Novell exteNd Composer

5.2

www.novell.com

HP3000 CONNECT USER'S GUIDE

Novell®

Legal Notices

Copyright © 2004 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher. This manual, and any portion thereof, may not be copied without the express written permission of Novell, Inc.

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to makes changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

This product may require export authorization from the U.S. Department of Commerce prior to exporting from the U.S. or Canada.

Copyright ©1997, 1998, 1999, 2000, 2001, 2002, 2003 SilverStream Software, LLC. All rights reserved.

SilverStream software products are copyrighted and all rights are reserved by SilverStream Software, LLC

Title to the Software and its documentation, and patents, copyrights and all other property rights applicable thereto, shall at all times remain solely and exclusively with SilverStream and its licensors, and you shall not take any action inconsistent with such title. The Software is protected by copyright laws and international treaty provisions. You shall not remove any copyright notices or other proprietary notices from the Software or its documentation, and you must reproduce such notices on all copies or extracts of the Software or its documentation. You do not acquire any rights of ownership in the Software.

Patent pending.

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.

www.novell.com

exteNd Composer *HP3000 Connect User's Guide*
June 2004

Online Documentation: To access the online documemntation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JDOM.JAR

Copyright (C) 2000-2002 Brett McLaughlin & Jason Hunter. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer

Pages, Java Naming and Directory Interface, JDK, JDBC, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, SunWorkShop, XView, Java WorkShop, the Java Coffee Cup logo, Visual Java, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Indiana University Extreme! Lab Software License

Version 1.1.1

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab (http://www.extreme.indiana.edu/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Indiana University" and "Indiana University Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/. 5. Products derived from this software may not use "Indiana University" name nor may "Indiana University" appear in their name, without prior written permission of the Indiana University.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Phaos

This Software is derived in part from the SSLavaTM Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved. Customer is prohibited from accessing the functionality of the Phaos software.

W3C

W3C® SOFTWARE NOTICE AND LICENSE

This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications: 1.The full text of this NOTICE in a location viewable to users of the redistributed or derivative work. 2.Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code. 3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Contents

About This Book . . . 7

1 Welcome to exteNd Composer and HP3000 Connect . . . 9
Before You Begin . . . 9
About exteNd Composer Connects . . . 9
What Is HP3000 Terminal? . . . 10
About exteNd Composer's HP3000 Component . . . 11
What Applications Can You Build Using the HP3000 User Interface Component Editor? . . . 11

2 Getting Started with the HP3000 Component Editor . . . 13
Steps Commonly Used to Create an HP3000 Component . . . 13
Templates for Your Component . . . 13
Creating an HP3000 Connection Resource . . . 14
About Connection Resources . . . 14
Code Page Support . . . 16
About Constant and Expression Driven Connections . . . 16

3 Creating an HP3000 Component . . . 19
Before Creating an HP3000 Component . . . 19
About the HP3000 Component Editor Window . . . 21
About the HP3000 Native Environment Pane . . . 22
About HP3000 Keyboard Support . . . 22
About the Screen Object . . . 25
What it is . . . 25
How it works . . . 25
HP3000-Specific Toolbar Buttons . . . 26
HP3000-Specific Menu Bar Items . . . 27
HP3000-Specific Context-Menu Items . . . 27
Native Environment Pane Context Menu . . . 27
Action Pane Context Menu . . . 28

4 Performing Basic HP3000 Actions . . . 29
About Actions . . . 29
About HP3000-Specific Actions . . . 29
The Send Buffer Action . . . 30
How Keys Are Displayed in the Action Model . . . 31
The Check Screen Action . . . 31
Using Actions in Record Mode . . . 34
HP3000-Specific Expression Builder Extensions . . . 34
Login . . . 34
Screen Methods . . . 35
Keys . . . 37
Screen Selections in the HP3000 Connect . . . 37
Selecting Continuous Data . . . 38
Selecting Rectangular Regions . . . 38
Recording an HP3000 Session . . . 39
Looping Over Multiple Rows in Search of Data . . . 43
Editing a Previously Recorded Action Model . . . 47
Changing an Existing Action . . . 48
Adding A New Action . . . 51
About Adding Alias Actions . . . 53

Deleting an Action . . . 54
Testing your HP3000 Component . . . 54
Using the Animation Tools . . . 55
Tips for Building Reliable HP3000 Components . . . 56
Using Other Actions in the HP3000 Component Editor . . . 57
Handling Errors and Messages . . . 57
Finding a “Bad” Action . . . 59

5 Advanced HP3000 Actions . . . 61
Data Sets that Span Screens . . . 61
Dealing with Redundant Data . . . 62
An Example of Looping over Multiple Screens . . . 63
Performance Considerations . . . 69

6 Logon Components, Connections, and Connection Pools . . . 71
HP3000 Session Performance . . . 71
When Will I Need Logon Components? . . . 71
Connection Pool Architecture . . . 72
The Logon Connection’s Role in Pooling . . . 74
How Many Pools Do I Need? . . . 74
Pieces Required for Pooling . . . 75
How Do I Implement Pooling? . . . 75
The HP3000 Logon Component . . . 75
Logon, Keep Alive, and Logoff Actions . . . 76
Logon Actions . . . 76
Keep Alive Actions . . . 78
Logoff Actions . . . 79
Logon Component Life Cycle . . . 79
The HP3000 Connection . . . 80
Connection Pooling with a Single Sign-On . . . 82
Creating a Connection Pool . . . 82
Overview . . . 82
Creating a Basic Connection . . . 83
Creating a Logon Component . . . 83
Creating a Logon Connection using a Pool Connection . . . 84
Creating a Logon Connection using a Session Connection . . . 88
Creating an HP3000 Component That Uses Pooled Connections . . . 89
Managing Pools . . . 91
Connection Pool Management and Deployed Services . . . 93
Connection Discard Behavior . . . 93
Screen Synchronization . . . 93

A Glossary . . . 95

B HP3000 Keyboard Equivalents . . . 97

C HP3000 Display Attributes . . . 101

D Reserved Words . . . 103

E Java Code Pages . . . 105
About Encodings . . . 105

About This Book

Purpose

The guide describes how to use exteNd Composer HP3000 Connect, referred to as the HP3000 Component Editor. The HP3000 Component Editor is a separately-installed component editor in exteNd Composer.

Audience

The audience for the guide is developers and system integrators using exteNd Composer to create services and components which integrate HP3000 applications.

Prerequisites

The guide assumes the reader is familiar with and has used exteNd Composer's development environment and deployment options. You must also have an understanding of the HP3000 environment and building or using applications utilizing HP3000 terminals (e.g. HP 700/92).

Additional documentation

For the complete set of Novell exteNd Composer documentation, see the Novell Documentation Web Site (**http://www.novell.com/documentation-index/index.jsp**).

Organization

The guide is organized as follows:

Chapter 1, *Welcome to exteNd Composer and HP3000*, gives a definition and overview of the HP3000 Component Editor.

Chapter 2, *Getting Started with the HP3000 Component Editor*, describes the necessary preparations for creating an HP3000 component.

Chapter 3, *Creating an HP3000 Component*, describes the parts of the component editor.

Chapter 4, *Performing HP3000 Actions*, describes how to use the basic HP3000 actions, as well as the unique drag-and-drop conventions of HP3000 Connect.

Chapter 5, *Advanced HP3000 Actions,* discusses techniques for solving common HP3000 computing problems in the context of an Action Model.

Chapter 6, *Logon Components, Connections, and Connection Pools,* describes how to enhance performance through use of shared connections.

Appendix A, is a glossary.

Appendix B, *HP3000 Keyboard Equivalents,* recognized and /or used by HP3000 Connect.

Appendix C, *HP3000 Display Attributes*, and their display significance along with a discussion of how to use the `getattribute( )`.

Appendix D, *Reserved Words*, lists those words used only for HP3000 Connect.

Appendix E, *Java Code Pages*, provides reference information on character encoding conversions.

Conventions Used in the Guide

The guide uses the following typographical conventions.

Bold typeface within instructions indicate action items, including:

- Menu selections
- Form selections
- Dialog box items

Sans-serif bold typeface is used for:

- Uniform Resource Identifiers
- File names
- Directories and partial pathnames

Italic typeface indicates:

- Variable information that you supply
- Technical terms used for the first time
- Title of other Novell publications

`Monospaced` typeface indicates:

- Method names
- Code examples
- System input
- Operating system objects

1 Welcome to exteNd Composer and HP3000 Connect

Before You Begin

Welcome to the HP3000 Connect Guide. This Guide is a companion to the *exteNd Composer User's Guide*, which details how to use all the features of exteNd Composer, except for the Connect Component Editors. If you haven't looked at the *Composer User's Guide* yet, please familiarize yourself with it before using this Guide.

exteNd Composer provides separate Component Editors for each Connect. The special features of each component editor are described in separate Guides like this one.

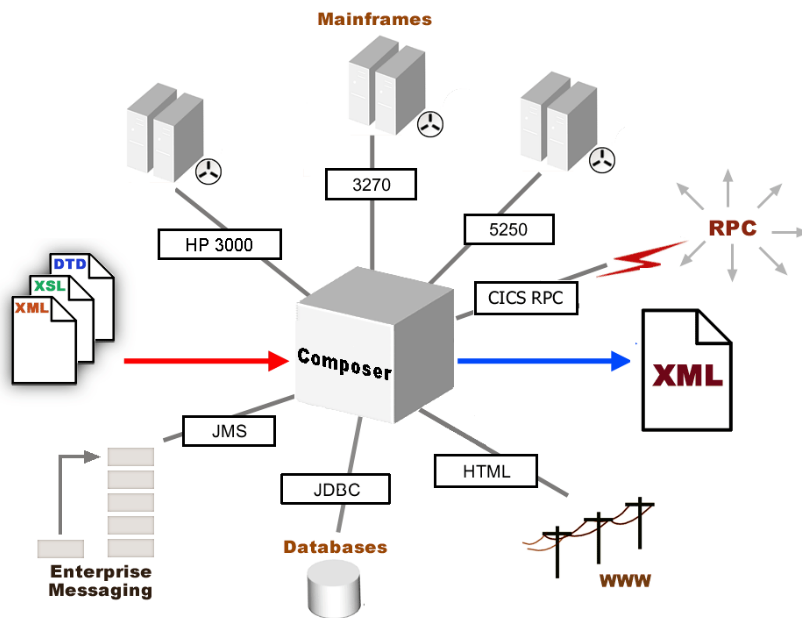
If you have been using exteNd Composer, and are familiar with the XML Map Component Editor, then this Guide should get you started with the HP3000 Component Editor.

Before you can begin working with the HP3000 Connect you must have installed it into your existing exteNd Composer. Likewise, before you can run any Services built with this Connect in the exteNd Composer Enterprise Server environment, you must have already installed the server-side software for this Connect into Composer Enterprise Server.

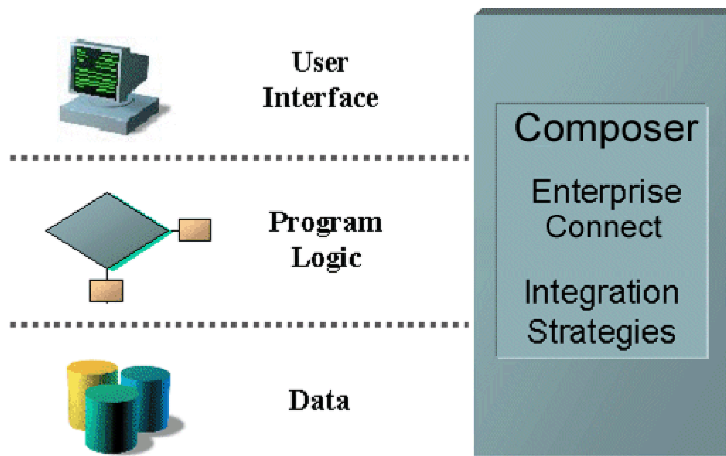
NOTE: To be successful with this Component Editor, you must be familiar with the HP3000 environment and the particular applications that you want to XML-enable.

About exteNd Composer Connects

exteNd Composer is built upon a simple hub and spoke architecture. As depicted in the graphic below, the hub is a robust XML transformation engine that accepts requests via XML documents, performs transformation processes on those documents and interfaces with XML-enabled applications, and returns an XML response document. The spokes, or Connects, are plug-in modules that "XML-enable" sources of data that are not XML aware, bringing their data into the hub for processing as XML. These data sources can be anything from legacy COBOL/applications to Message Queues to HTML pages.



exteNd Composer Connects can be categorized by the integration strategy each one employs to XML-enable an information source. The integration strategies are a reflection of the major divisions used in modern systems designs for Internet-based computing architectures. Depending on your B2B needs and the architecture of your legacy applications, exteNd Composer can integrate your business systems at the User Interface, Program Logic, or Data levels. (See below.)



What Is HP3000 Terminal?

The HP3000 Connect XML-enables HP3000 and HP/UX terminal-based applications using the User Interface integration strategy by hooking into the terminal data stream.

Many applications have been developed for HP3000 terminal based systems. These systems allow remote execution of their interface through the Telnet or NSVT protocols. Host screens can be sent to a client and keyed data can be accepted from the client. This interaction, through a so-called “dumb” terminal, means that all the data is processed on the host computer. HP3000 terminal emulation software can be used to make a microcomputer or PC act as if it were an HP3000-type terminal while it is communicating with a host computer.

Using the HP3000 Connect , you can make legacy applications and their business logic available to the internet, extranet, or intranet processes. The HP3000 Connect Component Editor allows you to build Web Services by simply navigating through an application as if you were at a terminal session. You will use XML documents to drive inquiries and updates into the screens rather than keying, use the messages returned from application screens to make the same decisions as if you were at a terminal, and move data and responses into XML documents that can be returned to the requestor or continue to be processed. The HP3000 terminal screens appear in the Native Environment Pane of the HP3000 Component Editor.

About exteNd Composer's HP3000 Component

Much like the XML Map component, the HP3000 Component is designed to map, transform, and transfer data between two different XML templates (i.e., request and response XML documents). However, it is specialized to make a connection (via TCP/IP) to a host application, process the data using elements from a screen, and then map the results to an output DOM. You can then act upon the output DOM in any way that makes sense for your integration application. In essence, you're able to capture data from, or push data to, a host system without ever having to alter the host system itself.

An HP3000 Component can perform simple data manipulations, such as mapping and transferring data from an XML document into a host program, or perform "screen scraping" of an HP3000 terminal program, putting the harvested data into an XML document. An HP3000 Component has all the functionality of the XML Map Component and can process XSL, send mail, and post and receive XML documents using the HTTP protocol.

What Applications Can You Build Using the HP3000 User Interface Component Editor?

exteNd Composer, and consequently the HP3000 Connect, can be applied to the the following types of applications:

- Business to Business Web Service interactions such as supply chain applications.
- Consumer to Business interactions such as self-service applications from Web Browsers.
- Enterprise Application Integrations where information from heterogeneous systems is combined or chained together.

Fundamentally, the HP3000 Component Editor allows you to extend any XML integration you are building to include any of your business applications that support HP3000-based terminal interactions (See the *exteNd Composer User's Guide* for more information.)

For example, you may have an application that retrieves a product's description, picture, price, and inventory from regularly updated databases and displays it in a Web browser. By using the Tandem Component Editor, you can now get the current product information from the operational systems and the static information (e.g., a picture) from a database and merge the information from these separate information sources before displaying it to a user. This provides the same current information to both your internal and external users.

2 Getting Started with the HP3000 Component Editor

Steps Commonly Used to Create an HP3000 Component

While there are many ways to go about creating HP3000 Components, the most commonly used steps in creating a simple component are as follows:

- Create XML Template(s) for the program.
- Create an HP3000 Connection Resource.
- Create an HP3000 Component.
- Enter Record mode and navigate to the program using terminal emulation available via the component editor's Native Environment Pane.
- Drag and drop input-document data into the screen as needed.
- Drag and drop screen results into the output document.
- Stop recording.

In this chapter, we'll focus on the first two steps: creating your XML Templates and creating and a configuring an HP3000 Connection Resource. Both of these are important first steps in being able to use HP3000 Components.

Templates for Your Component

Although it is not strictly necessary to do so, your HP3000 Component may require you to create XML templates so that you have sample documents for designing your component. (For more information, see Chapter 5, "Creating XML Templates," in the *exteNd Composer User's Guide*.)

In many cases, your input documents will be designed to contain data that a terminal operator might type into the program interactively. Likewise, the output documents are designed to receive data returned to the screen as a result of the operator's input. For example, in a typical business scenario, a terminal operator may receive a phone request from a customer interested in the price or availability of an item. The operator would typically query the host system via his or her HP3000 terminal session by entering information (such as a part number) into a terminal when prompted. A short time later, the host responds by returning data to the terminal screen, and the operator relays this information to the customer. This session could be carried out by an exteNd Composer Web Service that uses an HP3000 Component. The requested part number might be represented as a data element in an XML input document. The looked-up data returned from the host would appear in the component's *output* document. That data might in turn be output to a web page, or sent to another business process as XML, etc.

NOTE: Your component design may call for other xObject resources, such as custom scripts or Code Table maps. If so, it is also best to create these objects before creating the HP3000 Component. For more information, see the *exteNd Composer User's Guide*.

Creating an HP3000 Connection Resource

Once you have the XML templates in place, your next step will be to create a Connection Resource to access the host program. If you try to create an HP3000 Component in the absence of any available Connection Resources, a dialog will appear, asking if you wish to create a Connection Resource. By answering Yes to this dialog, you will be taken to the appropriate wizard.

