Attributes

**srow** Required integer. The row coordinate of the starting corner of the marking rectangle.

**scol** Required integer. The column coordinate of the starting corner of the marking rectangle.

**erow** Required integer. The row coordinate of the ending corner of the marking rectangle.

**ecol** Required integer. The column coordinate of the ending corner of the marking rectangle.

**type** Optional (default SELECT). Specify SELECT to draw a marking rectangle or DESELECT to remove a marking rectangle.

## XML samples

```
<boxselection srow="6" scol="16" erow="7" ecol="73" type="SELECT" />
```

*Figure 56. Examples for the <boxselection> element*

# <comment> element

## General

The <comment> element inserts a text comment as a subelement within a <screen> element. Limitations are:

- You cannot use a <comment> element outside a <screen> element.
- You cannot use more than one <comment> element inside the same <screen> element. If you do so then the Code Editor will discard all the <comment> elements inside that <screen> element except the last one.
- No matter where in the <screen> element you place the <comment> element, the Code Editor will move the comment up to be the first element within the <screen> element.

### A better method for inserting comments

A more flexible method for inserting a comment is to use the XML-style comment brackets `<!-- -->`. See "Inserting comments into a macro script" on page 137.

## Attributes

None.

## XML samples

```
<screen name="Screen2" entryscreen="false" exitscreen="true"
              transient="false">
   <comment>This comment provides information about this macro screen.
   </comment>
   ...
</screen>
```

*Figure 57. Examples for the <comment> element*

# <commwait> element

## General

The <commwait> action waits for the communication status of the session to change to some specified value (see "Comm wait action (<commwait> element)" on page 73). You must specify a timeout value.

## Attributes

**value** Required. The communication status to wait for. The value must be one of the following (see "Communication states" on page 74):

- CONNECTION_INIT
- CONNECTION_PND_ACTIVE
- CONNECTION_ACTIVE
- CONNECTION_READY
- CONNECTION_DEVICE_NAME_READY
- CONNECTION_WORKSTATION_ID_READY
- CONNECTION_PND_INACTIVE
- CONNECTION_INACTIVE

**timeout**

Required integer. A timeout value in milliseconds. The macro runtime terminates the action if the timeout expires before the specified communication status occurs.

## XML samples

```
<commwait value="CONNECTION_READY" timeout="10000" />
```

*Figure 58. Examples for the <commwait> element*

# <condition> element

## General

The <condition> element specifies a conditional expression that the macro runtime evaluates during screen recognition. If the expression evaluates to true then the macro runtime evaluates the descriptor as true. If the expression evaluates to false then the macro runtime evaluates the descriptor as false (see "Condition descriptor (<condition>) element" on page 66).

For more information on conditional expressions see “Conditional and logical operators and expressions” on page 36.

### Attributes

**value** Required expression. The conditional expression that the macro runtime is to evaluate. This conditional expression can contain arithmetic expressions, variables, return values from Java method calls, and other conditional expressions.

**optional**
Optional boolean (the default is false). See “Optional” on page 55.

**invertmatch**
Optional boolean. See “Inverse Descriptor” on page 55.

### XML samples

```
<description>
   <! Check the value of a variable -->
   <condition value="$intPartsComplete$ == 4"
            optional="false" invertmatch="false" />

   <!-- Check the return value of a Java method -->
   <condition value="$htHashTable.size()$ != 0"$
            optional="false" invertmatch="false" />
</description>
```

*Figure 59. Examples for the <condition> element*

## <create> element

### General

The <create> element creates and initializes a variable (see “Creating a new variable” on page 120).

The <create> element must occur inside a <vars> element.

### Attributes

**name** Required. The name that you assign to the variable. There are a few restrictions on the spelling of variable names (see “Variable names and type names” on page 123).

**type** Required. The type of the variable. The standard types are string, integer, double, boolean, field. You an also define an imported type representing a Java class (see “Creating a new variable” on page 120).

**value** Optional. The initial value for the variable. If you do not specify an initial value then the default initial value depends on the variable type (see Table 18 on page 120).

### XML samples

```
<HAScript ... usevars="true" ... >
   <import>
      <type class="java.util.Properties" name="Properties" />
   </import>

   <vars>
      <create name="$prp$" type="Properties" value="$new Properties()$" />
      <create name="$strAccountName$" type="string" value="" />
      <create name="$intAmount$" type="integer" value="0" />
      <create name="$dblDistance$" type="double" value="0.0" />
      <create name="$boolSignedUp$" type="boolean" value="false" />
      <create name="$fldFunction$" type="field" />
   </vars>
   ...
</HAScript>
```

*Figure 60. Examples for the <create> element*

# <cursor> element

## General

The <cursor> element is a descriptor that states the row and column location of the text cursor on the session window (see "Cursor descriptor (<cursor> element)" on page 65).

## Attributes

**row** Required integer. The row location of the text cursor.

**col** Required integer. The column location of the text cursor.

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean. See "Inverse Descriptor" on page 55.

## XML samples

```
<cursor row="4" col="14" optional="false" invertmatch="false" />
```

*Figure 61. Examples for the <cursor> element*

# <custom> element

## General

The <custom> element allows you to invoke a custom Java program from inside the <actions> element of a macro screen. However, you must use the separate Host Access Toolkit product.

Here is an overview of the process:

1. Suppose that you have a Java program that you want to invoke as an action during the processing of a macro screen's <actions> element.

2. In the Code Editor, add the following line to the <actions> element at the location at which you want to invoke the custom Java program:

   ```
   <custom id="'MyProgram1'" args="'arg1 arg2 arg3'"  />
   ```

3. Follow the instructions in the **MacroActionCustom** class of the Host Access Toolkit product. You will create a class that implements **MacroCustomActionListener**. The execute() method will be called with an event when the macro runtime performs the <custom> action in step 2.

## Attributes

**id** Required. An arbitrary string that identifies the custom Java program that you want to run.

**args** Optional. The arguments that you want to pass to the custom Java program.

## XML samples

```
<custom id="'MyProgram1'" args="'arg1 arg2 arg3'" />
<custom id="'MyProgram2'" args="'arg1 arg2'" />
```

*Figure 62. Examples for the <custom> element*

# <customreco> element

## General

This <customreco> element allows you to call out to custom description code. To use the <customreco> element you must have the separate Host Access Toolkit product.

The steps for creating a custom descriptor are as follows:

1. Choose a string to identify the custom description, such as `MyCustomDescriptor01`. An identifier is required because you can have several types of custom descriptions.
2. Implement the **ECLCustomRecoListener** interface. In the **doReco()** method:
   a. Add code to check the identification string to verify that it is yours.
   b. Add your custom description code.
   c. Return true if the custom description is satisfied or false if it is not.
3. Use the Code Editor to add a <customreco> element to the <description> element of the macro screen. The <customreco> element must specify the identifier you chose in step 2.

The macro runtime performs the <customreco> element after performing all the other descriptors.

## Attributes

**id** Required string. The identifier that you have assigned to this custom description.

**optional**
Optional boolean (the default is false). See “Optional” on page 55.

**invertmatch**
Optional boolean. See “Inverse Descriptor” on page 55.

### XML samples

```
<customreco id="'MyCustomDescriptor01'" optional="false" invertmatch="false" />
```

*Figure 63. Examples for the <customreco> element*

## <description> element

### General

The <actions> element, the <description> element, and the <nextscreens> element are the three primary structural elements that can occur inside the <screen> element (see “Conceptual view of a macro screen” on page 25).

The <description> element contains elements called descriptors, each of which states an identifying characteristic of an application screen (see Chapter 7, “Screen description and recognition”, on page 49). The macro runtime uses the descriptors to match the macro screen to an application screen.

### Attributes

**uselogic**
Optional boolean. Allows you to define more complex logical relations among multiple descriptors than are available with the default combining method (see “The uselogic attribute” on page 56).

### XML samples

```
<description uselogic="true">
   ...
</actions>
```

*Figure 64. Examples for the <description> element*

## <else> element

### General

The <else> element contains a sequence of macro actions and must occur immediately after an <if> element. The macro runtime evaluates the conditional expression in the <if> element. Then:

- If the conditional expression is true:
  - The macro runtime performs the sequence of macro actions in the <if> element; and
  - The macro runtime skips the following <else> element if there is one.
- If the conditional expression is false:
  - The macro runtime skips the sequence of macro actions in the <if> element; and

– The macro runtime performs the macro actions in the following <else> element if there is one.

The Macro object uses the <if> element, and if necessary the <else> element, to store a Conditional action (see "Conditional action (<if> element and <else> element)" on page 75).

## Attributes

None.

## XML samples

```
<if condition="($var_int$ > 10)">
   ...
</if>
<else>
   ...
</else>
```

*Figure 65. Examples for the <else> element*

# <extract> element

## General

This <extract> action captures data from the session window (see "Extract action (<extract> element)" on page 77).

## Attributes

For more information on the use of all these attributes see "Extract action (<extract> element)" on page 77.

**name** Required string. A name to be assigned to the extracted data. This name is useful only if you are using the IBM Host Access Toolkit product.

**planetype**
Required. The plane from which the data is to be extracted. To access a data plane other than the TEXT_PLANE you need the IBM Host Access Toolkit product (see "Using the Toolkit to capture data from any data plane" on page 82). Valid values are:

- TEXT_PLANE
- FIELD_PLANE
- COLOR_PLANE
- EXFIELD_PLANE
- DBCS_PLANE
- GRID_PLANE

**srow** Required integer. The row of the first pair of row and column coordinates.

**scol** Required integer. The column of the first pair of row and column coordinates.

**erow** Required integer. The row of the second pair of row and column coordinates.

**scol** Required integer. The column of the second pair of row and column coordinates.

**unwrap**
Optional boolean. Setting this attribute to true causes the macro runtime to capture the entire contents of any field that begins inside the specified rectangle. See "Unwrap Text option" on page 80.

**continuous**
Optional boolean. Setting this attribute to true causes the macro runtime to interpret the row-column coordinates as the beginning and ending locations of a continuous sequence of data that wraps from line to line if necessary. If this attribute is set to false then the macro runtime interprets the row-column coordinates as the upper left and lower right corners of a rectangular area of text. See "Capturing a sequence of text from the session window" on page 79.

**assigntovar**
Optional variable name. Setting this attribute to a variable name causes the macro runtime to store the text plane data as a string value into the variable. If the variable is of some standard type other than string (that is, boolean, integer, or double) then the data is converted to that standard type, if possible. If the data cannot be converted then the macro terminates with a run-time error (see "Specify the variable in which you want the text to be stored" on page 78).

## XML samples

```
<extract name="'Get Data'" srow="1" scol="1" erow="11" ecol="11"
            assignto="$strText$" />
```

*Figure 66. Examples for the <extract> element*

# <filexfer> element

## General

The <filexfer> action transfers a file from the workstation to the host or from the host to the workstation (see "Extract action (<extract> element)" on page 77).

## Attributes

**direction**
Required. Use `send` to transfer a file from the workstation to the host, or `receive` to transfer a file from the host to the workstation.

**pcfile** Required string. The name of the file on the workstation (see "Basic parameters" on page 100).

**hostfile**
Required string. The name of the file on the host (see "Basic parameters" on page 100).

**clear** Required boolean. Set to true for a 3270 Display session or false for a 5250 Display session (see "Advanced parameters" on page 100).

**timeout**
Required integer. A timeout value in milliseconds (the default value is

10000 milliseconds). The macro runtime terminates the transfer if this timeout expires before the file is transferred.

**options**
Optional string. Any additional parameters required by your host system.

**pccodepage**
Optional integer, such as 437. The PC code page to use in mapping characters from the workstation's character set to the host's character set and vice versa. The default value is the code page specified in the session configuration.

**hostorientation**
Optional. For BIDI sessions only (Arabic and Hebrew). Specifies whether text orientation for the host file is right-to-left or left-to-right.

**pcorientation**
Optional. For BIDI sessions only (Arabic and Hebrew). Specifies whether text orientation for the PC file is right-to-left or left-to-right..

**pcfiletype**
Optional. For BIDI sessions only (Arabic and Hebrew). Specifies whether the PC file type is visual or implicit.

**lamalefexpansion**
Optional boolean. For BIDI sessions only (Arabic only). Specifies whether Lam-Alef expansion is on.

**lamalefcompression**
Optional boolean. For BIDI sessions only (Arabic only). Specifies whether Lam-Alef compression is on.

### XML samples

```
<filexfer direction="send" pcfile="'c:\\myfile.txt'"
            hostfile="'myfile text A0'"
            clear="true" timeout="10000" pccodepage="437" />
```

*Figure 67. Examples for the <filexfer> element*

## <HAScript> element

### General

The <HAScript> element is the master element of a macro script. It contains the other elements and specifies global information about the macro (see "Conceptual view of a macro script" on page 20).

### Attributes

**name** Required. The name of the macro.

**description**
Optional. Descriptive text about this macro. You should include here any information that you want to remember about this macro.

**timeout**
Optional integer. The number of milliseconds allowed for screen recognition. If this timeout value is specified and it is exceeded, then the

macro runtime terminates the macro and displays a message (see “Timeout Between Screens (Macro tab)” on page 108). By default the Macro Editor sets this value to 60000 milliseconds (60 seconds).

**pausetime**
Optional integer. The number of milliseconds of delay after each action is performed (see “Pause Between Actions (Macro tab)” on page 111). By default the Macro Editor sets this value to 300 milliseconds.

**promptall**
Required boolean. If this attribute is set to true then the macro runtime, before performing any action in the first macro screen, collects user input for all the <prompt> elements inside the entire macro, combining the individual prompts into one large prompt. The **promptall** attribute of the <actions> element performs a similar function for all the <prompt> elements in one <actions> element (see “<actions> element” on page 144).