About Connection Resources

When you create a Connection Resource for the HP3000 Component, you will have two choices: a straight “HP3000 Connection” and an “HP3000 Logon Connection.” Generally speaking, you will use the straight HP3000 Connection to connect to your host environment. The Logon Connection is used for connection pooling, which will be explained in greater detail in Chapter 6 of this Guide.

You will use a live HP3000 Connection to connect to a host environment of your choice. After setting up your Connection Resource, it will be available for use by any number of HP3000 Components that might require a connection to the host in question.

➢ **To create an HP3000 Connection Resource:**

1 From the Composer **File** menu, select **New**, then **xObject**, then open the **Resource** tab and select **HP3000 Connection**.

NOTE: Alternatively, under **Resource** in the Composer window category pane, you can highlight **Connection**, click the right mouse button, then select **New**.

The **Create a New Connection Resource** Wizard appears.

Create a New Connection Resource

A Connection resource is used to establish communications with an Connector data source or with a server using HTTP authentication. You need to create connections for each type of data source or each HTTP server you wish to communicate with. Enter a name and, optionally, a description for this Connection. The name will appear in the Composer Detail Pane and in choice lists when you are prompted for objects in Composer. The name may not contain the characters: \ / : ? " < > . | Names are case insensitive.

Name:

HP3000Sample

Description:

Purpose:
Input:
Output:
Remarks:

Help Back Next Cancel

2 Type a **Name** for the connection object.

3 Optionally, type **Description** text.

4 Click **Next**. The second panel of the wizard appears.

Create a New Connection Resource

Specify the URL for the HP3000 host. The HP3000 Port (normally 23) needs to be set to the host's requirements unless using NSVT in which case it is ignored. Select a Terminal Type used during HP3000 negotiation. USERID and PASSWORD are available for mapping in ECMAScript expressions. You may create more than one HP3000 Connection. Checking 'Default' makes this Connection the initial selection when creating a HP3000 Component. Use the Test button to check your connection.

Connection Type: HP3000 Connection
Host or IP Address: myHP3000
Telnet Port: 23
Terminal Type: HP 700/92
Code Page: ISO-Latin-1
User ID: theboss
Password: *****
Use NSVT
Autowrap Characters
Test
Default
Help Back Finish Cancel

5 Select the **HP3000 Connection** type from the pulldown menu. The dialog changes appearance to show just the fields necessary for creating the HP3000 connection.

6 In the **Host or IP Address** field, enter the physical (IP) address or hostname alias for the machine to which you are connecting.

7 In the **Telnet Port** field, enter the number of the TCP/IP port. The default port number is 23.

8 In the **Terminal Type** field, enter the type of terminal you wish to specify when handshaking with the host. Select one of the values in the pulldown menu (currently HP 700/92 or HP 700/96).

9 In the **Code Page** field, specify a code page (See “About Code Page Support on page 22.”

10 Enter a **UserID** and **Password**. These are not actually submitted to the host during the establishment of a connection. They are simply defined here. (The Password is encrypted.) Right-mouse-click and choose Expression if you want to make these fields expression-driven. See discussion further below.

NOTE: After you’ve entered UserID and Password info in this dialog, the ECMAScript global variables USERID and PASSWORD will point to these values. You can then use these variables in Send Buffer expressions (or as described under “Native Environment Pane Context Menu” on page 27).

11 Check **NSVT** to use the NSVT protocol instead of Telnet. When this option is chosen, the Port number will be ignored and the connection will be made using NSVT port 1570.

12 Check **Autowrap Characters** if you want the cursor to move to the first position of the next line as soon as the cursor reaches the end of the line.

13 Check **8-bit Data Characters** if the communicated data is in 8-bit character format. For 7-bit characters, uncheck this option and the 8th bit will be truncated.

14 Check **Backspace Send Delete** if you want the BACKSPACE key to delete data.

15 Check **Terminal New Line** if the ENTER key should generate a carriage return/line feed combination.

16 Check **HP Return Equals Enter** to generate an Enter Key when pressing the Return Key.

17 If the system expects an answerback, fill in an **HP AnswerBack Message**. This can be entered either as a constant or an expression (see discussion about Constants and Expressions below).

18 Click the **Default** check box if you'd like this particular HP3000 connection to become the default connection for subsequent HP3000 Components.

19 Click **Finish**. The newly created resource connection object appears in the Composer Connection Resource detail pane.

Code Page Support

Code Page support in exteNd Composer Connection Resources allows you to specify which Character Encoding scheme to use when translating characters sent between exteNd Composer and other host systems. exteNd Composer data uses Unicode character encoding (the Java and XML standard). Existing legacy and other host systems use a variety of character encoding schemes (i.e., Code Pages) specific for their language or usage. A mechanism is needed to translate the character encoding between these systems if they are to communicate with one another. This is handled in exteNd Composer by specifying the Code Page used by a host system in the Connection Resource.

About Constant and Expression Driven Connections

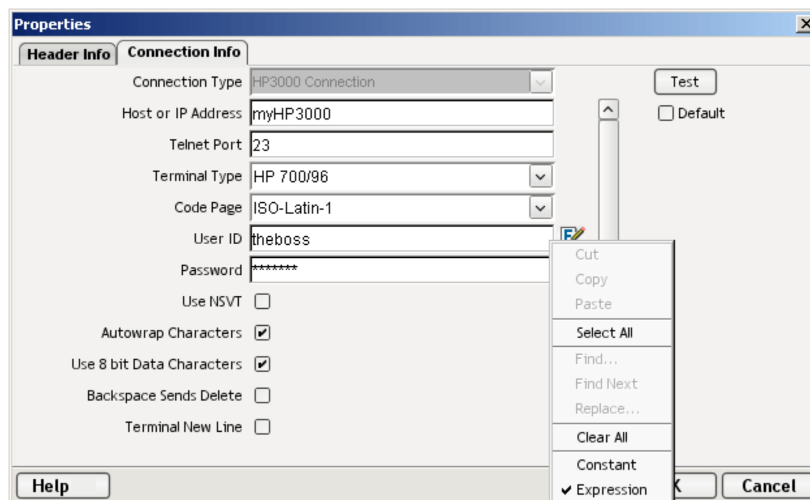
You can specify Connection parameter values in one of two ways: as Constants or as Expressions. A *constant-based* parameter uses the static value you supply in the Connection dialog every time the Connection is used. An *expression-based* parameter allows you to set the value in question using a programmatic expression (that is, an ECMAScript expression), which can result in a *different* value each time the connection is used at runtime. This allows the Connection's behavior to be flexible and vary based on runtime conditions.

For instance, one very simple use of an expression-driven parameter in an HP3000 Connection would be to define the User ID and Password as PROJECT Variables (e.g.: PROJECT.XPath("USERCONFIG/MyDeployUser"). This way, when you deploy the project, you can update the PROJECT Variables in the Deployment Wizard to values appropriate for the final deployment environment. At the other extreme, you could have a custom script that queries a Java business object in the Application Server to determine what User ID and Password to use.

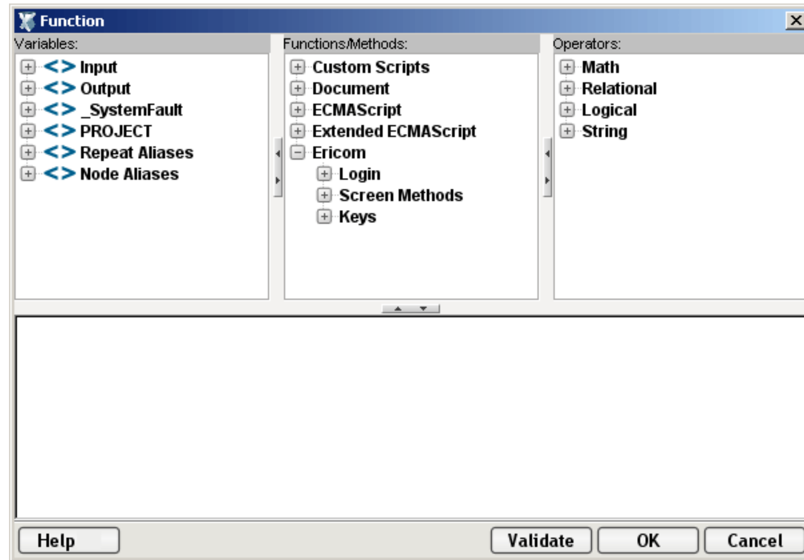
➢ **To switch a parameter from Constant-driven to Expression-driven:**

1 Click the right mouse button in the parameter field you are interested in changing.

2 Select **Expression** from the context menu and the editor button will appear or become enabled. See below.



3 Click on the **Expression Editor button**. The Expression Editor appears.



4 Create an expression (optionally using the pick lists in the upper portion of the window) that evaluates to a valid parameter value at runtime.

5 Click **OK**.

3 Creating an HP3000 Component

Before Creating an HP3000 Component

As with all exteNd Composer components, the first step in creating an HP3000 component—assuming a Connection Resource is available—is to prepare any XML templates needed by the component. (For more information, see “Creating a New XML Template” in the *Composer User's Guide*.) During the creation of your component, you will use these templates’ sample documents to represent the inputs and outputs processed by your component.

Also, as part of the process of creating an HP3000 component, you must specify an HP3000 connection for use with the component (or you can create a new one). See the previous chapter for information on creating HP3000 Connection Resources.

➤ **To create a new HP3000 Component:**

1 Select **File>New>xObject**, open the **Component** tab and select **HP3000**.

NOTE: Alternatively, under **Component** in the Composer window category pane you can highlight **HP3000 Terminal**, click the right mouse button, then select **New**.

2 The “Create a New HP3000 Component” Wizard appears.

Create a New HP3000 Terminal Component

A HP3000 Component connects to a host via the HP3000 protocol, processes data using elements from a DOM, and maps the results to an output DOM. Use this wizard to create a HP3000 Component. Enter a Name and Description for this component. The name will appear in the Composer window and in choice lists when you are prompted for objects of this type as you work in Composer. The Name is required, is case sensitive, and may not contain the characters: \ / : ? " < > . |

Name:

SampleHP3000Component

Description:

Purpose:
Input:
Output:
Remarks:

Help | Back | Next | Cancel

3 Enter a **Name** for the new HP3000 Component.

4 Optionally, type **Description** text.

5 Click **Next**. The XML Input/Output Property Info panel of the New HP3000 Component Wizard appears.

Create a New HP3000 Terminal Component

Specify one or more XML Templates to help design Input to this Component or Web Service and only one to design Output. The sample XML Documents in each Template are design time aids to help you build Action Models for the component. The samples are not actually used at runtime after deployment to your application server. The Identifier is fixed and represents the name used to refer to the XML Document during component execution. Selecting System {ANY} allows you to use an empty template (i.e. accept any document as Input).

Input Message

Part	Template Category	Template Name
Input	{System}	{ANY}

Add · Delete

Output Message

Part	Template Category	Template Name
Output	{System}	{ANY}

Add · Delete

Help · Back · Next · Cancel

6 Specify the Input and Output templates as follows.

- Type in a name for the template under **Part** if you wish the name to appear in the DOM as something other than "Input".
- Select a **Template Category** if it is different than the default category.
- Select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and choose a **Template Category** and **Template Name** for each.
- To remove an input XML template, select an entry and click **Delete**.

7 Select an XML template for use as an Output DOM using the same steps outlined above.

NOTE: You can specify an input or output XML template that contains no structure by selecting {System}{ANY} as the Input or Output template. For more information, see "Creating an Output DOM without Using a Template" in the User's Guide.

8 Click **Next**. The XML Temp/Fault Template Info panel of the New HP3000 Terminal Component Wizard appears.

Create a New HP3000 Terminal Component

Specify one or more Temp and Fault XML Templates to help design temporary parts and fault handling for this Component or Web Service. Use Temp documents for creating intermediate results or holding values for reference. Specify XML Templates to serve as Fault documents to be passed back to clients under error conditions.

Temp Message

Part	Template Category	Template Name

Add · Delete

Fault Message

Part	Template Category	Template Name
_SystemFault	{System}	{Fault}

Add · Delete

Help · Back · Next · Cancel

9 If desired, specify a template to be used as a scratchpad under the "Temp Message" pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Specify the templates as indicated in Step 6 above.

10 Under the “Fault Message” pane, select an XML template to be used to pass back to clients when an error condition occurs.

11 As above, to add additional temp or fault XML templates, click **Add** and choose a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an XML template, select an entry and click **Delete**.

12 Click **Next**. The Connection Info panel of the Create a New HP3000 Component Wizard appears.

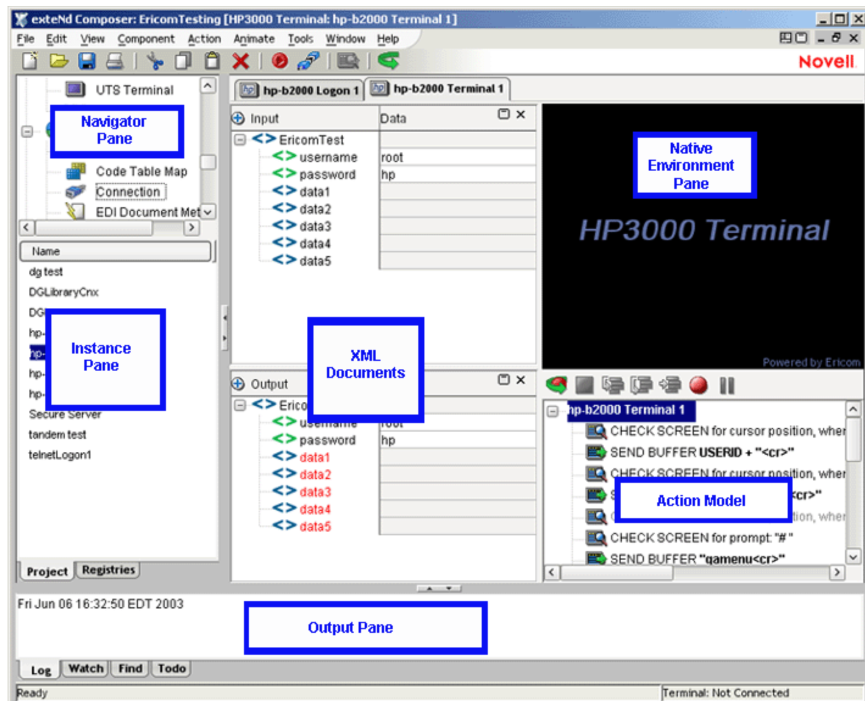
13 Select a **Connection** name from the pulldown list. For more information on the HP3000 Connection, see “Creating a Connection Resource” in Chapter 2 of this Guide.

14 Click **Finish**. The component is created and the HP3000 Component Editor appears.

About the HP3000 Component Editor Window

The HP3000 Component Editor includes all the functionality of exteNd Composer’s XML Map Component Editor. For example, it contains mapping panes for Input and Output XML documents as well as an Action pane.

There is one main difference, however. The HP3000 Component Editor also includes a Native Environment Pane featuring an HP3000 emulator. This screen appears black until you either click the Connection icon in the main toolbar or begin recording by clicking the Record button in the toolbar. Either action establishes an HP3000 emulation session inside the Native Environment Pane with the host that you specified in the connection resource used by this HP3000 component.



About the HP3000 Native Environment Pane

The HP3000 Native Environment Pane provides HP3000 emulation of your host environment. From this pane, you can execute an HP3000 session in real time, interacting with the Native Environment Pane exactly as you would with the screen on a “dumb terminal.” You can also do the following:

- Use data from an Input XML document (or other available DOM) as input for an HP3000 screen field. For example, you could drag a SKU number from an input DOM into the “part number” field of an HP3000 screen, which would then query the host and return data associated with that part number, such as description and price.
- Map the data from the returned HP3000 screen and put it into an Output XML document (or other available DOM, e.g., Temp, MyDom, etc.).
- Map header and detail information (such as a form with multiple line items) from the Native Environment Pane to an XML document using an ECMAScript expression or function.

About HP3000 Keyboard Support

The HP3000 Native Environment Pane supports the use of numerous special terminal keys. The Terminal Keypad dialog (see below) is comprised of four Tabs: Common Keys, NumPad Keys, Other Keys and North Keys. Each Tab contains a group of keys with specific functionality.

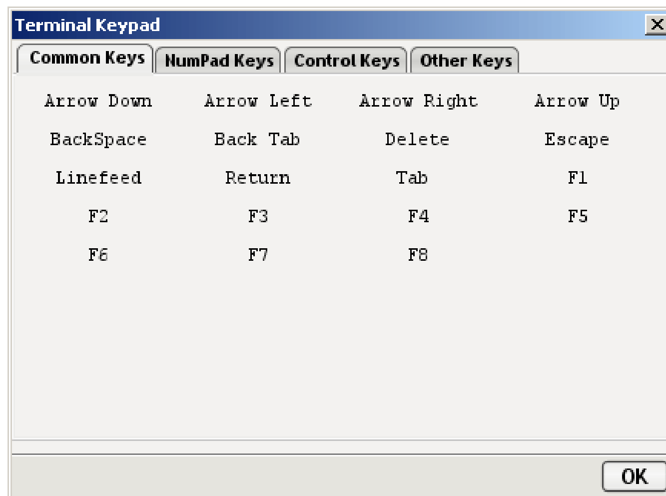
Note that you can also achieve the use of additional keys (such as Insert Char and Delete Char) by using the picklists in the Expression Builder dialog, Function/Methods column, under HP3000 > Keys.