**author** Optional. The author or authors of this macro.

**creationdate**
Optional. Information about the dates and versions of this macro.

**suppressclearevents**
Optional boolean (default false). Advanced feature that determines whether the system should ignore screen events when a host application sends a clear screen command immediately followed by an end of record indicator in the data stream. You might want to set this value to true if you have screens in your application flow that have all blanks in them. If there is a valid blank screen in the macro and clear commands are not ignored, it is possible that a screen event with all blanks will be generated by clear commands coming from an ill-behaved host application. This will cause a screen recognition event to be processed and the valid blank screen will match when it shouldn’t have matched.

**usevars**
Required boolean (default false). If this attribute is set to true then the macro uses the advanced macro format (see “Choosing a macro format” on page 31).

**ignorepauseforenhancedtn**
Optional. 3270 Display sessions only. If this attribute is set to true then the macro runtime skips all <pause> elements if the session is a TN3270E session running in contention-resolution mode (see “Attributes that deal with screen completion” on page 114). To re-enable a particular <pause> element see the **ignorepauseoverrideforenhancedtn** attribute of the <pause> element.

**delayifnotenhancedtn**
Optional. 3270 Display Sessions only. This attribute specifies a value in milliseconds and has an effect only when the session is *not* a TN3270E session running in contention-resolution mode. In that situation, this attribute causes the macro runtime to add a pause of the specified duration each time the macro runtime receives a notification that the OIA indicator has changed (see “Attributes that deal with screen completion” on page 114).

## XML samples

```
<HAScript name="ispf_ex2" description="ISPF Sample2" timeout="60000"
            pausetime="300" promptall="true" author="Owner"
            creationdate="Sun Jun 08 12:04:26 PDT 2003"
            supressclearevents="false" usevars="true"
            ignorepauseforenhancedtn="false"
            delayifnotenhancedtn="0">
   ...
</HAScript>
```

*Figure 68. Examples for the <HAScript> element*

# <if> element

## General

The <if> element contains a conditional expression and a sequence of macro actions. The macro runtime evaluates the conditional expression in the <if> element. Then:

- If the conditional expression is true:
  - The macro runtime performs the sequence of macro actions in the <if> element; and
  - The macro runtime skips the following <else> element if there is one.
- If the conditional expression is false:
  - The macro runtime skips the sequence of macro actions in the <if> element; and
  - The macro runtime performs the macro actions in the following <else> element if there is one.

The Macro object uses the <if> element, and if necessary the <else> element, to store a Conditional action (see "Conditional action (<if> element and <else> element)" on page 75).

## Attributes

**condition**

Required. A conditional expression. The conditional expression can contain logical operators and conditional operators and can contain terms that include arithmetic expressions, immediate values, variables, and calls to Java methods (see "Conditional and logical operators and expressions" on page 36).

## XML samples

```
<vars>
   <create name="$condition1$" type="string"/>
   <create name="$condition2$" type="boolean" value="false"/>
   <create name="$condition3$" type="integer"/>
</vars>
<screen>
   <description>
        ...
   </description>
   <actions promptall="true">
      <extract name="Get condition 1" srow="2" scol="1" erow="2"
               ecol="80" assigntovar="$condition1$"/>
      <extract name="Get condition 2" srow="3" scol="1" erow="3"
               ecol="80" assigntovar="$condition2$"/>
      <extract name="Get condition 3" srow="4" scol="1" erow="4"
               ecol="80" assigntovar="$condition3$"/>

      <if condition=
               "(($condition1$ !='')&&
               ($condition2$)||($condition3$ < 100))">
           ...
      </if>
      <else>
           ...
      </else>
   </actions>
</screen>
```

*Figure 69. Examples for the <if> element*

# <import> element

## General

The <import> element, the <vars> element, and the <screen> element are the three primary structural elements that occur inside the <HAScript> element (see "Conceptual view of a macro script" on page 20).

The <import> element is optional. It contains <type> elements each of which declares an imported type based on a Java class (see "Creating an imported type for a Java class" on page 121).

The <import> element must occur after the <HAScript> begin tag and before the <vars> element.

## Attributes

None.

## XML samples

```
<HAScript .... >
   <import>
      <type class="java.util.Properties" name="Properties" />
   </import>

   <vars>
      <create name="$prp$" type="Properties" value="$new Properties()$" />
   </vars>
...
</HAScript>
```

*Figure 70. Examples for the <import> element*

# <input> element

## General

The <input> element sends a sequence of keystrokes to the session window. The sequence can include keys that display a character (such as a, b, c, #, &, and so on) and also action keys (such as [enterreset], [copy], [paste], and others) (see "Input action (<input> element)" on page 82).

## Attributes

**value** Required string. The sequence of keys to be sent to the session window (see "Input string" on page 83).

**row** Optional integer (default is the current position of the text cursor). Row at which typing begins (see "Location at which typing begins" on page 82).

**col** Optional integer (default is the current position of the text cursor). Column at which typing begins (see "Location at which typing begins" on page 82).

**movecursor**
Optional boolean (default is true). Setting this attribute to true causes the macro runtime to move the text cursor to the end of the input (see "Move Cursor to End of Input" on page 83).

**xlatehostkeys**
Optional boolean (default is true). Setting this attribute to true causes the macro runtime to interpret the name of an action key (such as `[enter]`) as an action key rather than as a character sequence (see "Translate Host Action Keys" on page 83).

## XML samples

```
<input value="'3[enter]'" row="4" column="14" movecursor="true"
           xlatehostkeys="true" />
```

*Figure 71. Examples for the <input> element*

## <message> element

### General

The <message> element displays a popup window that includes a title, a message, and an OK button. The macro runtime waits until the user clicks OK before going on to the next action (see "Message action (<message> element)" on page 84).

### Attributes

**title** Optional string (the default is the macro name). A string to be displayed in the caption bar of the popup window.

**value** Required string. The message to be displayed in the popup window.

### XML samples

```
<message title="'Ready'" value="'Ready to process. Click OK to proceed.'" />
```

*Figure 72. Examples for the <message> element*

## <mouseclick> element

### General

The <mouseclick> element simulates a mouse click on the session window by the user. As with a real mouse click, the text cursor jumps to the row and column position where the mouse icon was pointing when the click occurred (see "Mouse click action (<mouseclick> element)" on page 85).

### Attributes

**row** Required integer. The row of the row and column location on the session window where the mouse click occurs.

**col** Required integer. The column of the row and column location on the session window where the mouse click occurs.

### XML samples

```
<mouseclick row="20" col="16" />
```

*Figure 73. Examples for the <mouseclick> element*

## <nextscreen> element

### General

The <nextscreen> element specifies the name of a <screen> element (macro screen) that the macro runtime should consider, among others, as a candidate to be the next macro screen to be processed (see "Valid next screens" on page 103).

The <nextscreen> element must occur within a <nextscreens> element.