How to Use the Floating Keypad:

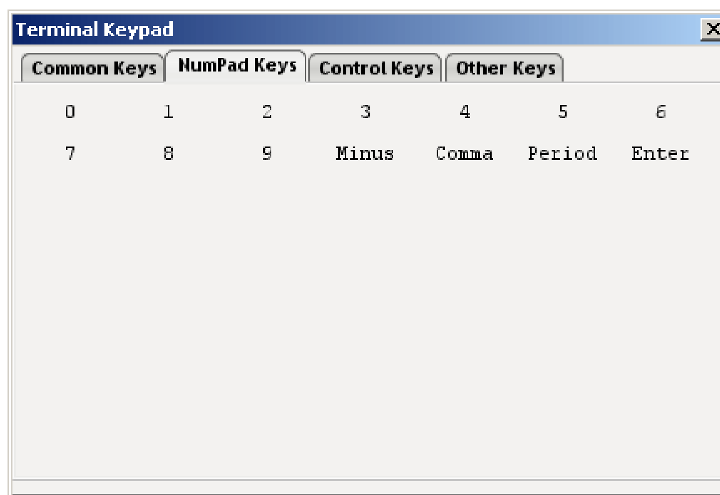
1. Select **View/Terminal Keypad** from the Composer Menu. A floating Keypad appears. The Keypad window contains a series of tabs, including the following: Common Keys, NumPad Keys, Control Keys and Other Keys.
2. Click on the appropriate Tab to display the keys you wish to view on the Terminal Keypad.
3. Click on the key you wish to invoke. If you require help, hover the mouse over that key. Help will display the HP3000 keyboard equivalent for that key. You will see the result of the key you clicked in the Native Environment Pane.
4. Click **OK** to close the keypad. In order for the keypad to redisplay, you must repeat step 1. When you display the keypad, you will return to the last Tab that you were using.

The following pages illustrate the four Tabs and corresponding keys that can be used to interact with HP3000.

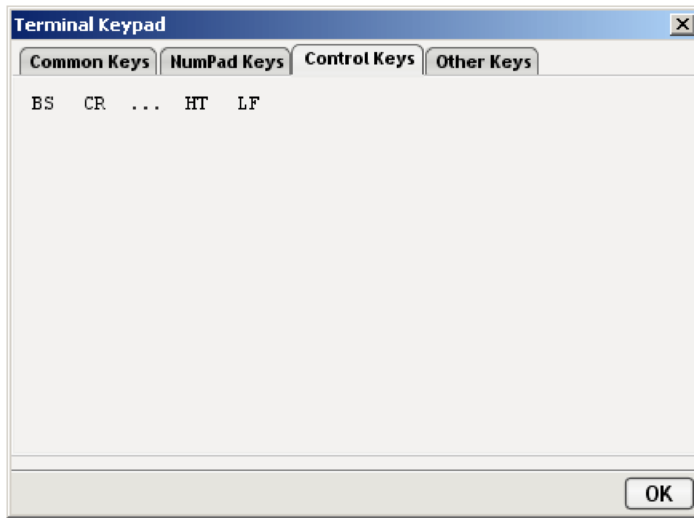
Common Keys: Includes directional keys, (Arrow Down, Arrow Left, Arrow Right, Arrow Up, BackSpace, BackTab) as well as Delete, Escape, Linefeed, Return, and Tab. The function keys, F1 through F8, are also displayed.



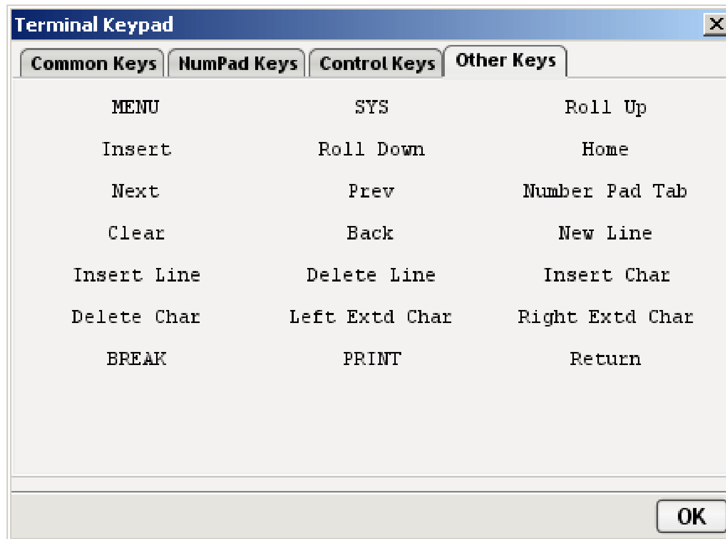
NumPad Keys: Includes the digits 0-9, Minus, Comma, Period and Enter keys.



Control Keys: Includes 5 keys associated with specific functions. Refer to Appendix B for a complete listing.



Other Keys: Includes keys to perform common functions for example: the Home key.



NOTE: The complete list of special (non-printing) keys and their ANSI equivalents is shown in Appendix B.

About the Screen Object

The Screen Object is a byte-array representation of the emulator screen shown in the Native Environment Pane, with methods for manipulating the screen contents.

What it is

The HP3000 component communicates with the host environment via a character-mode terminal data stream, in a TCP/IP *session*. The user sends data to the host in the form of keystrokes (or XML data mapped to cursor prompts). The host, in turn, sends the terminal a stream of data which may contain anything from a single byte to a whole screen's worth of information. The Screen Object represents the current screen's worth of data. For a 24 x 80 terminal screen, this is 1,920 bytes of data.

How it works

When character data arrive from the host, appropriate updates to the Native Environment Pane occur in real time. Those updates might be anything from a simple cursor repositioning to a complete repaint of the terminal screen. The screen content is, in this sense, highly dynamic.

When you have signaled exteNd Composer (via a Check Screen action) that you wish to operate on the current screen's contents, the screen buffer is packaged into a *Screen Object* that is made accessible to your component through ECMAScript.

Many times, it is not necessary for your component to "know" or understand the complete screen contents prior to sending keystrokes back to the host or prior to mapping data into a prompt. But when mapping outbound from the screen to a DOM, it can be useful to have programmatic access to the Screen Object. To make this possible, the Connect for HP3000 defines a number of ECMAScript extensions for manipulating screen contents. These extensions are described in further detail in the next chapter. For now, a simple example will suffice. Suppose you are interested in obtaining a string value that occurs on the screen in row 5 at column position 20. If the string is 10 characters long, you could obtain its value by using the following ECMAScript expression as the Source in a Map action (with an output DOM or temp DOM as the Target):

```
Screen.getTextAt( 5, 20, 10 )
```

The 10 characters beginning at row 5, column 20 on the screen would be mapped to the Target of the Map action.

For more examples (and complete API documentation for the Screen object), see the section on "HP3000-Specific Expression Builder Extensions" in the next chapter.

HP3000-Specific Toolbar Buttons

If you are familiar with exteNd Composer, you will notice immediately that the HP3000 Connect includes a number of Connect-specific tool icons on the component editor's main toolbar. They appear as shown below.

Record Button

Record icon (normal state)

Record icon (recording in progress)

Record icon (disabled)

The Record button allows you to capture keyboard and screen manipulations as you interact with the Native Environment Pane. Recorded operations are placed in the Action Model as actions, which you can then "play back" during testing.

Connection Button

Connection (disconnected state)

Connection (connected state)

Connection (connected/disabled state)

The Connection button on Composer's main toolbar toggles the connection state of the component (using settings you provided during the creation of the Connection Resource associated with the component).

NOTE: When you are recording or animating, a connection is automatically established, in which case the button will be shown in the "connected/disabled" state. When you turn off recording, the connection the button will return to the enabled state.

Create Check Screen Button

The Create Check Screen button on Composer's main toolbar should be clicked before the first user interaction with any given terminal screen. This signals exteNd Composer that you intend to work with the screen data as currently shown in the Native Environment Pane. Clicking this button causes a new Check Screen Action to be inserted into the Action Model. (See the next chapter for a detailed discussion of this action type.)

HP3000-Specific Menu Bar Items

Component Menu

Two additional items have been added to the Component drop down menu for the HP3000 Connect. These are Start/Stop Recording and Connect/Disconnect (depending on your current status).

Start/Stop Recording—This menu option manages the automatic creation of actions as you interact with a host program. **Start** will enable the automatic creation of actions as you interact with the screen and **Stop** will end action creation.

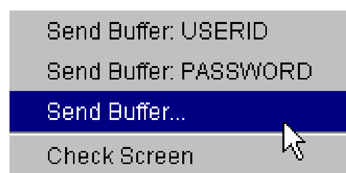
Connect/Disconnect—This menu option allows you to control the connection to the host. When you are recording or animating, a connection is automatically established (and consequently, the connection icon is shown in the "connected/disabled" state). However, this button is useful if you are *not* recording and you merely want to establish a connection for the purpose of navigating the HP3000 environment.

HP3000-Specific Context-Menu Items

The HP3000 Connect also includes context-menu items that are specific to this Connect. To view the context menu, place your cursor in the appropriate pane (Native Environment or Action) and click the right mouse button.

Native Environment Pane Context Menu

When you right-mouse-click in the Native Environment Pane, you will see a contextual menu. The menu items will be greyed out if you are not in record mode. In record mode, the context menu has the following appearance:



The four commands work as follows:

Send Buffer: USERID—Automatically sends User ID information to the host, based on the value you supplied (if any) for User ID in the HP3000 Connection Resource for this component. Also creates the corresponding Send Buffer action in the Action Model.

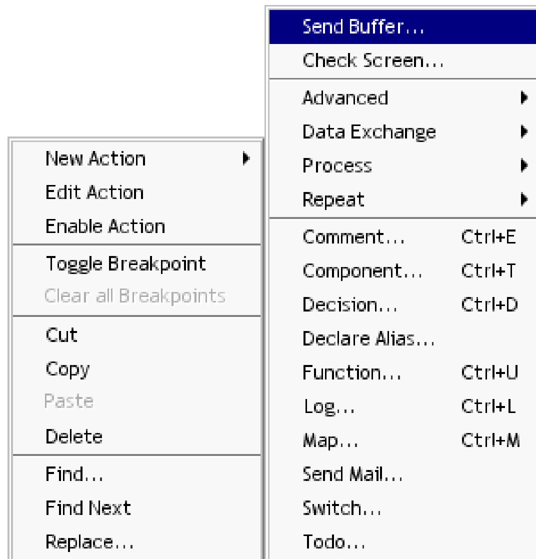
Send Buffer: PASSWORD—Automically transmits Password information to the host, based on the Password you supplied (if any) in the HP3000 Connection Resource for this component. Also creates the corresponding Send Buffer action in the Action Model.

Send Buffer—Brings up the Send Buffer dialog, allowing you to create a new Send Buffer Action. (See the next chapter for a detailed discussion of the use of this command.)

Check Screen—Creates a new Check Screen action without bringing up a dialog (same as a click on the Create Check Screen button in the toolbar).

Action Pane Context Menu

If you click the right mouse button when the mouse is located anywhere in the Action pane, a context menu appears as shown.



The HP3000-specific functions of the context menu items are as follows:

Send Buffer—Allows you to create a Send Buffer action, so you can enter text and/or control-key commands that will be sent to the HP3000 host application. (See the next chapter for a detailed discussion of the use of this command.)

Check Screen— Allows you to create a new Check Screen action which is used to make sure the appropriate screen is present before the component continues processing. A dialog appears, allowing you to specify various go-ahead criteria as well as a Timeout value. (The next chapter contains a detailed discussion of the Check Screen action.)

4 Performing Basic HP3000 Actions

About Actions

An *action* is similar to a programming statement in that it takes input in the form of parameters and performs specific tasks. Please see the chapters in the *Composer User's Guide* devoted to Actions.

Within the HP3000 Component Editor, a set of instructions for processing XML documents or communicating with non-XML data sources is created as part of an Action Model. The Action Model performs all data mapping, data transformation, data transfer between hosts and XML documents, and data transfer within components and services.

An Action Model is made up of a list of actions that work together. As an example, one Action Model might read invoice data from a disk, retrieve data from a host inventory database, map the result to a temporary XML document, make a conversion, and map the converted data to an output XML document.

The Action Model mentioned above would be composed of several actions. These actions would:

- Open an invoice document and perform an HP3000 command to retrieve invoice data from a host database
- Map the result to a temporary XML document
- Convert a numeric code using a Code Table
- Map the result to an Output XML document

About HP3000-Specific Actions

The HP3000 Connect includes two actions that are specific to the HP3000 environment: Check Screen and Send Buffer.

HP3000 Action	Description
Send Buffer	Buffers a string for transmission to the host. In Field Mode and Block Mode, HP3000 Connect will act as a “dumb” terminal and might only change the internal buffers of the sreen without sending data to the host. The string is formed from Map actions and/or from user keystrokes. (The Send Buffer action can be created manually, but will more often be generated automatically when the user types into the screen or maps data to the current prompt.)
Check Screen	Allows the component to stay in sync with the host application. This action signals the component that execution must not proceed until the screen is in a particular state (which can be specified in the Check Screen setup dialog), subject to a user-specified timeout value.

The purpose of these actions is to allow the HP3000 component (running in a deployed service) to replicate, at runtime, the terminal/host interactions that occur in an HP3000 session. The usage and meanings of these actions are described in further detail below.

The Send Buffer Action

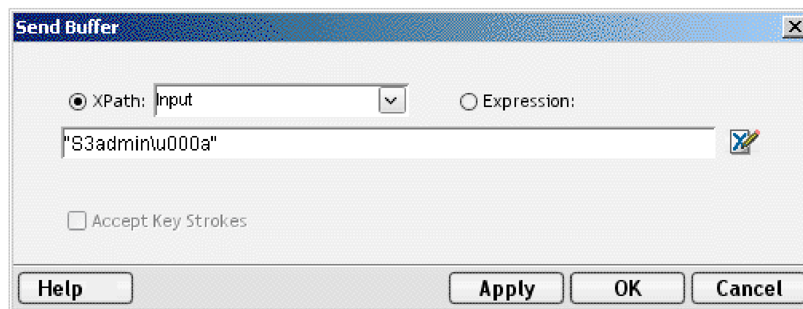
The Send Buffer action encapsulates "keystroke data" (whether actually obtained from keystrokes, or through a drag-and-drop mapping, or via an ECMAScript expression built with the Expression Builder) that will be sent to the host in a single transmission at component execution time. When the Send Buffer action executes, the buffered data are sent to the host in the form of a properly HP 700/92-escaped byte stream. Send Buffer actions should always be preceded by a Check Screen action (see next section).

The Send Buffer action can be created in several ways:

- In Record mode, just begin typing in the Native Environment Pane. Keystrokes are automatically captured to a new Send Buffer action.
- Right-mouse-click anywhere in the Action Model; a contextual menu appears. Select **New Action** and **Send Buffer**.
- In the main menu bar, under **Action**, select **New Action** and **Send Buffer**.

➢ **To create a Send Buffer action using menu commands:**

1 Right-mouse-click anywhere in the Action Model and select **New Action**, then **Send Buffer** from the contextual menu (or use the Action menu as described above). The Send Buffer dialog will appear.



2 To map a DOM element's contents to the buffer, click the **XPath** radio button, then select a DOM from the pulldown list and type the appropriate XPath node name in the text area (or click the Expression icon at right and build the node name using the Expression Builder).

3 To specify the buffer's contents using ECMAScript, click the **Expression** radio button, then use the Expression Builder dialog to create an ECMAScript expression that evaluates to a string.

4 To specify the contents of the buffer manually (by typing a string into the text field), first check the **Accept Key Strokes** checkbox, then begin typing. The Expression radio button will become selected automatically and every key you press will be entered into a quoted string in the text area. Control keys (arrow keys, function keys, etc.) will automatically be translated to the appropriate escape sequences. (See discussion below.)

5 Click **OK**.

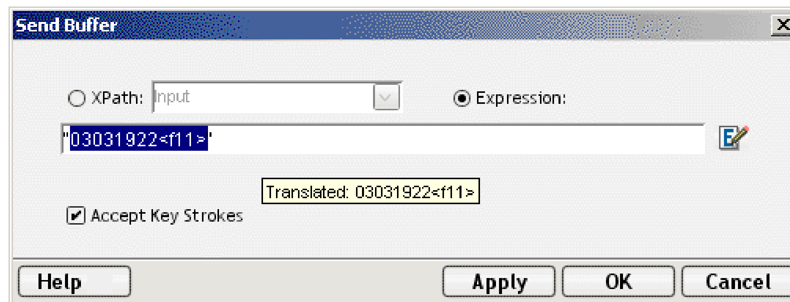
Editing Text in the Send Buffer Dialog

When you are in "Accept Key Strokes" mode, normal editing of text via backspacing, cut/paste, etc. is not possible, since every keystroke is captured to the dialog as an escaped string-literal value. For example, if you hit the backspace key, a value of "<backspace>" will be appended to the string buffer, instead of the previous character being deleted. This may not be what you want.

To edit the buffer contents directly (using cut, paste, backspace, and so on), first uncheck the Accept Key Strokes checkbox. Then edit your text. To return to key-capture mode, check Accept Key Strokes. Any additional keystrokes will then be translated to escape sequences and appended to the existing text.

On some occasions, you may wish to enter an escape value manually. You can do this by unchecking Accept Key Strokes and typing the value in question anywhere in the current text string. If you don't know the key sequence for a given control key or function key, you can find it by clicking the Expression icon to the right of the text area (which brings up the Expression Builder dialog) and then double-clicking the appropriate control-key entry in the picklist in the upper part of the Expression Builder dialog.