### Attributes

**name** Required. The name of the <screen> element that is a candidate to be the next macro screen to be processed.

### XML samples

```
<!--
The effect of the following <nextscreens> element and its contents
is that when the macro runtime finishes performing the actions in
the current screen, it adds ScreenS and ScreenG to the runtime list of
valid next screens.
-->
<nextscreens>
   <nextscreen name="ScreenS">
   <nextscreen name="ScreenG">
</nextscreens>
```

## <nextscreens> element

### General

The <actions> element, the <description> element, and the <nextscreens> element are the three primary structural elements that occur inside the <screen> element (see "Conceptual view of a macro screen" on page 25).

The <nextscreens> element contains <nextscreen> elements, each of which states the name of a macro screen that can validly occur after the current macro screen (see Chapter 9, "Screen Recognition, Part 2", on page 103).

### Attributes

**timeout**

Optional integer. The value in milliseconds of the screen recognition timeout. The macro runtime terminates the macro if it cannot match a macro screen whose name is on the runtime list of valid next screens to the application screen before this timeout expires (see "Timeout settings for screen recognition" on page 107).

### XML samples

```
<!--
The effect of the following <nextscreens> element and its contents
is that when the macro runtime finishes performing the actions in
the current screen, it will attempt to recognize ScreenS and ScreenG.
-->
<nextscreens>
   <nextscreen name="ScreenS">
   <nextscreen name="ScreenG">
</nextscreens>
```

## <numfields> element

### General

The <numfields> element is a descriptor that states the number of 3270 or 5250 fields of all types that exist in the session window (see "Number of Fields descriptor (<numfields> element)" on page 60).

## Attributes

**number**
Required integer. The number of fields in the session window.

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean (the default is false). See "Inverse Descriptor" on page 55.

## XML samples

```
<numfields number="10" optional="false" invertmatch="false" />
```

*Figure 74. Examples for the <numfields> element*

# <numinputfields> element

## General

The <numinputfields> element is a descriptor that states the number of 3270 or 5250 input fields that exist in the session window (see "Number of Input Fields descriptor (<numinputfields> element)" on page 61).

## Attributes

**number**
Required integer. The number of fields in the session window.

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean (the default is false). See "Inverse Descriptor" on page 55.

## XML samples

```
<numinputfields number="10" optional="false" invertmatch="false" />
```

*Figure 75. Examples for the <numinputfields> element*

# <oia> element

## General

The <oia> element is a descriptor that describes the state of the input inhibited indicator in the session window (see "Wait for OIA to Become Uninhibited descriptor (<oia> element)" on page 59).

## Attributes

**status** Required. The value can be:

- NOTINHIBITED

The macro runtime evaluates the descriptor as true if the input inhibited indicator is cleared, or false if the input inhibited indicator is set.

- DONTCARE

  The macro runtime always evaluates the descriptor as true.

- An expression that evaluates to either NOTINHIBITED or DONTCARE

  The macro runtime evaluates the expression and then, depending on the result, evaluates the descriptor as usual.

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean. See "Inverse Descriptor" on page 55.

### XML samples

```
<oia status="NOTINHIBITED" optional="false" invertmatch="false" />
```

*Figure 76. Examples for the <oia> element*

## <pause> element

### General

The <pause> element waits for the specified number of milliseconds (see "Pause action (<pause> element)" on page 86).

### Attributes

**value** Optional integer. The number of milliseconds to wait. If you do not specify this attribute then the Macro object will add the attribute ″value=10000″ (10 seconds) to the element when it saves the script.

**ignorepauseoverrideforenhancedtn**
Optional boolean (the default is false). For 3270 Display sessions only. Setting this attribute to true causes the macro runtime to process the <pause> element even if the **ignorepauseforenhancedtn** attribute of the <HAScript> element is set to true (see "Attributes that deal with screen completion" on page 114).

### XML samples

```
<pause timeout="5000">
```

*Figure 77. Examples for the <pause> element*

## <perform> element

### General

The <perform> element invokes a method belonging to a Java class that you have imported (see "Creating an imported type for a Java class" on page 121).

You can invoke a method in many other contexts besides the <perform> element. However, the <perform> element is useful when you want to invoke a method that does not return a value (see “Perform action (<perform> element)” on page 86).

## Attributes

**value** Required. You must enclose a method call in dollar signs ($), just as you would a variable (see “Syntax of a method call” on page 126). You should specify the parameters, if any, of the method call in the same format that you would use if you were creating a Perform action in the Macro Editor.

## XML samples

```
<!-- Call the update() method associated with the class to which
     importedVar belongs (such as mypackage.MyClass).
-->
<perform value="$importedVar.update( 5, 'Application', $str$)$" />
```

*Figure 78. Examples for the <perform> element*

# <playmacro> element

## General

The <playmacro> element terminates the current macro and launches another macro (see “PlayMacro action (<playmacro> element)” on page 88). This process is called chaining macros.

There are restrictions on where in the <actions> element you can place a <playmacro> element (see “Adding a PlayMacro action” on page 88).

If you are using the Host Access Toolkit then you need to perform the following actions:

- Use the **MacroManager** bean or implement your own **MacroIOProvider** class. Only managed macros can be chained.
- Assign a name to the macro. Macros are chained by macro name.

## Attributes

**name** Required. The name of the target macro. The target macro must reside in the same location as the calling macro (see “Target macro file name and starting screen” on page 89).

**startscreen**
Optional. The name of the macro screen (<screen> element) at which you want the macro runtime to start processing the target macro. Use the value *DEFAULT* or omit this parameter to have the macro runtime start at the usual starting screen of the target macro.

**transfervars**
Required. Setting this attribute to `Transfer` causes the macro runtime to transfer the variables belonging to the calling macro to the target macro (see “Transferring variables” on page 89). The default is `No Transfer`.

## XML samples

```
<playmacro name="ispf_ex1.mac" startscreen="ScreenA"
                transfervars="Transfer" />
```

*Figure 79. Examples for the <playmacro> element*

# <print> element

## General

The <print> element provides printing functions. Three primary print actions (start, extract, and end) are specified through the **action** attribute (see "Print actions (<print> element)" on page 90).

## Attributes

**action** Required. The print action to be performed. Must be one of the following: `start`, `extract`, `end`.

- `start`:

  The macro runtime instantiates a print bean object for the current macro using the Printer Setup options and Page Setup options that you specify (see "Print Start" on page 90).

  Because of the great number of printer setup options and page setup options, and because changing one option might require several other options to be adjusted, you should not use the macro language to specify printer setup options and page setup options. Instead, use the Macro Editor to create a Print start action and use the Printer Setup and Page Setup windows to specify these options (see "Printer Setup and Page Setup" on page 91).

- `extract`:

  The macro runtime copies the text from a rectangular area of the session window that you specify and sends the text to the current print bean (see "Print Extract" on page 91).

- `end`:

  The macro runtime terminates the print bean if one exists (see "Print End" on page 92).

**srow** Required integer when **action** is `extract`. The row of the first pair of row and column coordinates for the rectangular area of text to be printed.