If you want to know what a given escape sequence means in plain English, simply select (highlight) the escape sequence(s) of interest and let the mouse hover over the selection. See below.



A hover-help box will appear, containing the escape sequence's plain-English translation. For example, in the graphic above, the key sequence "03031922< F11>" has been highlighted and the mouse is hovering over the selection. The hover-help box shows the key sequence translates to "03031922< F11>".

If a group of key sequences is selected, you will see (in the hover-help box) all character equivalents, wrapped in angle brackets.

All special (non-printing) keys and their ANSI equivalents are listed in "HP3000 Keyboard Equivalents" in Appendix B.

How Keys Are Displayed in the Action Model

When a Send Buffer action is created, the keystrokes that are captured in real time are displayed in the Action Model either as plain alphanumeric values or as a string which represents the key name in angle brackets. For example, an *up arrow* will be translated into <up> and F7 will be translated into <f7>. Backspace and delete keystrokes are also represented as strings. Therefore, if you wish to correct typos in your Send Buffer action, you may want to doubleclick the action in the Action Model (which brings up the Send Buffer dialog) and edit the buffer string by hand.

The Check Screen Action

Because of the latency involved in terminal sessions and the possibility that screen data may arrive in an arbitrary, host-application-defined order, it is essential that your component can depend on the terminal screen being in a given state before it operates on the current screen data. The Check Screen action makes it possible for your component to stay "in sync" with the host. You will manually create Check Screen actions at various points in your Action Model so that precisely the correct screens are acted on at precisely the right time(s).

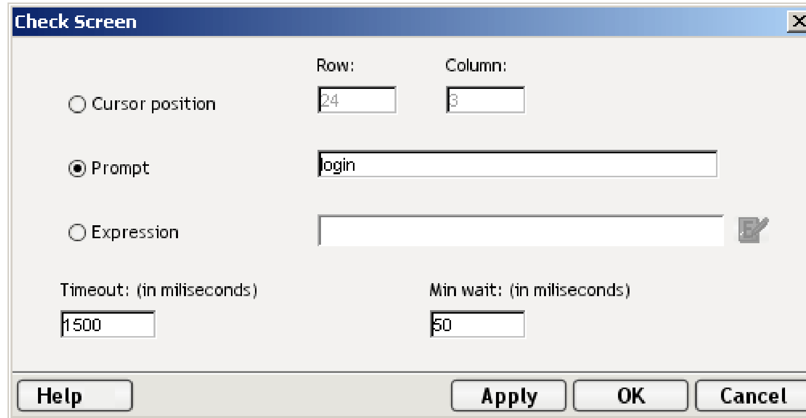
To create a new Check Screen action, you can do one of the following:

- Click on the "Create Check Screen Action" button on the main toolbar, or
- Perform a right mouse click inside the action list, then select **New Action** and **Check Screen** from the contextual menu, or
- In the component editor's main menu bar, select **Action**, then **New Action**, then **Check Screen**
- While you are in Record mode, with your cursor in the Native Environment Pane, right-click then select **Check Screen**.

NOTE: You will most often use the toolbar button when you are in Record mode.

➤ **To create a Check Screen action using a menu command:**

1. With your cursor positioned in the Action Model on the action item after which you want your new item to appear, perform a right mouse click. Then select **New Action** and **Check Screen** from the contextual menu (or use the Action menu in the main menu bar as described above). The Check Screen dialog appears.



2. Click one of the three radio buttons (**Cursor position**, **Prompt**, or **Expression**), depending on how you want to specify the go-ahead (screen readiness) criterion. (The default is "Cursor position.") See discussion below.
3. Specify a **Timeout** value in milliseconds. (See discussion further below.)
4. Specify a **Min wait** value in milliseconds. (See discussion further below.)
5. Click **OK**.

Understanding the Check Screen Action

It is important that the execution of actions in your Action Model not proceed until the host application is ready, and all screen data have arrived (that is, the screen is in a known state).

Your component must have some way of "knowing" when the current screen is ready. The Check Screen Action is how you specify the readiness criteria.

The purpose of the Check Screen Action dialog is twofold:

- It allows you to specify a wait time for program synchronization.
- It allows you to specify an expression which will be used as a criterion to judge whether the screen is in a state of readiness at execution time.

These factors are discussed in some detail below. Be sure to read and understand the following sections before creating your first HP3000 Component.

Cursor Position

You can base readiness on the location of the terminal's cursor. Simply enter the row and column number of the cursor's "prompt position." (The values shown in the Row and Column fields of the dialog will always automatically default to the cursor's current position. You will normally not have to enter the numbers manually.)

Prompt

The current prompt position can be specified on the basis of the character string that immediately precedes the cursor position in the terminal emulation window. For example, the prompt may say "Choose one: (A, B, C, D)". In this instance, you could specify "Choose one: (A, B, C, D)", or "(A, B, C, D)", or perhaps simply ")", as the go-ahead prompt. (The default value shown for the prompt string will be the current screen contents for the line in which the cursor is positioned. The default string will include all characters from the beginning of the prompt line up to and including the last space character, if any, preceding the cursor.)

Expression

It is possible that the prompt position or prompt text could vary dynamically at runtime. For the ultimate flexibility in determining the go-ahead criterion, you can click the Expression radio button in the Check Screen Action dialog and enter an ECMAScript expression in the associated text field. At runtime, if the expression evaluates as "true," the screen will be considered ready; but not otherwise.

Expressions are discussed in detail in the heading titled "HP3000-Specific Expression Builder Extensions" below.

Timeout

The timeout value (in milliseconds) represents the maximum amount of time that your component will wait for screen data to both arrive *and* meet the readiness criterion specified in the top part of the dialog. If the available screen data do not meet the readiness criteria before the specified number of milliseconds have elapsed, an exception is thrown.

NOTE: Obviously, since the latency involved in a terminal session can vary greatly from application to application, from connection to connection, or even from screen to screen, a great deal of discretion should be exercised in deciding on a Timeout value. Careful testing of the component at design time as well as on the server will be required in order to determine "safe" timeout values.

The default Timeout value will vary depending on whether you are in Record mode or you are merely creating Actions manually. In Record mode, the default Timeout value is a calculated value based on the actual time that elapses between the last operation and the loading of the new screen. (The value displayed in the dialog is twice this "observed load time," rounded up to the nearest full second.) When you are creating a Check Screen action manually (not in Record mode), the default value is 1500 milliseconds.

Min Wait

The Min Wait time (in milliseconds) represents the amount of time your component should wait before the initial check of the screen buffer. For example, if you specify a Min Wait of 500, your component will check the screen for readiness (according to the criteria you specified) after waiting 500 milliseconds. If the go-ahead criteria are met, the screen will be rechecked after another 100 milliseconds. Only if the second check is also good will execution of the component proceed. If not, the screen will be rechecked at 100-millisecond intervals until the Timeout value (above) has been reached. At that point, if the screen *still* does not meet readiness requirements, an exception is thrown.

NOTE: Every Check Screen action checks the screen a minimum of two times. Go-ahead doesn't occur unless two consecutive checks are passed.

The default value for Min Wait is 50 milliseconds. But regardless of the Min Wait time, the screen will be checked one final time at the expiration of the Timeout period, so that even if the Min Wait time is greater than the Timeout value, the screen will still be checked once.

Using Actions in Record Mode

The easiest way to create an Action Model for your component is to use Record mode. When you build an Action Model in this way, a new Send Buffer action is created for you automatically as soon as you begin typing or drag an element from the Input DOM into the appropriate field onscreen. All you need then do is wait for the next screen to arrive from the host, add a Check Screen action to make sure you are on the right screen and begin the process again, repeatedly. In this fashion, a sequence of Check Screens, Send Buffers and Check Screens can be built very quickly and naturally.

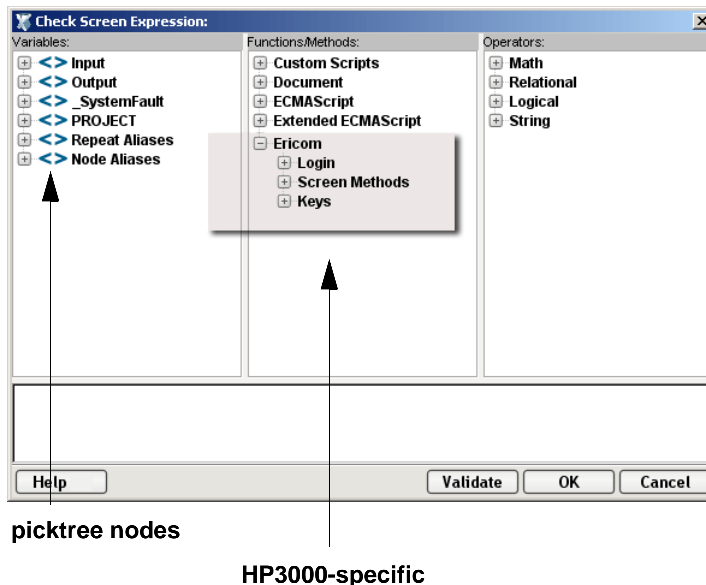
When a Send Buffer action has been created automatically for you, all of your subsequent keystrokes will be captured to the buffer until one of the following occurs:

- You perform a right-mouse-click.
- You begin to create a new action in the Action Model.
- You drag data into or out of the Native Environment Pane.
- You toggle the Record button to the non-recording state.

Working in record mode will be discussed further below in the section entitled “Recording an HP3000 Session”.

HP3000-Specific Expression Builder Extensions

The Connect for HP3000 exposes a number of HP3000-specific ECMAScript global variables and object extensions, which are visible in Expression Builder picklists. The HP3000-specific items are listed under the node labelled “HP3000.” There are three child nodes: Login, Screen Methods, and Keys. See illustration below.



Login

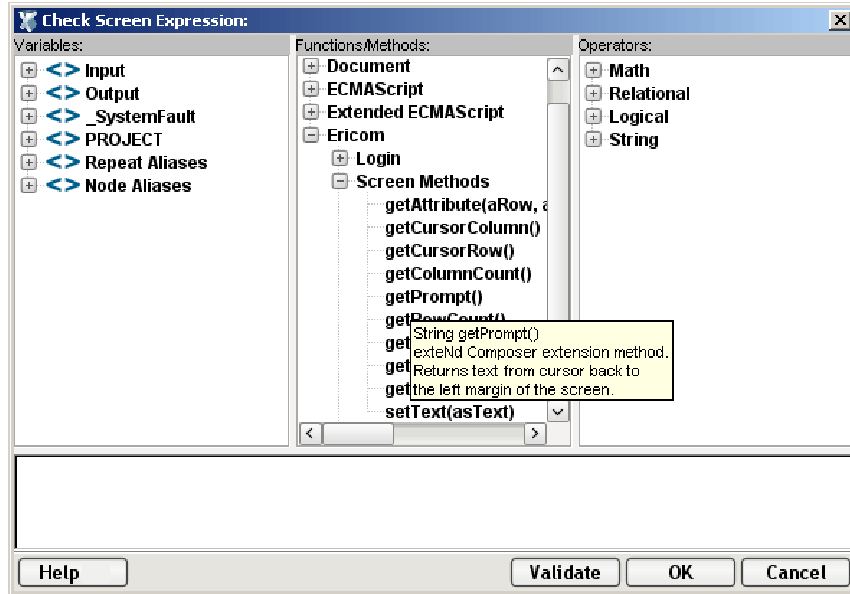
HP3000 Connection Resources have two global variables that are accessible from Expression Builder dialogs: the USERID and PASSWORD. These properties (available under the Login node of the picktree) specify the User ID and Password values that may be requested by the host system when you connect. You can map these variables into the terminal screen, which eliminates the need for typing user and password information explicitly in a map action.

NOTE: You can also create a Send Buffer action where the XPath source is defined as $PASSWORD.

Screen Methods

When an Expression Builder window is accessed from a Map or Function action in the HP3000 Component, the picklists at the top of the window expose special HP3000-specific ECMAScript extensions, consisting of various methods of the Screen object and predefined escape sequences corresponding to various “special keys” on the virtual terminal’s keyboard.

Hover-help is available if you let the mouse loiter over a given picktree item. (See illustration.)



In addition, you can obtain more complete online help by clicking Help in the lower left corner of the dialog.

The Screen object offers methods with the following names, signatures, and usage conventions:

int getAttribute(*nRow, nColumn*)

This method will return the *display attribute* value of the character at the screen position given by nRow, nColumn. The complete set of possible display attribute values is listed in Appendix C. An example of using this method is:

```
if (Screen.getAttribute( 5, 20 ) == 1) // if character at 5, 20 is bold
     // do something
```

int getCursorColumn(*void*)

This method returns the current column position of the cursor in the HP3000 terminal emulator screen (Native Environment Pane). Column positions are one-based rather than zero-based. In other words, in 24x80 mode, this method would return a value from 1 to 80, inclusive.

int getCursorRow(*void*)

This method returns the current row position of the cursor in the HP3000 terminal emulator screen (Native Environment Pane). Row positions are one-based rather than zero-based. In other words, in 24x80 mode, this method would return a value from 1 to 24, inclusive.

int getColumnCount(*void*)

This method returns the native column-width dimension of the current screen. (Due to possible mode changes in the course of host-program execution, this value can change from screen to screen. Do not depend on this value staying constant over the life of the component.) When the program is in 24x80 mode, this method will return 80. To retrieve all of the contents of row 15 of the current screen, regardless of its native dimensions, you could do:

```
var myRow = Screen.getTextAt( 15, 1, Screen.getMaxColumn() );
```

String getPrompt(*void*)

The `getPrompt()` method returns the string representing all characters in the cursor's row, starting at column 1 and continuing to, but not including, `getCursorColumn()`—in other words, everything from the beginning of the line to the cursor position. (This is the same as the default prompt string shown in the Check Screen dialog.) Example:

```
var thePrompt = Screen.getPrompt();
if (thePrompt().toLowerCase().indexOf("password") != -1)
     Screen.setText(PASSWORD);
```

int getRowCount(*void*)

This method returns the native vertical dimension of the current screen. (Due to possible mode changes in the course of host-program execution, this value can change from screen to screen. Do not depend on this value staying constant over the life of the component.) When a program is in 24x80 mode, this method will return 24. To loop over all rows of a screen, regardless of its native dimensions, you could do:

```
for (var i = 1; i <= Screen.getMaxRow(); i++)
{
      var myRow = Screen.getTextAt( i, 1, Screen.getMaxColumn() );
      // do something with myRow
}
```

String getText(*nOffset, nLength*)

This method returns the string of characters (of length `nLength`) that occurs in the Screen object at the byte offset given by `nOffset`. Note that the offset is one-based, not zero-based. Thus, to obtain *all* of a 24 x 80 screen as an ECMAScript String, you would do:

```
var wholeScreen = Screen.getText( 1, 24 * 80 );
```

Any attempt to obtain character data beyond the bounds of the screen buffer will result in an exception. For example, the following call will fail:

```
var wholeScreen = Screen.getText( 1, 1 + 24 * 80 ); // ERROR!
```

String getTextAt(*nRow, nColumn, nLength*)

This method returns an ECMAScript String that represents the sequence of characters (of length `nLength`) in the current screen starting at the row and column position specified. Note that `nRow` and `nColumn` are one-based, not zero-based. *A zero value for either of these parameters will cause an exception.*

To obtain all of row 20 of a 24x80 screen, you would do:

```
var myRow = Screen.getTextAt( 20, 1, 80 );
```

The `getTextAt()` technique is used internally in drag-and-drop Map actions involving screen selections created as described in "Selecting Continuous Data" further below.

String getTextFromRectangle(*nStartRow, nStartColumn, nEndRow, nEndColumn*)

This method returns a single String consisting of substrings (one per row) comprising all the characters within the bounding box defined by the top left and bottom right row/column coordinates specified as parameters. So for example, in 24x80 mode, you could obtain the upper left quarter of the screen by doing:

```
var topLeftQuadrant = Screen.getTextFromRectangle(1,1,12,40);
```

The `getTextFromRectangle()` method is used internally in drag-and-drop Map actions involving rectangular screen selection regions created using the Shift-selection method (see “Selecting Rectangular Regions” below).

Note that the string returned by this method contains newline (\u000a) delimiters between substrings. That is, there will be one newline at the end of each row’s worth of data. The overall length of the returned string will thus be the number of rows times the number of columns, plus the number of rows. For example, `Screen.getTextFromRectangle(1,1,4,4).length` will equal 20.

void setText(*String*)

The `setText()` method allows you to send data to the screen (and therefore the host application) programmatically, without explicitly creating a Send Buffer action. Example:

```
var myPhone = "(203) 225-1800";
if (Screen.getPrompt().indexOf("Phone") != -1)
     Screen.setText( myPhone + "\r" ); // send string + CR
```

Keys

The Keys node of the HP3000-specific picktree in the Expression Builder dialog has child nodes labelled Common Keys, NumPad Keys, Control Keys and Other Keys. These keys were discussed in detail in “About HP3000 Keyboard Support” on page -22. By double-clicking the picklist items under these categories, you can automatically generate the key string for any non-printing characters, special keys and function keys you wish to transmit to the host. The detailed contents of these picktree items can be found in Appendix B.

Screen Selections in the HP3000 Connect

There are two main ways of selecting data on the terminal screen (in the Native Environment Pane) at design time, for purposes of dragging *out*. One method selects text in a continuous stream, from one screen-buffer offset to another; the other method selects text in an arbitrary onscreen bounding box or region.

Selecting Continuous Data

When you drag across multiple rows of data without holding the Shift key down, *all* characters from the initial screen offset (at the mouse-down event) to the final screen offset (at mouse-up) are selected, as shown in the graphic below. (The selected text is “reversed out.” A partial row has been selected, followed by three complete rows, followed by a partial row.)

```
You searched for the AUTHOR: clancy tom                  CONSULS:All Locations
                                                              Record 3 of 12
AUTHOR       Clancy, Tom, 1947-
TITLE        Debt of honor / Tom Clancy.
PUBLISHER    New York : G.P. Putnam's Sons, c1994.
DESCRIPT     766 p. ; 24 cm.
NOTE         11/95, c.1 $25.95 gift.
SUBJECT      Ryan, Jack (Fictitious character) -- Fiction.
             Intelligence service -- United States -- Fiction.
ISBN         0399139540 (alk. paper) :

    LOCATION                  CALL NO.                       STATUS
1 > CCSU Stack Level 5        PS3553 L245 D43 1994           AVAILABLE
2 > SCSU Main                 PS3553.L245 D43 1994           AVAILABLE

Key NUMBER to see more information, OR
R > RETURN to Browsing                 N > NEW Search
F > FORWARD browse                     A > ANOTHER Search by AUTHOR
B > BACKWARD browse                    + > ADDITIONAL options
Choose one (1-2,R,F,B,N,A,Z,S,P,G,E,Y,+)
```

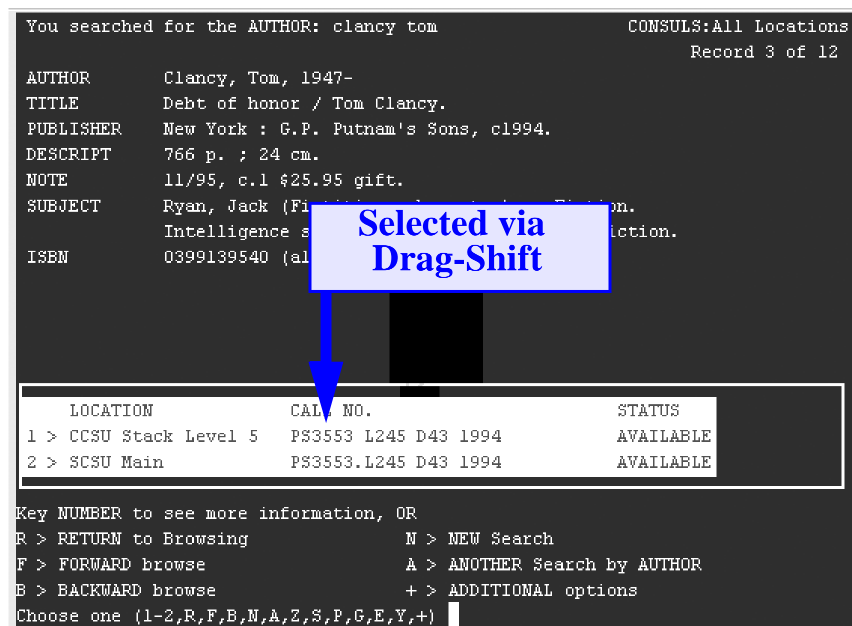
Selected via Drag

As indicated in the component editor window’s status line (lower left), the selection in the above example actually begins at row 5, column 26, and ends at row 9, column 35. If you were to drag this selection out of the Native Environment Pane, into a DOM, a Map action would be generated as follows:

Notice that the `getTextAt()` method is used. This means the captured screen characters form one string, which is mapped to **Output/Inquiry/Response/Info**. No newlines or other special characters are inserted into the string. (Areas of the screen shown in black are simply represented as space characters in the string.)

Selecting Rectangular Regions

Sometimes you may not want the selection behavior described above. In certain cases, screen data may be grouped into zones with their own natural boundaries. For example, in the screen shown previously, there is a box two-thirds of the way down the screen containing information on the availability of a given book. You may want to capture (for drag-out purposes) just the data enclosed within this particular rectangular region on the screen. To do this, first hold the Shift key down, then drag your mouse across the portion of the screen that you want to select. The selected area is highlighted and the appropriate row/column start and end points are displayed in the status line of the component editor’s window, as below:



In this instance, when you drag the rectangular highlight region out of the Native Environment Pane, into a DOM, the resulting Map action uses the `getTextFromRectangle()` method described on page 37.The resulting action looks like:

MAP **Screen.getTextFromRectangle(16,2,18,67)** TO **$Output/InquiryResponse/Status**

This method operates in a different fashion from `getTextAt()`, because the string returned by `getTextFromRectangle()` is wrapped at the rectangle's right edge. Newlines are inserted at the wrap points as discussed in the API description of `getTextFromRectangle()`, further above.

Recording an HP3000 Session

The HP3000 Component differs from other components in that a major portion of the Action Model is built for you automatically. This happens as you interact with the host in the Native Environment pane as part of a live HP3000 terminal session. Composer records your interactions as a set of auto-generated actions in the Action Model. Typically, in other exteNd Composer components (such as a JDBC Component), you must manually create actions in the Action Model, which then perform the mapping, logging, transformation, communication, and other tasks required by the component or service. By contrast, when you create an HP3000 Component, you *record* requests and responses to and from the host, which end up as actions in the Action Model. In addition, you can add standard actions (Map, Log, Function, etc.) to the Action Model just the same as in other components.

NOTE: In order to successfully build an HP3000 Component, you should be familiar with HP 700/92 commands and the specifics of the application you intend to use in your XML integration project.

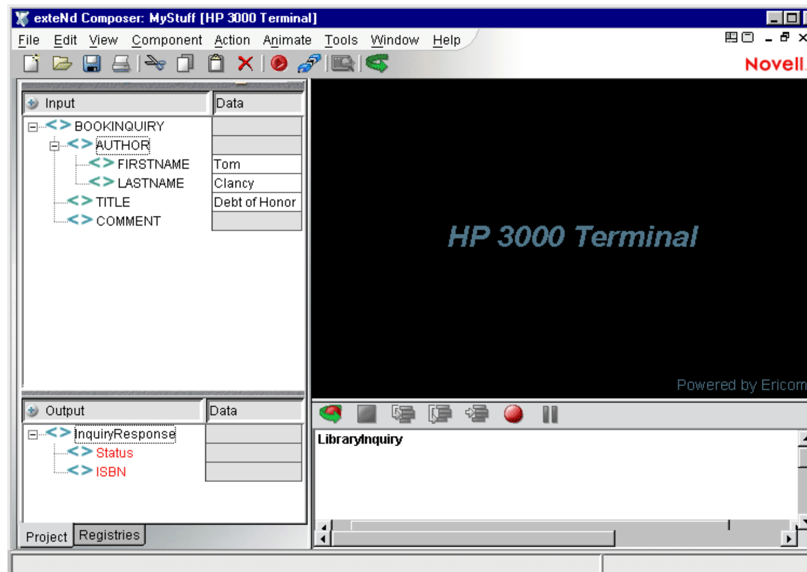
The following example demonstrates several common tasks that you will encounter in building HP3000 Components, such as:

- Creation of Check Screen actions
- Automatic creation of Send Buffer actions
- Drag-and-drop mapping of Input DOM elements to HP3000-screen prompts
- Drag-and-drop mapping from the Native Environment Screen to the Output DOM
- The use of ECMAScript expressions to manipulate Screen object elements

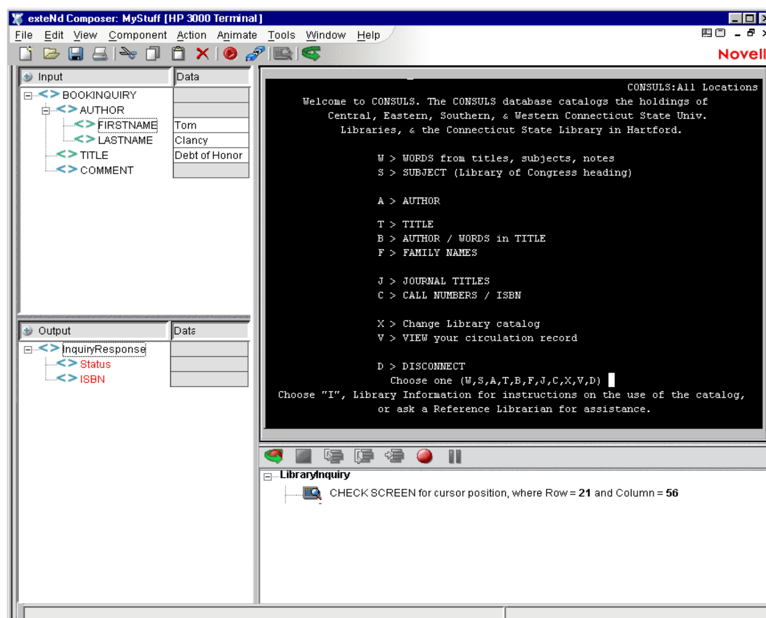
In the following example, we start with an input XML document that contains the title and author of a book. The goal of our Web Service is to do an author search online, using the terminal app, to see if a book by the given title exists in the library system. If so, we retrieve its ISBN (International Standard Book Number) code in an Output DOM. Whether we succeed or not, we insert an appropriate status message in the Output DOM.

➤ To record an HP3000 session:

1. Create an HP3000 Component per the procedure shown on page 19 of the previous chapter.
2. Once created, the HP3000 Component Editor window appears, with the words “HP3000 Terminal Emulation” in the center of the Native Environment Pane, indicating that no connection has yet been established with a host.



3. Click the **Record** button. You are automatically connected to the host that you selected in the Connection Resource for the component. An input screen appears in the Native Environment pane as shown below.

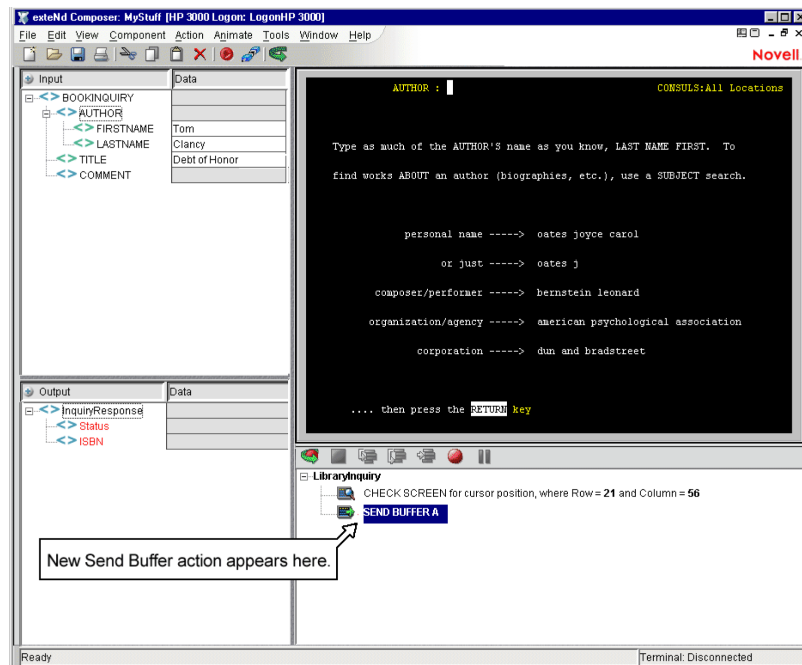


4 Click the **Create Check Screen Action** button in the toolbar. A new Check Screen action appears in the action list. It defaults to a go-ahead condition based on the *current cursor position* (which is assumed to always be 21,56 on this screen, with every future execution of this component—an assumption worth questioning). You may decide to accept the default Timeout of 1500 milliseconds for your Check Screen action if the program you are accessing has a relatively quick response time. (Even so, careful testing of the component should be done in order to verify that this timeout value is safe.)

5 Type the letter **A** (for Author) in the input screen of the HP3000 environment pane. A new Send Buffer action appears automatically in your component's action list. Notice that the 'A' you typed is already in the action.

NOTE: Terminal commands are often case-sensitive and should generally be entered in ALL CAPS.

In this part of this particular host application, merely typing a single character (without hitting Enter or Return) causes a new screen to appear. The host, in other words, processes the typed character immediately. This is a common terminal idiom. You will not always need to hit Return or Enter to get to a new screen.



In response to 'A', the host program sends the new screen shown above.

6 Because we wish to terminate the Send Buffer action and go on to interact with the new screen, you should click the Check Screen button in the toolbar, at this point, to allow the component to "sync" our next action with the current screen. Click the **Create Check Screen Action** button now. The new Check Screen action appears in the action list.

NOTE: Were you to simply start typing your next command at this point (without first creating a new Check Screen action), the command would be appended to the still-active Send Buffer. In essence, you would be creating a "type-ahead" buffer. At runtime, the buffer (containing two sets of screen commands concatenated together) would be sent all at once. While this would work okay in this particular program, the type-ahead technique could fail in other real-world terminal programs. Therefore, use caution when deliberately overloading a Send Buffer action. A "best practices" approach is to create a *new* Check Screen action for *every* new screen that appears during your session.

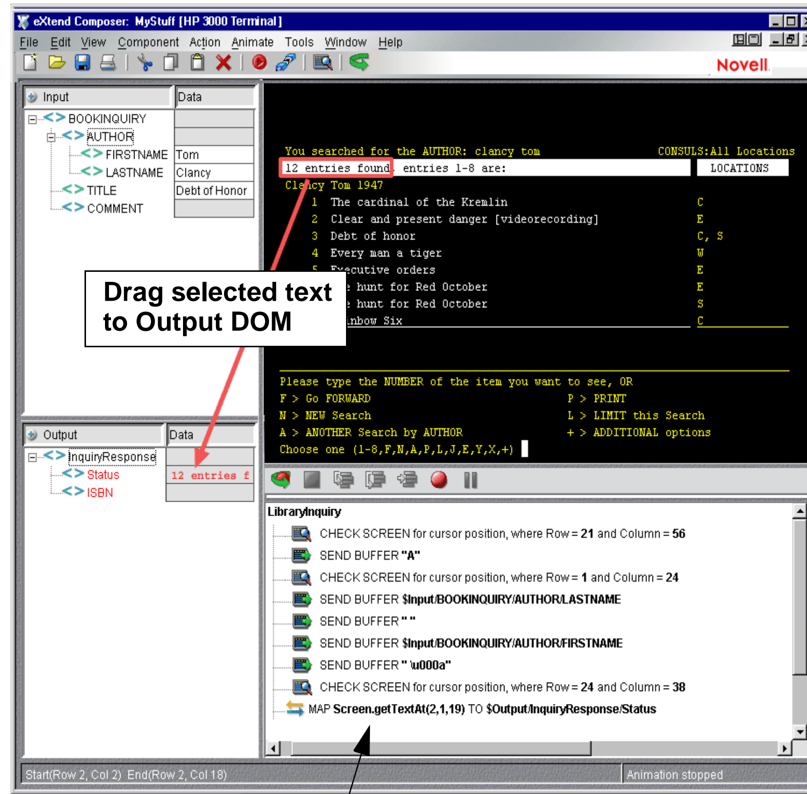
7 Drag the BOOKINQUIRY/AUTHOR/LASTNAME node from the Input DOM to the cursor position in the Native Environment Pane. "Clancy" (without quotation marks) appears in the prompt zone and a new Send Buffer action appears automatically in the Action Model.

NOTE: This terminal application is expecting the author's name to be provided as Last Name followed by First Name (with a space in between). Hence, we dragged the LASTNAME element first.

8 Hit the spacebar on your keyboard. Notice that a space character is added to "Clancy" in the Native Environment Pane. Also, a new Send Buffer action is created containing just the space character.

9 Drag the BOOKINQUIRY/AUTHOR/FIRSTNAME element from the Input DOM to the cursor position in the Native Environment Pane. "Tom" (without quotation marks) appears after "Clancy " in the prompt zone and a new Send Buffer action appears in the Action Model.

10 Note that the terminal screen has not changed (the host has not acted on our input), because it is waiting for Return or Enter. Press **Enter** to tell the host that our query string (the author's name) is complete. A new Send Buffer action appears, containing **<enter>**, and the Native Environment Pane updates to reflect the query results.



New action appears here

11 Click the **Create Check Screen** button in the toolbar. A new Check Screen action appears, with a default go-ahead condition based on the cursor location of row 24, column 38. (Row 24 is the bottom row and column 38 is about halfway across the 80-column screen; see screenshot above.) There is no need to change the Check Screen default in this case.

12 In the Native Environment Pane, select the terminal-screen text in row 2, from column 2 to column 18, by clicking and dragging the mouse.

NOTE: Notice that as you click and drag, the onscreen row/column coordinates of the selected area are displayed in the status line of the component editor window (lower left corner).

13 Lift your finger off the mouse button and place the mouse over the selected text. A finger cursor will appear. Click-drag the selection to the Output DOM **InquiryResponse/Status** node. The selected text is inserted into the DOM at the desired location, and a new Map Action is generated in the Action Model automatically.