**scol** Required integer when **action** is `extract`. The column of the first pair of row and column coordinates for the rectangular area of text to be printed.

**erow** Required integer when **action** is `extract`. The row of the second pair of row and column coordinates for the rectangular area of text to be printed.

**ecol** Required integer when **action** is `extract`. The column of the second pair of row and column coordinates for the rectangular area of text to be printed.

**assigntovar**
Optional variable. Specifies the name of a variable to contain the return code from the print action.

## XML samples

```
<print action="start" assigntovar="$intReturnCode$" />
<print action="extract" srow="1" scol="1" erow="-1" ecol="-1"  />
<print action="end" />
```

---

# <prompt> element

## General

The <prompt> element displays a popup window prompting the user for input, waits for the user to click OK, and then sends the input to the session window (see "Prompt action (<prompt> element)" on page 92).

## Attributes

**name** Optional string. The text that is to be displayed in the popup window, such as `'Enter your response here:'` (see "Parts of the prompt window" on page 93).

**description**
Optional string. A description of this action. This description is not displayed (see "Parts of the prompt window" on page 93).

**row** Required integer. The row on the session window at which you want the macro runtime to start typing the input from the user.

**col** Required integer. The column on the session window at which you want the macro runtime to start typing the input from the user (see "Handling the input sequence in the session window" on page 94).

**len** Required integer. The number of characters that the user is allowed to enter into the prompt input field (see "Response Length" on page 93).

**default**
Optional string. The text to be displayed in the input field of the popup window. If the user does not type any input into the input field but just clicks OK, the macro runtime will send this default input to the session window (see "Default Response" on page 93).

**clearfield**
Optional boolean. Setting this attribute to true causes the macro runtime, before sending the input sequence to the session window, to clear the input field of the session window in which the row and column location occur (see "Handling the input sequence in the session window" on page 94).

**encrypted**
Optional boolean. Setting this attribute to true causes the macro runtime, when the user types a key into the input field of the window, to display an asterisk (*) instead of the character associated with the key (see "Password Response" on page 93).

**movecursor**
Optional boolean. Setting this attribute to true causes the macro runtime to move the cursor to the end of the input (see "Handling the input sequence in the session window" on page 94).

**xlatehostkeys**
Optional boolean. Setting this attribute to true causes the macro runtime to

interpret the names of action keys (such as [enter]) as action keys rather than as sequences of characters (see "Action keys and Translate Host Action Keys" on page 94).

**assigntovar**
Optional variable name. Setting this attribute to a variable name causes the macro runtime to store the input into the variable whose name you specify here (see "Assigning the input sequence to a variable" on page 94).

**varupdateonly**
Optional boolean. Setting this attribute to true causes the macro runtime to store the input into a variable and not to send it to the session window (see "Handling the input sequence in the session window" on page 94). This attribute takes effect only if the **assigntovar** attribute is set to true.

## XML samples

```
<prompt name="'ID'" row="1" col="1" len="8" description="'ID for Logon'"
        default="'guest'" clearfield="true" encrypted="true"
        assigntovar="$userID$" varupdateonly="true"/>
```

*Figure 80. Examples for the <prompt> element*

# <recolimit> element

## General

The <recolimit> element is an optional element that occurs within a <screen> element, at the same level as the <description>, <actions>, and <nextscreens> elements (see "Recognition limit (General tab of the Screens tab)" on page 108).

The <recolimit> element allows you to take action if the macro runtime processes the macro screen in which this element occurs more than some specified number of times.

## Attributes

**value** Required integer. The recognition limit. If the macro runtime recognizes the macro screen this many times, then the macro runtime does not process the actions of this macro screen but instead performs the specified action.

**goto** Optional string (the default is for the macro runtime to display an error message and terminate the macro). The name of a macro screen that you want the macro runtime to start processing when the recognition limit is reached.

## XML samples

```
<recolimit value="1" goto="RecoveryScreen1" />
```

*Figure 81. Examples for the <recolimit*

## <runprogram> element

### General

The <runprogram> element launches a native application and optionally waits for it to terminate. You can provide input parameters for the application and store the return code into a variable (see “Run program action (<runprogram> element)” on page 95).

### Attributes

**exe** Required string. The path and name of the native application (see “Launching the native application” on page 95).

**param** Optional string. Any arguments that should be specified when the native application is launched.

**wait** Optional boolean. Setting this attribute to true causes the macro runtime to wait until the native application terminates.

**assignexitvalue**
Optional variable. The name of a variable into which the return value from native application should be stored.

### XML samples

```
<runprogram exe=
      "'c:\\Program Files\\Windows NT\\Accessories\\Wordpad.exe'"
      param="'c:\\tm\\new_file.doc'" wait="true"
      assignexitvalue="$intReturn$" />
<message title="" value="'Return value is '+
         $intReturn$" />
```

*Figure 82. Examples for the <runprogram> element*

## <screen> element

### General

The <screen> element, the <import> element, and the <vars> element are the three primary structural elements that occur inside the <HAScript> element (see “Conceptual view of a macro script” on page 20).

Multiple screen elements can occur inside a macro. One <screen> element contains all the information for one macro screen (see “The macro screen and its subcomponents” on page 23).

The <screen> element contains three primary structural elements: the <actions> element, the <description> element, and <nextscreens> (see “Conceptual view of a macro screen” on page 25).

### Attributes

**name** Required. The name of this <screen> element (macro screen). The name must not be the same as the name of an already existing <screen> element.

**entryscreen**
Optional boolean (the default is false). Setting this attribute to true causes the macro runtime to treat this <screen> element as a valid beginning screen for the macro (see "Entry screens" on page 105).

**exitscreen**
Optional boolean (the default is false). Setting this attribute to true causes the macro runtime to treat this <screen> element as a valid ending screen for the macro (see "Exit screens" on page 106).

**transient**
Optional boolean (the default is false). Setting this attribute to true causes the macro runtime to treat this <screen> element as a screen that can appear at any time and that always needs to be cleared (see "Transient screens" on page 106).

**pause** Optional integer (the default is -1). Specifying a value in milliseconds for this attribute causes the macro runtime, for this <screen> element, to ignore the default pause time between actions (set using the **pausetime** attribute of the <HAScript> element) and to use this value instead (see "Set Pause Time (General tab of the Screens tab)" on page 111).

## XML samples

```
<screen name="ScreenB" entryscreen="false" exitscreen="false"
            transient="false">
   <description>
      ...
   </description>
   <actions>
      ...
   </actions>
   <nextscreens>
      ...
   </nextscreens>
</screen>
```

*Figure 83. Examples for the <screen> element*

# <string> element

## General

The <string> element is a descriptor that specifies a sequence of characters and a rectangular area of the session window in which the sequence occurs (see "String descriptor (<string> element)" on page 61).

The sequence of characters can occur anywhere in the rectangular block.

## Attributes

**value** Required string. The sequence of characters.

**row** Optional integer (the default is to search the entire screen). The row location of one corner of a rectangular block of text.

**col** Optional integer. The column location of one corner of a rectangular block of text.

**erow** Optional integer. The row location of the opposite corner of a rectangular block of text.

**ecol** Optional integer. The column location of the opposite corner of a rectangular block of text.

**casesense**
Optional boolean (the default is false). Setting this attribute to true causes the macro runtime to do a case-sensitive string compare.

**wrap** Optional boolean (the default is false).

- Setting this attribute to false causes the macro runtime to search for the sequence of characters in each separate row of the rectangular block of text. If the sequence of characters wraps from one row to the next, the macro runtime will not find it.
- Setting this attribute to true causes the macro runtime to check for the sequence of characters occurring in any row or wrapping from one row to the next of the rectangular block of text (see "How the macro runtime searches the rectangular area (Wrap option)" on page 62).

**optional**
Optional boolean (the default is false). See "Optional" on page 55.

**invertmatch**
Optional boolean. See "Inverse Descriptor" on page 55.

### XML samples

```
<!-- The string must occur in one specific area of a single row  -->
<string value="'Utility Selection Panel'" row="3" col="28"
            erow="3" ecol="51" casesense="false" wrap="false"
            optional="false" invertmatch="false" />

<!-- The string can occur in any single row of the session area -->
<string value="'Utility Selection Panel'" row="1" col="1"
            erow="-1" ecol="-1" casesense="false" wrap="false"
            optional="false" invertmatch="false" />
```

*Figure 84. Examples for the <string> element*

## <trace> element

### General

The <trace> element sends a trace message to a trace destination that you specify, such as the Java console (see "Trace action (<trace> element)" on page 96).

### Attributes

**type** Required. The destination for the trace data. The destination must be one of the following:

- HODTRACE: The Host On-Demand Trace Facility.
- USER: A user trace handler.
- SYSOUT: The Java console.

**value** Required string. The string that is to be sent to the trace destination.

## XML samples

```
<trace type="SYSOUT" value="'The value is '+$strData$" />
```

*Figure 85. Examples for the <trace> element*

# <type> element

## General

The <type> element declares an imported type (such as `Properties`) that represents a Java class (such as `java.util.Properties`). After you have declared the type, you can create variables based on the type, create an instance of the Java class, and call methods on the instance (see "Creating an imported type for a Java class" on page 121).

A type can also be used for directly calling static methods (no need to instantiate).

The <type> element must occur inside a <import> element.

## Attributes

**class** Required. The fully qualified class name of the class being imported, including the package name if any (such as `java.util.Properties`).

**name** Optional. A short name (such as `Properties`) that you can use elsewhere in the macro to refer to the imported type. If you do not specify a short name, then the short name is the same as the fully qualified class name. There are a few restrictions on the spelling of type names (see "Variable names and type names" on page 123).

## XML samples

```
<import>
   <type class="java.util.Date" name="Date"/>
   <type class="java.io.FileInputStream"/>
   <type class="com.ibm.eNetwork.beans.HOD.HODBean" name="HODBean"/>
   <type class="myPackage.MyClass" name="MyClass"/>
</import>
```

# <vars> element

## General

The <vars> element, the <import> element, and the <screen> element are the three primary structural elements that occur inside the <HAScript> element (see "Conceptual view of a macro script" on page 20).

The <vars> element is optional. It contains <create> elements, each of which declares and initializes a variable (see "Creating a new variable" on page 120). The <vars> element must occur after the <import> element and before the first <screen> element.

To use variables, you must set the **usevars** element in <HAScript> to true.

## Attributes

None.

## XML samples

```
<HAScript ... usevars="true" .... >
   <import>
      <type class="java.util.Properties" name="Properties" />
   </import>

   <vars>
      <create name="$prp$" type="Properties" value="$new Properties()$" />
      <create name="$strAccountName$" type="string" value="" />
      <create name="$intAmount$" type="integer" value="0" />
      <create name="$dblDistance$" type="double" value="0.0" />
      <create name="$boolSignedUp$" type="boolean" value="false" />
      <create name="$fldFunction$" type="field" />
   </vars>
   ...
</HAScript>
```

*Figure 86. Examples for the <vars> element*

# <varupdate> element

## General

The <varupdate> element causes the macro runtime to store a specified value into a specified variable. The value can be an immediate value, a variable, a call to a Java method, or an arithmetic expression that can contain any of these values. If the value is an expression, then during macro playback the macro runtime evaluates the expression and stores the resulting value into the specified variable (see "Variable update action (<varupdate> element)" on page 97).

You can also use the <varupdate> action in a <description> element (see "Variable update action (<varupdate> element)" on page 97).

For more information on variables see Chapter 11, "Variables and imported Java classes", on page 117.

## Attributes

**name** Required. The name of the variable.

**value** Required string. The value or expression to be assigned to the variable.

## XML samples

```
<type>
   <type class="mypackage.MyClass" name="MyClass" />
   <type class="java.util.Hashtable" name="Hashtable" />
   <type class="java.lang.Object" name="Object" />
</type>

<vars>
   ...
</vars>

<screen>
   <description>
   ...
   </description>
   <actions>
      <varupdate name="$var_boolean1$"  value="false" />
      <varupdate name="$var_int1$"      value="5" />
      <varupdate name="$var_double1$"   value="5" />
      <varupdate name="$var_string1$"   value="'oak tree'" />
      <varupdate name="$var_field1$"    value="4,5" />

      <!-- null keyword -->
      <varupdate name="$var_importedMC1$"  value="null" />
      <!--  Equivalent to null keyword for an imported type  -->
      <varupdate name="$var_importedMC2$"  value="" />

      <varupdate name="$var_importedMC4$"
                 value="$new MyClass( 'myparam1', 'myparam2' )$" />
      <varupdate name="$var_importedMC5$"
                 value="$var_importedMC4$" />
      <varupdate name="$var_importedMC6$"
                 value="$MyClass.createInstance( 'mystringparam1' )$" />
      <varupdate name="$var_boolean2$"
                 value="$var_importedMC4.isEmpty()$" />
      <varupdate name="$var_int2$"
                 value="$($var_importedMC4.getHashtable()$).size()$" />
      <varupdate name="$var_double2$"
                 value="$var_importedMC4.getMeters()$" />
      <varupdate name="$var_string2$"
                 value="$var_importedMC4.toString()" />
   </actions>
</screen>
```

*Figure 87. Examples for the <varupdate> element*

# Chapter 15. Sample macro code

## Copy CICS transaction records into Excel spreadsheet or DB2 database

### Introduction

NOTE: This sample requires Microsoft Excel or IBM DB2.

This sample macro reads transaction records from the CICS transaction **amnu**, which is a very small sample database, and writes the records into an Excel spreadsheet or IBM DB2.