14 Click the Record button to turn recording off.

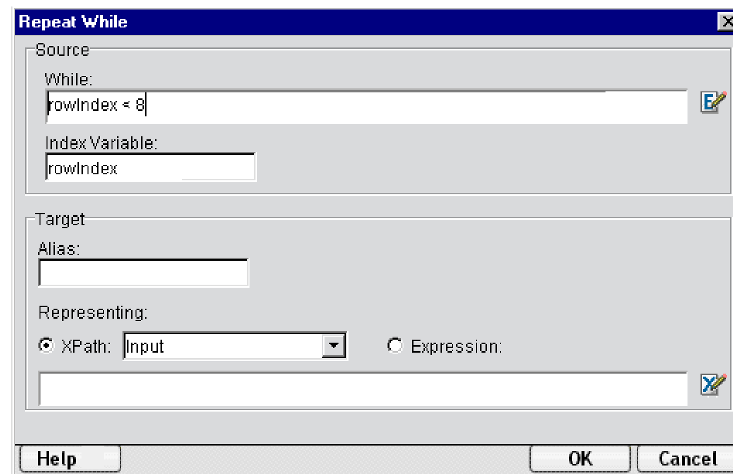
Looping Over Multiple Rows in Search of Data

In the example above, the goal is to find the ISBN (International Standard Book Number) information for the book we're interested in and map it into the Output DOM. Therefore, when the application shows the result of your author search, you need to scan that screen, looking for the book title in question. If the title exists, your next action should be to send the corresponding line number, which will cause the application to display a *new* screen showing detailed information (including ISBN) for the book.

By simple visual inspection of the terminal emulator screen (see previous illustration), it's easy to see that Tom Clancy's *Debt of Honor* is listed as line-item number 3 in the search-results screen. But this only holds true for this particular search. A search on a different author/title combination might yield a hit at a different line position. (Or if Tom Clancy writes more books, *Debt of Honor* could assume a different listing position.) To determine the line position of the book at runtime, we should iterate through lines 4 through 11 of the terminal screen, searching for the string stored in the BOOKINQUIRY/TITLE node of our Input DOM. The next example shows how to do this, building on the previous example.

➤ **To search for a data item one row at a time:**

1. At the bottom of the Action Model, add a new Repeat While action. (Perform a right-mouse-click, then select **New Action**, **Advanced**, and **Repeat While**.) The Repeat While dialog appears.



2. In the **While** text-entry box, type an expression representing the loop-termination condition you wish to apply to this loop. In this case, our condition involves a check of the index variable, `rowIndex`. We will be checking 8 rows of screen data in all.
3. In the **Index Variable** text-entry area, enter the name of your index variable (in this case, `rowIndex`).
4. Since we are only retreiving a single value (one book) from the screen, we do not need to fill in the optional Target portion of the dialog. Therefore, just click **OK**. A new Repeat While action is added to the component's Action Model.

5. In this example, we're looking for a specific string within a given row. If the string is found, we will take several actions, then break out of the loop. We will perform our row parsing and string search within a Decision Action. Create a new Decision Action by clicking the right mouse button and selecting **New Action** > **Decision** from the contextual menu. The Decision Action dialog appears.

6. Enter a **Decision Expression**. In this example, the three-line expression is:

```
var myRow = Screen.getTextAt(rowIndex+4, 1, 80).toLowerCase();
var bookTitle =        String(Input.XPath("BOOKINQUIRY/TITLE")).toLowerCase();
myRow.indexOf( bookTitle ) != -1
```

The first line uses the Screen object's `getTextAt()` method (see page 35) to retrieve the 80 characters of data (i.e., one full line, in a 24x80 terminal screen) at `rowIndex + 4`. We add an offset of 4 to the index variable because our search of screen data should *begin* at row 4 and continue through row 11. (The index variable itself will have values from 0 to 7. The loop terminates when `rowIndex` reaches 8.)

The second line of code above simply retrieves the book title as a lowercase string from the Input DOM. (Notice that because we don't want our search to be case-sensitive, we force both strings—the query string and the target-object string—to be lowercase.)

The final line of code is the actual "condition check." It relies on the core-ECMAScript String method `indexOf()`, which returns –1 when the argument string is *not* a substring of the string on which the method is being called.

7. In the TRUE branch of the Decision Action, create a new **Send Buffer** action. (Right-mouse-click, then choose **New Action** > **Send Buffer** from the contextual menu.) The Send Buffer dialog appears.

8. Click the **Expression** radio button and then enter an ECMAScript expression in the text-edit area. In this example, we've entered:

```
var item = Screen.getTextAt( rowIndex + 4, 1,10);
var regex = new RegExp("\\d+");
item.match(regex)[0];
```

The first line retrieves the first ten characters of data in the "hit" row using the `getTextAt()` method. Within this string, we want the first substring of numeric characters, representing the line number of the book (i.e., 3). One way to extract this substring is with the ECMAScript String method, `match()`, which takes a regular expression object as an argument. On success, this method returns an array, of which the zeroth item is the matched text. Our regular expression consists of backslash-d followed by a plus sign, which means "one or more digit characters in a row."

NOTE: The RegExp constructor takes a String argument, in which backslashes that are to appear as *literal* backslashes "must be escaped with a backslash."

The net result of these lines of ECMAScript is that the number preceding the book title in the target row (namely, '3') is supplied to the host application via a Send Buffer action. No newline need accompany the number '3'. Upon receiving this number, the host application will immediately send back a new screen giving detailed information about the indicated book, as shown below.

```
You searched for the AUTHOR: clancy tom              CONSULS:All Locations
                                                          Record 3 of 12
AUTHOR       Clancy, Tom, 1947-
TITLE        Debt of honor / Tom Clancy.
PUBLISHER    New York : G.P. Putnam's Sons, c1994.
DESCRIPT     766 p. ; 24 cm.
NOTE         11/95, c.1 $25.95 gift.
SUBJECT      Ryan, Jack (Fictitious character) -- Fiction.
             Intelligence service -- United States -- Fiction.
ISBN         0399139540 (alk. paper) :

    LOCATION            CALL NO.                      STATUS
1 > CCSU Stack Level 5  PS3553 L245 D43 1994          AVAILABLE
2 > SCSU Main           PS3553.L245 D43 1994          AVAILABLE

Key NUMBER to see more information, OR
R > RETURN to Browsing              N > NEW Search
F > FORWARD browse                  A > ANOTHER Search by AUTHOR
B > BACKWARD browse                 + > ADDITIONAL options
Choose one (1-2,R,F,B,N,A,Z,S,P,G,E,Y,+)
```

9 Create a new Check Screen action by performing a right-mouse-click and selecting **New Action > Check Screen** from the contextual menu. The Check Screen dialog appears.

10 Select the **Expression** radio button and enter “true” in the text-edit area. Set a **Min wait** value of 100, or any value you know from experience is generous.

NOTE: The combination of “true” and 100 means you will *automatically accept* any screen data that gets sent within 100 milliseconds.

11 Create a new Function Action. (Right-mouse-click: select **New Action > Function**.) In this action, we will retrieve the first ISBN number on the page, if one exists, and store it into an ECMAScript global.

```
The expression we will use is:
this.isbn = "Not found"; // set up global
var screen = Screen.getText( 1, 24 * 80 ); // fetch whole screen
if (screen.indexOf('ISBN') != -1) // if ‘ISBN’ occurs, get it
    this.isbn = lTrim( screen.split('ISBN')[1] ).split(' ')[0];
```

The first line above simply declares and initializes an ECMAScript global variable (which, on success, will be overwritten with a valid ISBN value).

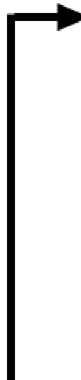
The second line of code retrieves the entire screen buffer as a string and places it in a local variable, `text`. (We assume here that we’re in 24x80 mode.)

The third line checks the screen buffer to see if “ISBN” occurs in it. If so, we split the buffer into an array of substrings using “ISBN” as the delimiter. The array member at index 1 will contain the ISBN number, trailed by a partial screen’s worth of information (and possibly containing one or more leading space characters). The custom ECMAScript function `lTrim()` is used to trimming leading spaces, while the split method is again employed to break our string into an array of substrings, assuming spaces to be the delimiters. The zeroth item of this final array is the ISBN string that we’re looking for. See the series of graphics below.

`screen.split('ISBN')[0]`

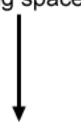
```
You searched for the AUTHOR: clancy tom                 CONSULS:All Locations
                                                                Record 3 of 12
AUTHOR       Clancy, Tom, 1947-
TITLE        Debt of honor / Tom Clancy.
PUBLISHER    New York : G.P. Putnam's Sons, c1994.
DESCRIPT     766 p. ; 24 cm.
NOTE         11/95, c.1 $25.95 gift.
SUBJECT      Ryan, Jack (Fictitious character) -- Fiction.
             Intelligence service -- United States -- Fiction.
```

```
ISBN         0399139540 (alk. paper) :

    LOCATION             CALL NO.                        STATUS
1 > CCSU Stack Level 5   PS3553 L245 D43 1994            AVAILABLE
2 > SCSU Main            PS3553.L245 D43 1994            AVAILABLE

Key NUMBER to see more information, OR
R > RETURN to Browsing              N > NEW Search
F > FORWARD browse                  A > ANOTHER Search by AUTHOR
B > BACKWARD browse                 + > ADDITIONAL options
Choose one (1-2,R,F,B,N,A,Z,S,P,G,E,Y,+)
```



`screen.split('ISBN')[1]`

leading spaces



`lTrim( screen.split('ISBN')[1] )`

```
0399139540 (alk. paper) :

    LOCATION             CALL NO.                        STATUS
1 > CCSU Stack Level 5   PS3553 L245 D43 1994            AVAILABLE
2 > SCSU Main            PS3553.L245 D43 1994            AVAILABLE

Key NUMBER to see more information, OR
R > RETURN to Browsing              N > NEW Search
F > FORWARD browse                  A > ANOTHER Search by AUTHOR
B > BACKWARD browse                 + > ADDITIONAL options
Choose one (1-2,R,F,B,N,A,Z,S,P,G,E,Y,+)
```

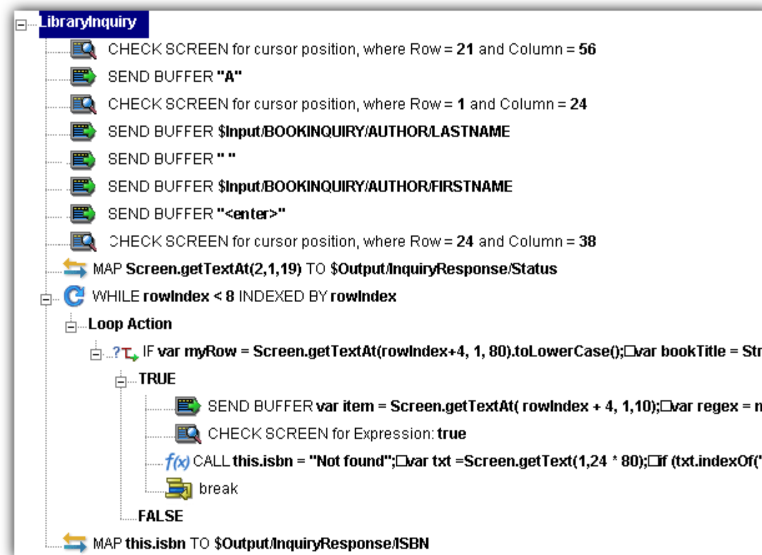
```
(screen.split('ISBN')[1]).split(' ')[0]
```

```
0399139540
                          (alk. paper) :

    LOCATION                CALL NO.                      STATUS
1 > CCSU Stack Level 5      PS3553 L245 D43 1994          AVAILABLE
2 > SCSU Main               PS3553.L245 D43 1994          AVAILABLE

Key NUMBER to see more information, OR
R > RETURN to Browsing                  N > NEW Search
F > FORWARD browse                      A > ANOTHER Search by AUTHOR
B > BACKWARD browse                     + > ADDITIONAL options
Choose one (1-2,R,F,B,N,A,Z,S,P,G,E,Y,+
```

12 On finding the information we're looking for, we no longer need to iterate through line items. Therefore, create a **Break Action** to break out of the loop. (Right-mouse-click; New Action; Break.)

13 Create a **Map action** that maps `this.isbn` to the **InquiryResponse/ISBN** node of the Output DOM.

The completed Action Model looks like this:



Editing a Previously Recorded Action Model

You will encounter times when you need to edit a previously recorded action model. Unlike the situation with other components, editing an HP3000 Component requires extra attention. When an HP3000 Component executes, it plays back a sequence of actions that expect certain screens and data to appear at certain times in order to work properly. So when editing a component you must be careful not to make the action model sequence inconsistent with the host program execution sequence you recorded earlier.

In general, to ensure successful edits, the following recommendations apply:

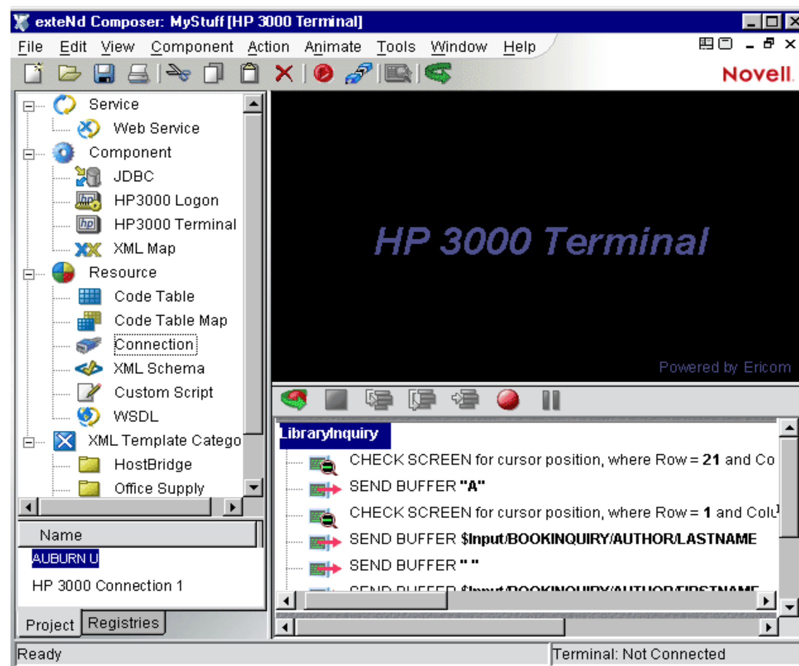
- Exercise extreme care when using Cut, Copy, and/or Paste to delete, move, or replicate actions in your Action Model. Actions that were created automatically during a “Record” session will often create data dependencies that are easily overlooked in the editing process.
- When you need to use drag-and-drop to add new Map actions to your Action Model, click the Start Animation button in the Action Pane toolbar and step to the line of interest in your Action Model; then Pause animation and turn on Record mode. At this point, you can safely drag to and from the screen. Following this procedure will prevent your Action Model from getting out of sync with the host or conflicting with previously mapped DOM data.

Changing an Existing Action

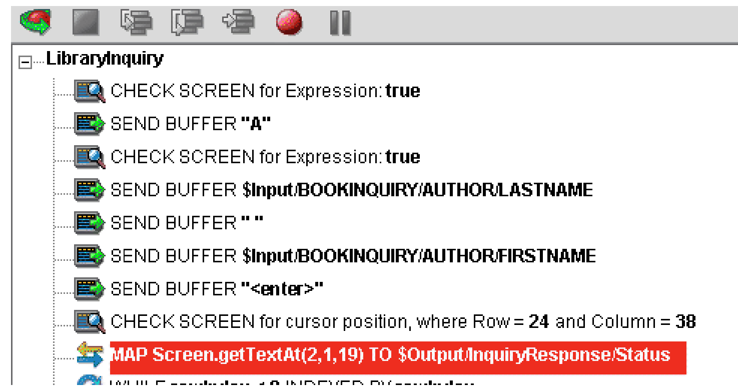
The following procedure will explain how to change an existing action in a previously recorded session.

➢ To Change an existing action in a previously recorded Action Model:

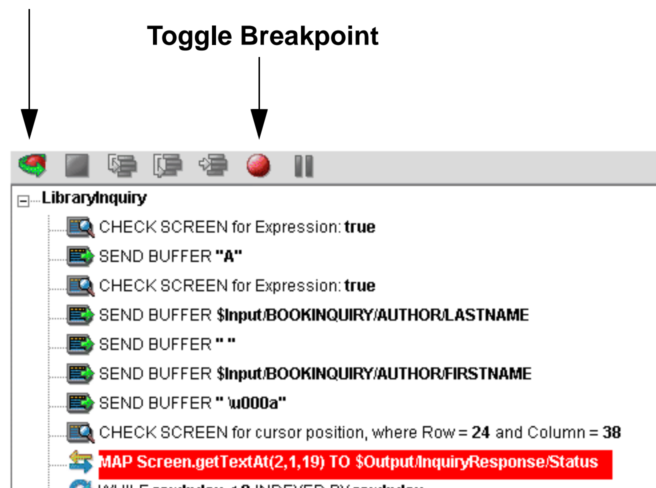
1 Open the component that includes the Action Model you'd like to edit. The component appears in the HP3000 Component Editor window.



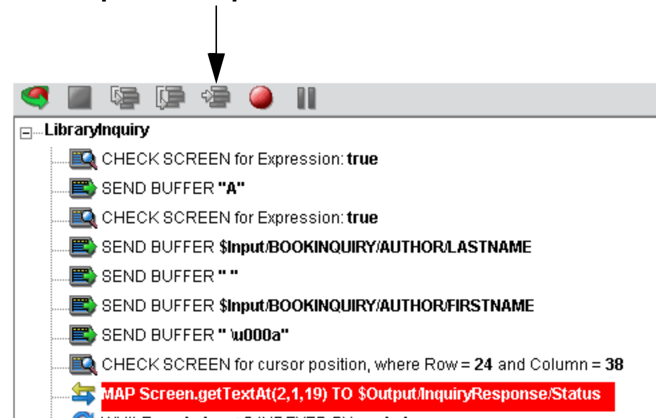
2 Navigate to the action in the Action Model where you'd like to make your edit and highlight the action.



3 Click the **Toggle Breakpoint** button (or press **F2**). The highlighted action becomes red.

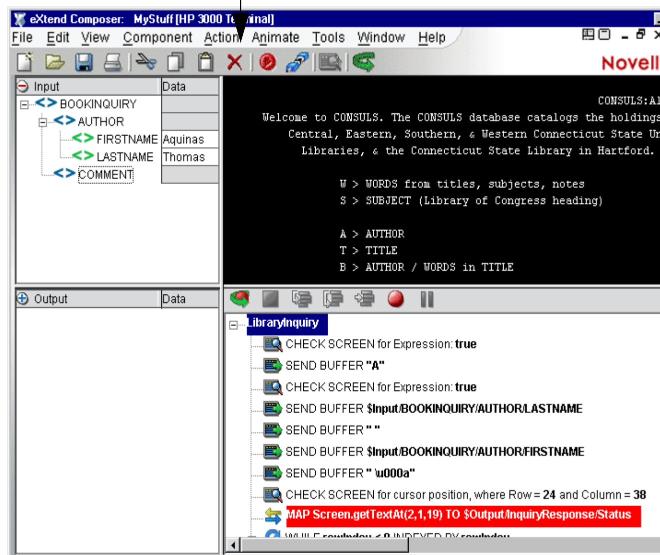


4 Click the **Start Animation** button. The animation tools (in the Actions pane's toolbar) become enabled.



5 Click the **Step to Breakpoint/End** button. The Action Model executes all of the actions from the beginning of the Action Model to the breakpoint you set in step 3 above.

6 In the Component Editor tool bar, click the **Record** button.



7 Perform any additional drag-and-drop (or other) actions that you'd like to make to the Action Model.

8 Turn off recording. (Toggle the **Record** button.)

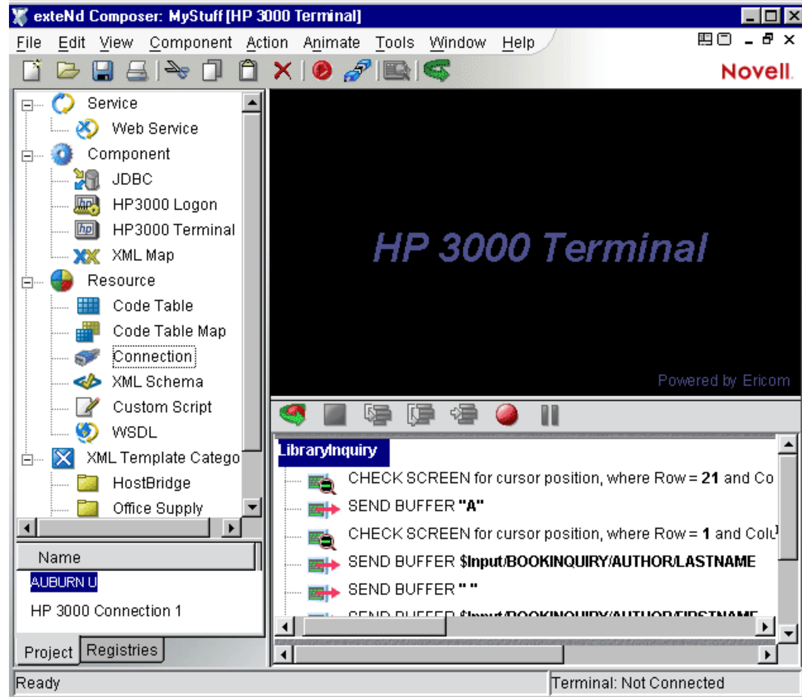
9 Test your component.

Adding A New Action

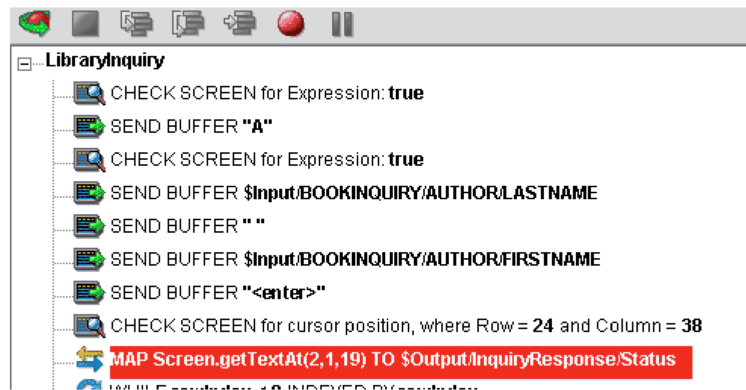
The following procedure explains how to add a new action in a previously recorded session.

➤ **To Add a Action to a previously recorded Action Model:**

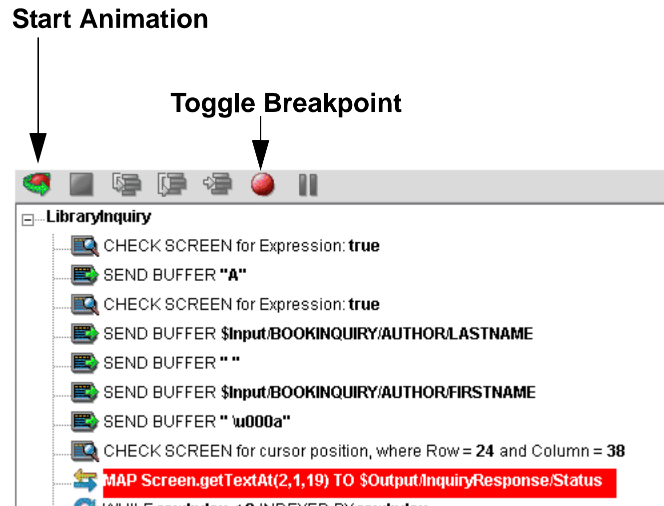
1 Open the component that includes the Action Model you'd like to add an action in. The component appears in the HP3000 Component Editor window.



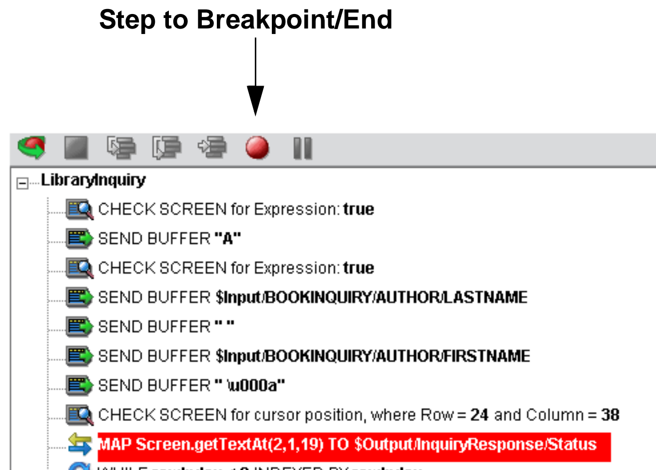
2 Navigate to the action in the Action Model where you'd like to make your addition and highlight the action.



3 Click the **Toggle Breakpoint** button (or press **F2**). The highlighted action becomes red.

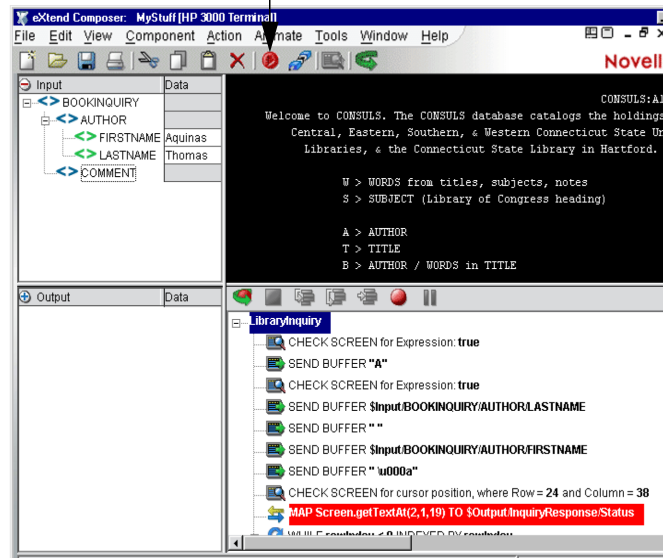


4 Click the **Start Animation** button. The animation tools (in the Actions pane’s toolbar) become enabled.



5 Click the **Step to Breakpoint/End** button. The Action Model executes all of the actions from the beginning of the Action Model to the breakpoint you set in step 3 above.

6 In the Component Editor tool bar, click the **Record** button.



7 Use Composer's drag and drop features to add new Map actions that interact with the screen. The new action will be added directly under the highlighted line.

8 Turn off recording. (Toggle the **Record** button.)

9 Test your component.

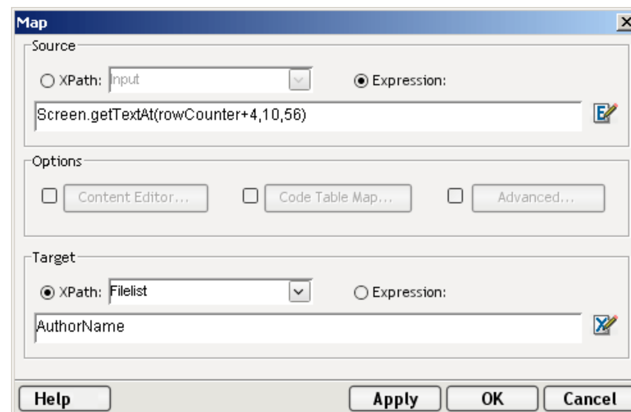
About Adding Alias Actions

If you are adding Map Actions in a loop that are alias perform the following steps:

➤ To Add an Alias Action to a previously recorded Action Model:

1 Open a component.

2 From the **Action** menu, select **New Action/Advanced,** then **Map**. The Map Action dialog box displays.



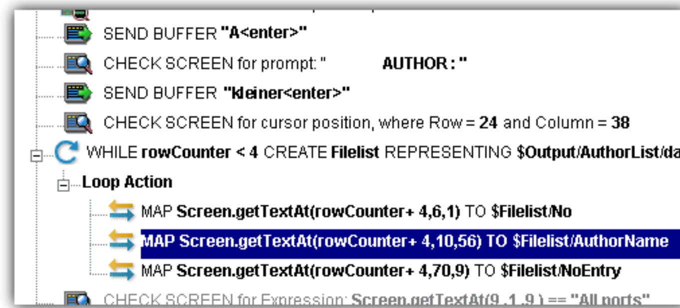
3 Select the Expression for Source, and the dropdown box is grayed out.

4 Either type in the information, or click the **Expression Builder** button and create a new expression.

5 Create an XPath to be represented by the alias. Click from the dropdown list for the alias.

6 Click **OK**.

7 The new action is inserted below the line you select. (New line is highlighted in the screen below to show it was inserted.

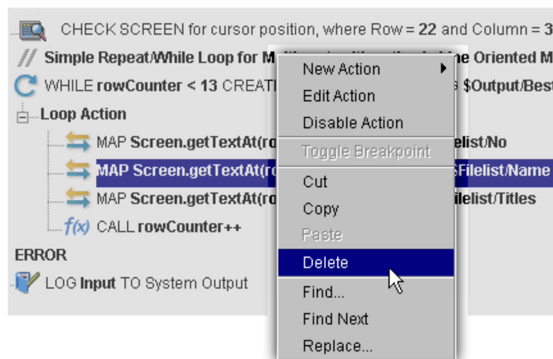


Deleting an Action

The following procedure explains how to delete an action in a previously recorded session

➤ **To Delete an Action to a previously recorded Action Model:**

Highlight the action line that you want to delete and click on the RMB and select Delete from the menu. You may also highlight the line and press the Delete button on your keyboard.



Testing your HP3000 Component

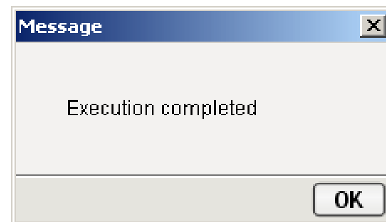
Composer includes animation tools that allow you to easily test your component. On the HP3000 Component Editor tool bar you'll find the Execute button, which allows you to execute the entire Action Model and verify that your component works as you intend. It is important to test a newly created HP3000 Component to be sure that Timeout values in all Check Screen actions are appropriate and that Send Buffer and other actions work as intended.

➤ To execute an HP3000 Component:

1 Open an HP3000 Component. The HP3000 Component Editor window appears.

Execute button

2 Select the **Execute** button. The actions in the Action Model execute. If the component executes successfully, a message appears as follows.



3 Click **OK**.

After executing the component, you may want to doublecheck the contents of your DOMs to be sure all of the appropriate data mappings occurred as expected. To make all data elements visible, select **Expand XML Documents** from the **View** menu. This expands all of the parents, children, data elements, etc. of the DOM trees, so that you can easily see the results of execution of the component.

Using the Animation Tools

In the Action Model, you'll find animation tools that allow you to test a particular section of the Action Model by setting one or more breakpoints. Using these tools, you can run through the actions that work properly, stop at the actions that are giving you trouble, and then troubleshoot the problem actions one at a time.

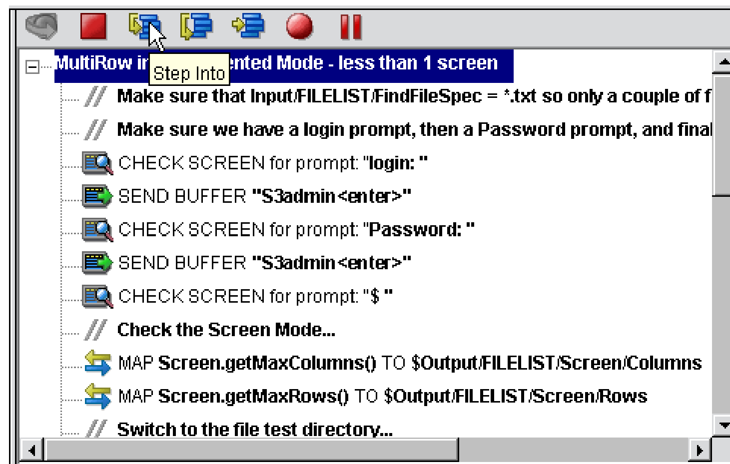
•The following procedure is a brief example of the functionality of the animation tools. For a complete description of all the animation tools and their functionality, please refer to the *exteNd Composer User's Guide*.

➢ **To run an HP3000 Component using Animation Tools:**

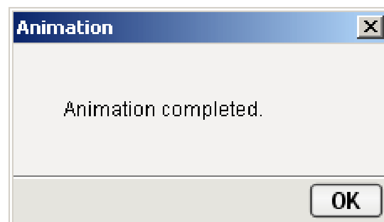
1. 1.Open an HP3000 Component. The component appears in the HP3000 Component Editor window.

   NOTE: Animation and Recording are *mutually exclusive modes* in the component. In order to record during animation, you must either pause, or stop animation and then turn on record mode.

2. Click the **Start Animation** button in the Action Model tool bar, or press **F5** on the keyboard. All of the tools on the tool bar become active, and a connection is established with the host. The Native Environment Pane becomes active.
3. Click the **Step Into** button. The first Check Screen action becomes highlighted.



4. Click the **Step Into** button again. The Check Screen action (above) executes and the next action becomes highlighted.
5. Click the **Step Into** button repeatedly to execute actions one-by-one.
6. Click other buttons (Step Over, Run To Breakpoint, Pause, etc.) as desired to control the execution of the component. Note that you can set a breakpoint at any time during execution by clicking the mouse on an action line and hitting F2 or using the Set Breakpoint button.
7. Once execution is complete, the following message appears.



Tips for Building Reliable HP3000 Components

The following tips may be helpful to you in building reliable HP3000 Components.

- Always precede a Send Buffer action with a Check Screen action.
- Always follow a Send Buffer action with a Check Screen action.
- In Check Screen actions, accept the default go-ahead condition (based on cursor position) only when you are certain that the absolute cursor position will always be constant for the given screen. Many times, it is safer to write a custom expression.

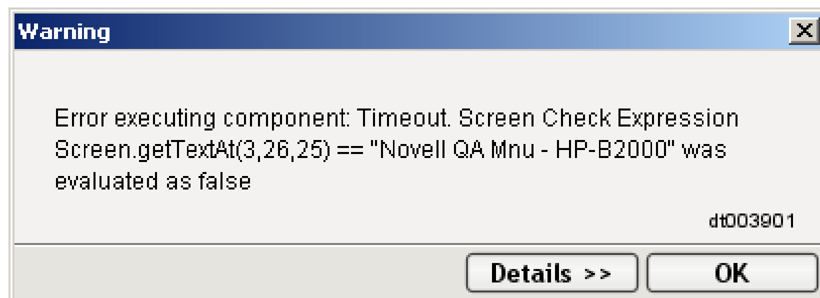
- A fast, accurate way to create a prompt-based Check Screen action during recording is to highlight (select) the characters of interest immediately preceding the cursor (up to but not including the cursor position), then click the right mouse button and select Check Screen. This automatically creates a Check Screen action based on the prompt you highlighted.
- When typing a custom prompt string under Prompt (in the Check Screen dialog), remember to escape any quotation marks that might appear within the prompt string.
- Avoid using Check Screen go-ahead criteria based on variable information, such as dates, times, etc.
- Avoid Check Screens that do nothing but wait a specified period of time using the Min Wait setting. While this technique may work, it can create significant performance bottlenecks.
- Remember that the default Timeout values used in Check Screen actions are calculated from actual response times during the design session. This has a couple of implications. First, the default Timeout value may need to be increased, for load-sensitive applications. Secondly, deleting a Check Screen action may cause synchronization timeouts on subsequent executions. Careful testing will reveal these sorts of problems.
- When disjointed go-ahead criteria come into play, such as when the middle of a screen remains constant during a repaint but the first and last lines change, you may want to create two Check Screen actions then combine them into one action that's based on an expression.