The files for this sample are stored in the following directory, where <install> stands for the Host On-Demand installation directory, and where xx stands for your language id (such as en):

```
<install>\HOD\xx\doc\macro\samples\amnu
```

The files in this sample are the following:

- amnu.mac, amnudb2.mac

  These are the macro files. The first one is for use with the Excel spreadsheet. The second is for use with DB2.
- amnu.xls

  This is the Excel spreadsheet.
- EditDB.java, EditDB.jar

  EditDB.java is a Java source file containing source code for the EditDB class. The macro uses the EditDB class to write to the spreadsheet or database. EditDB.class is compiled with Java 2 and stored in the EditDB.jar. EditDB.jar is a Java 2 JAR file.

### Steps for running Excel sample (Sun Java 2 plug-in, Windows only)

1. Configure Java security policy

   You will need to grant certain permissions for the Host On-Demand applet in order to run this sample. You can alter the .java.policy file by using policytool or you can create a new policy file and specify this file in the plug-in Java Runtime Parameters(`-Djava.security.policy=PolicyFileName`).

   Your new policy file should contain the following and should be located in your local home directory.

   ```
   grant {
       permission java.lang.RuntimePermission
           "accessClassInPackage.sun.jdbc.odbc";
       permission java.util.PropertyPermission
          "file.encoding", "read";   };
   ```

   If you want to change .java.policy (and not set the parameter above in your plug-in), launch the policytool executable in the bin directory of your Java plug-in install and set the permissions specified in the lines above.
2. Set up the Excel spreadsheet as an ODBC data source

© Copyright IBM Corp. 2003

a. On your Windows machine, go to Settings->Control Panel->Administrative Tools->Data Sources(ODBC).
b. Click Add...
c. Select Microsoft Excel Driver (*.xls).
d. Hit Finish.
e. Give a data source name of amnu and give any description you desire (or leave it blank).
f. Use the Select Workbook button to find the spreadsheet provided in this example. Hit OK.
g. Deselect the Read Only option for this source. You may need to hit an Options>> button to find this option.
h. Hit OK. You now have the amnu.xls spread sheet available as the ODBC data source amnu.

.

3. Create a new Deployment Wizard page that gives the Host On-Demand client access to the EditDB class.
   a. Start Deployment Wizard.
   b. On the Additional Options page, click "Advanced Options..."
   c. On the Add HTML Parameters panel, add a parameter with Name "AdditionalArchives" and Value "amnu"
4. Place amnu.jar in your Host On-Demand publish directory
5. Open your newly created page in a web browser and start your CICS session.

   USING AMNU:

   The transaction amnu is a small sample database that is provided with CICS. To start amnu, follow these steps:
   a. Log onto CICS
   b. At the CICS prompt, type amnu and type enter.

   The amnu menu comes up in the upper left quadrant of the session window and displays operator instructions.

   To see if there are any records in the database, follow these steps:
   a. In the ENTER TRANSACTION: field, type `abrw`
   b. Leave the NUMBER field blank.
   c. Type enter.

   The amnu transaction displays the first 4 records. Follow the instructions on the screen to browse the database.

   If the database is empty, you need to add records to it before you run the macro. To add records to the database, follow these steps:
   a. In the ENTER TRANSACTION: field, type `aadd`
   b. In the NUMBER FIELD, enter a number for the record that you want to add, such as `40`.
   c. Type enter.
   d. Follow the instructions on the screen to provide information for the new record.
6. Load amnu.mac into your session.
   a. If it is not already visible, display the Macro Manager toolbar by selecting View->Macro Manager on the session toolbar.

b. Click on the Edit current macro properties icon.
c. Click on the Import... button on the Macro Editor.
d. Browse to the location of amnu.mac and open it.
e. Hit Save to save the macro to your current session and close the Macro Editor.

7. Navigate to the amnu menu screen and hit the Play macro icon.

   You will be prompted to enter a record number. Enter a number of a transaction that you have entered or that you saw when you browsed through the database in step 5. Hit OK. The application screen will display the contents of the record corresponding to the number that you entered. You will be prompted again to ask if you would like to save the record on the screen to your database. The default response is ″Y.″ Hit OK. You will again be prompted to Enter a transaction number. You could continue to enter as many record numbers as you like (and you will be notified if you enter an invalid number), but this time hit Q to quit. Hit OK to close the prompt, and OK again to dismiss the message ″Good Bye!″ The macro will end, and amnu.xls will open. You should see the contents of the record you just saved inside the spreadsheet.

   Hopefully this sample will get you thinking about the powerful ways in which you can put macros to use for your business. Note that this sample can easily be modified, for example to write to a different kind of database (see below for directions on writing to DB2) or to read from the local database and write to the amnu database. Note that this sample was designed to be short and simple, not a lesson in best practices of Java or macro coding. For example, you may have noticed that we are connecting to and disconnecting from the local database every time we write a record out. This could be avoided by writing ″connect″ and ″disconnect″ macros that are linked to the amnu macro such that there is only one connect and one disconnect for each macro play.

If you look inside the Excel macro amnu.mac, you'll see that it is using the driver sun.jdbc.odbc.JdbcOdbcDriver to connect to the Excel spreadsheet. If this class is not in your classpath, the sample will not run properly. This class is included in the Sun Java 2 plug-ins but not in IBM plug-ins.

## Steps for running DB2 sample

If you are using an IBM plug-in and do not have the sun.jdbc.odbc package in your classpath, you can instead run this sample with IBM DB2.

1. Create a DB2 database. Call the database ″AMNU″ and create a table ″CUSTRECS″ with columns ″NAME″, ″ADDRESS″, ″PHONE″, ″DATE″, ″AMOUNT″ and ″COMMENT″.
2. Put appropriate DB2 drivers in your classpath. Rather than using sun.jdbc.odbc.JdbcOdbcDriver, we will now use COM.ibm.db2.jdbc.net.DB2Driver to connect to our local database. This and other needed classes are found in db2java.zip, which was likely placed in \SQLLIB\java when you installed DB2. There are different ways of getting these needed files in your classpath, depending on your setup.
3. Edit the macro for your DB2 database.
   a. In the line:

```
<create name="$database$" type="DB"
     value="$new DB('jdbc:db2://hostname:6789/AMNU',
    'COM.ibm.db2.jdbc.net.DB2Driver')$" />
```

      change hostname to the name of your machine running DB2.

   b. Find the following two perform actions:

```
<perform value="$database.setUserID('db2admin')$"  />
<perform value="$database.setPassword('db2admin')$"  />
```

and modify them with and ID and Password that will connect to your DB2 database.

4. Follow steps 3-7 above, this time importing AMNUDB2.mac.

Note some differences in the two macros:

- The syntax of the table name in the SQL query is different. For Excel:
  ```
  [CUSTRECS$]
  ```
  For DB2:
  ```
  CUSTRECS
  ```
- As mentioned before, we are using a different driver to connect to the database.
- This time we needed to use EditDB's setUserID and setPassword methods to specify the appropriate ID and Password to connect to the database.
- This macro does not launch the local database when you are done making your additions. You can verify that the records were properly added by performing a "Select * from CUSTRECS" query on the database.

# Appendix A. Additional information

## The default combining rule for multiple descriptors in one macro screen

### Statement of the rule

Here is the rule:

1. Evaluate all the required descriptors (that is, descriptors for which the Optional field is set to false).
   a. If all are true, then the screen matches.
   b. Otherwise, go to step 2.
2. Start evaluating the optional descriptors (descriptors for which the Optional field is set to true).
   a. If any optional descriptor is true, then the screen matches.
   b. Otherwise, keep evaluating optional descriptors.
3. If you reach here, then the macro screen does not match the application screen.

## Mnemonic keywords for the Input action

This section contains the mnemonic keywords for the Input action and the type of session or essions in which the mnemonic is supported. Session support for a given mnemonic is denoted by an X, along with any special notes that apply to the function.

*Table 20. Keywords for the Input action*

| Function: | Keyword: | 3270: | 5250: | VT: | CICS: |
|---|---|---|---|---|---|
| Attention | [attn] | x | x | | x |
| Alternate view | [altview] | x[3] | x[3] | | x[3] |
| Backspace | [backspace] | x | x | x[1] | x |
| Backtab | [backtab] | x | x | | x |
| Beginning of Field | [bof] | x | x | | x |
| Clear | [clear] | x | x | x[1] | x |
| Cursor Down | [down] | x | x | x[1] | x |
| Cursor Left | [left] | x | x | x[1] | x |
| Cursor Right | [right] | x | x | x[1] | x |
| Cursor Select | [cursel] | x | x | x[1] | x |
| Cursor Up | [up] | x | x | x[1] | x |
| Delete Character | [delete] | x | x | x[1, 2] | x |
| Display SO/SI | [dspsosi] | x[3] | x[3] | | x[3] |
| Dup Field | [dup] | x | x | | x |
| Enter | [enter] | x | x | x | x |

© Copyright IBM Corp. 2003

*Table 20. Keywords for the Input action (continued)*

| Function: | Keyword: | 3270: | 5250: | VT: | CICS: |
|---|---|---|---|---|---|
| End of Field | [eof] | x | x | x[1, 2] | x |
| Erase EOF | [eraseeof] | x | x | | x |
| Erase Field | [erasefld] | x | x | | x |
| Erase Input | [erinp] | x | x | | x |
| Field Exit | [fldext] | | x | | |
| Field Mark | [fieldmark] | x | x | | |
| Field Minus | [field-] | | x | | |
| Field Plus | [field+] | | x | | |
| F1 | [pf1] | x | x | x | x |
| F2 | [pf2] | x | x | x | x |
| F3 | [pf3] | x | x | x | x |
| F4 | [pf4] | x | x | x | x |
| F5 | [pf5] | x | x | x | x |
| F6 | [pf6] | x | x | x | x |
| F7 | [pf7] | x | x | x | x |
| F8 | [pf8] | x | x | x | x |
| F9 | [pf9] | x | x | x | x |
| F10 | [pf10] | x | x | x | x |
| F11 | [pf11] | x | x | x | x |
| F12 | [pf12] | x | x | x | x |
| F13 | [pf13] | x | x | x | x |
| F14 | [pf14] | x | x | x | x |
| F15 | [pf15] | x | x | x | x |
| F16 | [pf16] | x | x | x | x |
| F17 | [pf17] | x | x | x | x |
| F18 | [pf18] | x | x | x | x |
| F19 | [pf19] | x | x | x | x |
| F20 | [pf20] | x | x | x | x |
| F21 | [pf21] | x | | x | x |
| F22 | [pf22] | x | | x | x |
| F23 | [pf23] | x | | x | x |
| F24 | [pf24] | x | | x | x |
| Help | [help] | | x | | |
| Home | [home] | x | x | x[1, 2] | x |
| Insert | [insert] | x | x | x[1, 2] | x |
| Keypad 0 | [keypad0] | | | x | |
| Keypad 1 | [keypad1] | | | x | |
| Keypad 2 | [keypad2] | | | x | |
| Keypad 3 | [keypad3] | | | x | |
| Keypad 4 | [keypad4] | | | x | |

*Table 20. Keywords for the Input action (continued)*

| Function: | Keyword: | 3270: | 5250: | VT: | CICS: |
|---|---|---|---|---|---|
| Keypad 5 | [keypad5] | | | x | |
| Keypad 6 | [keypad6] | | | x | |
| Keypad 7 | [keypad7] | | | x | |
| Keypad 8 | [keypad8] | | | x | |
| Keypad 9 | [keypad9] | | | x | |
| Keypad Dot | [keypad.] | | | x | |
| Keypad Enter | [keypadenter] | | | x | |
| Keypad Comma | [keypad,] | | | x | |
| Keypad Minus | [keypad-] | | | x | |
| New Line | [newline] | x | x | | x |
| PA1 | [pa1] | x | x | | x |
| PA2 | [pa2] | x | x | | x |
| PA3 | [pa3] | x | x | | x |
| Page Up | [pageup] | x | x | x[1, 2] | x |
| Page Down | [pagedn] | x | x | x[1, 2] | x |
| Reset | [reset] | x | x | x | x |
| System Request | [sysreq] | x | x | | x |
| Tab Field | [tab] | x | x | x[1] | x |
| Test Request | [test] | | x | | |

1. VT supports this function but it is up to the host application to act on it.
2. Supported in VT200 mode only.
3. The function is only available in a DBCS session.

The following table shows the bidirectional keywords for the Input action.

*Table 21. Bidirectional keywords for the Input action*

| Function: | Keyword: | 3270: | 5250: | VT: | CICS: |
|---|---|---|---|---|---|
| Auto Push | [autopush] | x | | | x |
| Auto Reverse | [autorev] | x | | x | |
| Base | [base] | x | x | | |
| BIDI Layer | [bidilayer] | | | | |
| Close | [close] | | x | | |
| CSD | [csd] | x | | | x |
| End Push | [endpush] | x | | | x |
| Field Reverse | [fldrev] | x | x | | x |
| Field Shape | [fieldshape] | x | | | x |
| Final | [final] | x | | | x |
| Initial | [initial] | x | | | x |
| Isolated | [isolated] | x | | | x |

*Table 21. Bidirectional keywords for the Input action (continued)*

| Function: | Keyword: | 3270: | 5250: | VT: | CICS: |
|---|---|---|---|---|---|
| Latin Layer | [latinlayer] | x | x | | |
| Middle | [middle] | x | | | x |
| Push | [push] | x | | | x |
| Screen Reverse | [screenrev] | x | x | | x |

# Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or region or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan
```

**The following paragraph does not apply to the United Kingdom or any other country or region where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2003

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Corporation
Department T01
Building B062
P.O. Box 12195
Research Triangle Park, NC 27709-2195
U.S.A.
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee. The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

# Appendix C. Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both: **IBM**

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation.

Other company, product, and service names may be trademarks or service marks of others.

© Copyright IBM Corp. 2003

Printed in U.S.A.

SC31-6378-00