Using Other Actions in the HP3000 Component Editor

In addition to the Check Screen and Send Buffer actions, you have all the standard Basic and Advanced Composer actions at your disposal as well. The complete listing of Basic Composer Actions can be found in Chapter 7 of the *Composer User's Guide*. Chapter 8 contains a listing of the more Advanced Actions available to you.

Handling Errors and Messages

In testing an HP3000 Component, you may encounter errors relating to Check Screen and/or Send Buffer actions. The result is a dialog similar to the following:



This section discusses possible error conditions and how to deal with them.

Check Screen Errors

Most of the errors you are likely to encounter at execution time will be related to Check Screen actions. It is important to realize that *every* one of the Check Screen errors discussed below is a timeout error. If one of the errors described below occurs, it means that the go-ahead criteria you specified in the Check Screen setup dialog *were not met within the Timeout period*. Therefore, you should first try to determine whether slow host response might be the real problem (in which case, the solution is to increase the Timeout value for the Check Screen action in question). If the error still occurs after the Timeout value has been increased, then you can be sure the error is due to an incorrect or inappropriate go-ahead condition in your Check Screen action.

The following paragraphs describe typical error messages and their meanings.

"Expected cursor position (Row = {0}, Column = {1}) was not established"

This error means that the Check Screen failed because the cursor was not at the expected location at the expiration of the Timeout period. Perhaps the host application changed, or the prompt line may be varying dynamically in some way that you weren't anticipating, etc. It's also possible, as explained above, that the Check Screen simply "timed out" for reasons having to do with heavy host load or a bad connection. Try increasing the Timeout value for the given Check Screen action. If that doesn't help (or if you suspect that the problem involves an inappropriate choice of go-ahead criteria), try rewriting the Check Screen go-ahead condition based on something other than fixed cursor coordinates. For example, specify a prompt string, or use an Expression to validate the screen contents in some way.

"Expected prompt text {0} was not established"

This error means that the Check Screen failed because the prompt was not identical to the specified (expected) prompt string prior to the expiration of the Timeout period. The prompt line may be varying dynamically in some way that you weren't anticipating. Or (as explained above) the host response time may simply have increased unexpectedly due to heavy load or other factors. If you suspect that host latency is a problem, try increasing the Timeout value for the Check Screen action. Otherwise, rewrite your Check Screen go-ahead criteria to be based on something other than a hard-coded prompt value. For example, specify an Expression that validates the prompt in some way.

"Screen Check Expression {0} was evaluated as false"

This error happens when the Check Screen go-ahead is based on an ECMAScript expression and the expression happens to evaluate as *false* at execution time. Once again, it's important to realize that this sort of error can be triggered simply on the basis of slow host response (timeout). When the host is slow to respond, it means that your ECMAScript expression will be evaluated on the basis of *whatever is in the screen buffer as of the moment of timeout*. If no data (or insufficient data) have arrived, the expression is bound to evaluate as false.

To fix this sort of problem, either increase the Timeout value for this Check Screen action (if you suspect that the problem is host latency) or try modifying the logic in your ECMAScript expression.

Send Buffer Errors

Send Buffer errors will, in general, be rare. Be on guard, however, for Send Buffers that contain more than one screen's worth of commands (so-called "type-ahead" buffering). Such actions are easy to create accidentally. An Action Model with overloaded Send Buffers may work correctly as you step through actions at animation time, but can fail when the component-as-a-whole is executed, due to screen synchronization problems. The way to avoid problems here is to make sure that for every Send Buffer action, there is always be a corresponding Check Screen action.

Errors Involving Connections

If connection pooling, which is discussed in detail in Chapter 6, is used, and there has been an attempt to log on with a bad UserID or Password, that connection instance will not be usable and that member of the pool will be skipped over in subsequent connection requests. An error message will be sent to the server log saying “Logon connection in pool <Pool name> was discarded for User ID <User ID>.” You should check for messages of this sort during preproduction testing and/or any time performance issues arise.

Finding a “Bad” Action

When you have a large Action Model (containing dozens or hundreds of Check Screen and Send Buffer actions), simply locating the action that’s responsible for an error can be a challenge. One way to find the problematic action is to:

1. Select and Copy the text after “Expected” in the error dialog. (Click the Details button if need be, to expose the full error description. Highlight the relevant text, such as cursor coordinates. Then use Control-C to Copy.)
2. Click inside the Action Model.
3. Use Control-F to initiate a search.
4. Paste the error text into the search dialog.
5. Execute the search.

Of course, if you have multiple Check Screen actions that are based on identical go-ahead criteria, the foregoing technique won’t necessarily be helpful. If that’s the case, set a breakpoint at the midpoint of your Action Model, and run the component. If the error doesn’t occur, move the breakpoint to a spot halfway between the original breakpoint and the end of the action list. (Otherwise, if the error *does* happen, set the breakpoint at a spot one quarter of the way down from the top of the action list.) Run the component again. Keep relocating the breakpoint, each time *halving* the distance between the last breakpoint or the top or bottom of the action list, as appropriate. In this way, you can quickly narrow down the location of the problematic action. (Using this “binary search” strategy, you should be able to debug an Action Model containing 128 actions in just 7 tries.)

5 Advanced HP3000 Actions

Terminal-based computing differs from other types of computing (including other IBM terminal-based interactions) in a number of important ways:

- Data arrive a character at a time, rather than in chunks.
- There is no obvious structure to arriving data; and the data may arrive in an arbitrary order.
- Screen updates may involve just a portion of the screen (perhaps a single character) or the whole screen.
- Retrieval of data sets may require repeated roundtrip communications with the host. (One query may bring many screens' worth of data, which must be captured through multiple "page forward" commands, etc.)
- Information that spans screens may be (and often is) partially duplicated on the final screen.

These factors can make automating a terminal interaction (via an Action Model) challenging. The goal of this chapter is to suggest some strategies for dealing with common (yet potentially problematic) terminal-computing situations in the context of an eXtend Action Model.

To get the most out of this chapter, you should already have read Chapter 4, "Performing HP3000 Actions" and you should be familiar with Action Model programming constructs (such as looping via the Repeat While action). In addition, you should have some experience using ECMAScript.

Data Sets that Span Screens

A common requirement in terminal computing is to capture a data set that spans multiple screens. In cases where the screen contains a line that says something like "Page 1 of 4," it's a straightforward matter to inspect the screen at the point where this line occurs (using one of the ECMAScript Screen-object methods described earlier, in the section titled "HP3000-Specific Expression Builder Extensions") and construct a loop that iterates through all available screens. But sometimes it's not obvious how many screens' worth of data there may be. In some cases, the only clue that you have may be the presence of a "More" command (for example) at the top or bottom of the screen, which changes to "Back" (or "End," or some other message) when you reach the final screen. In other cases, you may be told how many total *records* exist, and you may be able to determine (by visual inspection) how many records are displayed *per screen*; hence, you can calculate the total number of screens of information awaiting you.

The point is that if your query results in (potentially) more than one screen's worth of information, you must be prepared to iterate through all available screens using a Repeat/While action, and stop when no additional screens are available. You will have to supply your own custom logic for deciding when to stop iterating. Your logic might depend on one or more of the following strategies:

- Determine the total number of screens to visit by "scraping" that information, if available, off the first screen.
- Divide "total records" (if this information is available) by the number of records per screen (if this is known in advance), and add one.
- Visit screens one-by-one and break when a blank record is detected.
- Visit screens one-by-one until a special string (such as "End" or "Go Back") is detected.
- Visit screens one-by-one until two consecutive identical screens have been encountered.

Obviously, the strategy or strategies you should use will depend on the implementation specifics of the terminal application in question. For some applications, iterating through screens until a blank record is encountered would be appropriate, whereas for others, it wouldn't be.

An example of an Action Model that combines two of these strategies will be discussed in detail further below.

Dealing with Redundant Data

In terminal applications, it's common for the final screen of a multiscreen result set to be "padded" with data from the previous screen. In this way, the appearance of a full screen is maintained.

Consider the following two screen shots. The top one shows the next-to-last screen's worth of information in a query that returned six screens of information. Notice that the reversed-out status line (row 2 from the top) says "43 entries found, entries 33–40 are:", followed by line entries. Since there are 43 records in the overall data set, and the next-to-last screen ends with record number 40, you'd expect the next (and final) screen to show records 41 through 43. Instead, the final screen looks like the one at the bottom of the next page. Notice that it shows records 36 through 43—that is, it contains five records (36 through 40) from the *previous* screen. In most cases, you will not want to capture this redundant data. The question is: How can you detect and reject redundant records of this sort?

ECMAScript offers an easy and convenient way of maintaining unduplicated lists. The trick is to create a bare (uninitialized) Object, then attach record names as properties. Since no object can ever have two properties with identical names, assigning record names as property names means the object's property list is an unduplicated list of record names.

```
 You searched for the AUTHOR: thomas aquinas          CONSULS:All Locations
 43 entries found, entries 33-40 are:                     LOCATIONS
 Thomas Aquinas Saint 1225 1274
    33  Summa contra gentiles.                             W
    34  Summa contra gentiles.                             E
    35  The Summa contra gentiles of Saint Thomas Aquinas,  S
    36  Summa theologiae : a concise translation           E
    37  Summa theologica                                   C
    38  Summa theologica.                                  C, S, W
    39  Summa theologica.                                  C
    40  Summa theologica.                                  C

Please type the NUMBER of the item you want to see, OR
F > Go FORWARD       A > ANOTHER Search by AUTHOR     + > ADDITIONAL options
B > Go BACKWARD      P > PRINT
N > NEW Search       L > LIMIT this Search
Choose one (33-40,F,B,N,A,P,L,J,E,Y,X,+)
```

```
You searched for the AUTHOR: thomas aquinas               CONSULS:All Locations
43 entries found, entries 36-43 are:                          LOCATIONS
Thomas Aquinas Saint 1225 1274
   36  Summa theologiae : a concise translation             E
   37  Summa theologica                                     C
   38  Summa theologica.                                    C, S, W
   39  Summa theologica.                                    C
   40  Summa theologica.                                    C
   41  Summa theologica. Prima secundae. Quaestio 90-97.    S
   42  The teacher : The mind : Truth, questions X, XI      C
   43  The teacher, The mind (Truth, questions X, XI)       S

Please type the NUMBER of the item you want to see, OR
B > Go BACKWARD                             P > PRINT
N > NEW Search                              L > LIMIT this Search
A > ANOTHER Search by AUTHOR                + > ADDITIONAL options
Choose one (36-43,B,N,A,P,L,J,E,Y,X,+)
```

A short example will make this clearer. Suppose you have an array of items in which some items are listed more than once:

```
var myArray = new Array( "Tom","Amy","Greg","Tom","Amy");
```

To unduplicate this array, you could assign properties to a bare object, where the property names equal the array values:

```
var myObject = new Object(); // create a bare object
for (var i = 0; i < myArray.length; i++) // loop over array
{
        var arrayMember = myArray[ i ]; // fetch array member
        myObject[ arrayMember ] = true; // create the property
}

 // Now obtain all property names
 // in a new, unduplicated array:
var uniqueValues = new Array();
var n = 0; // counter
for (var propertyName in myObject) // enumerate property names
       uniqueValues[ n++ ] = propertyName;

// Now 'uniqueValues' contains just "Tom","Amy","Greg"
```

We will use this trick to our advantage in the terminal application example discussed below.

An Example of Looping over Multiple Screens

Let's look at a sample HP3000 component that combines several of the strategies we've been talking about. The host application is a university library system's book locator service. In this example, we have an input document that specifies an author's name. Based on that name, we want to query the library for all available book titles by that author and capture the results to an output DOM. We want the output document to contain an unduplicated list of titles.

This example will demonstrate:

- How to "scrape" data from multiple screens, without knowing in advance how many screens there are.
- How to reject duplicate records as they are encountered.
- How to create Output DOM nodes programmatically.
- Breaking out of the main loop if a blank record is encountered *or* the final screen has been reached.

The logic for our Action Model's main loop can be summarized (in pseudocode) as follows:

```
Determine the number of records-per-screen
While (true) // enter a "forever" loop
    Fetch a record
    IF Record is Valid   // i.e., not blank
         Write data to Output DOM
         IF Screen has been completely processed
                IF this is not the final screen
                         Fetch next screen
                ELSE BREAK   // final screen processed
    ELSE BREAK     // blank record reached
```

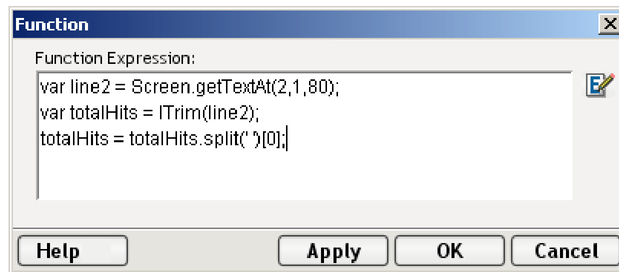
Initial Actions

The initial portion of the Action Model for this example looks exactly like the actions created in the earlier example (in the "HP3000 Actions" chapter) under "Recording an HP3000 Session", except that in this case our author is Thomas Aquinas. The initial actions are simply the Check Screen and Send Buffer actions necessary to conduct an Author search on "Thomas Aquinas."

The initial screen of our result set looks like:

```
 You searched for the AUTHOR: thomas aquinas           CONSULS:All Locations
 43 entries found, entries 1-8 are:                       LOCATIONS
 Thomas Aquinas Saint 1225 1274
     1  An Aquinas reader                                 S
     2  Aquinas Scripture series. 1966 --> See THOMAS, AQUINAS, SAINT, 1225?-1
     3  Aquinas: selected political writings.             S
     4  Commentary on Aristotle's Physics.                C, S
     5  Commentary on the De anima of Aristotle           C
     6  Concerning being and essence (De ente et essentia) E
     7  De regno, ad regem Cypri.                         S
     8  An introduction to the metaphysics of St. Thomas Aquina E

Please type the NUMBER of the item you want to see, OR
F > Go FORWARD                              P > PRINT
N > NEW Search                              L > LIMIT this Search
A > ANOTHER Search by AUTHOR                + > ADDITIONAL options
Choose one (1-8,F,N,A,P,L,J,E,Y,X,+)
```

At the very beginning of the second row, we're told how many records ("entries") were found. We can capture this information by using a Function Action:



This three-line script obtains all of Row 2 in a local variable called `line2`, trims leading spaces off the line, and splits the line on space characters (capturing the zeroth member of the resulting array into a variable, `totalHits`). After this, it's a simple matter to write the "total hits" number into the Output DOM using a Map Action.

At this point, we could use the "total hits" number as the basis for our main loop. But for illustration purposes, we're going to bypass that tactic, because not every HP3000 host reports "total hits" information on the first response screen. We will, however, take advantage of the fact that this particular application reports the number of records per screen (in row two). Here again, though, it's possible—with clever ECMAScript programming—to determine "records-per-screen" information dynamically, at runtime. Alternatively, you can just hard-code this value after visually inspecting the screen.

NOTE: At some point, you will have to decide whether (and under what circumstances) it makes sense to hard-code something like the number of records per screen, as opposed to applying runtime logic. With terminal applications, it's rare that you can count on being able to determine *every* important screen characteristic dynamically. Some fore-knowledge of the host application's behavior will almost always be implicit in the final Action Model.

We will store the records-per-screen number in an ECMAScript variable, `booksPerScreen`. In this example, there are eight records per screen.

Setting Up the Main Loop

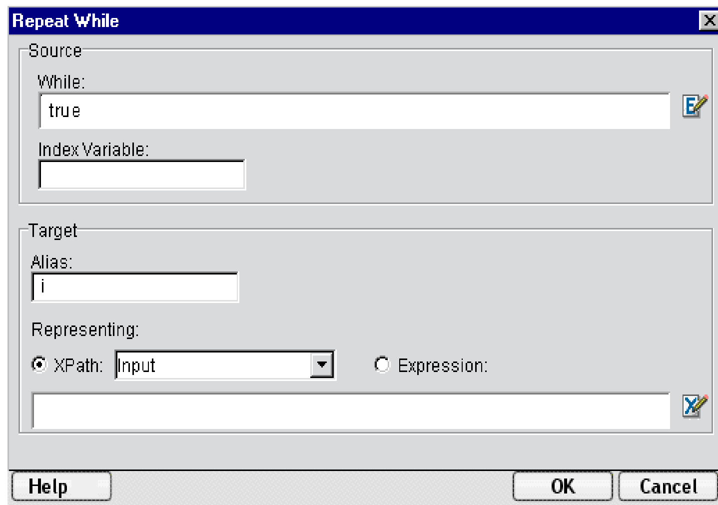
Before creating our main loop, we need to set up an index variable that will be used when creating nodes in our Output DOM. This index (called `bookNumber`) will start at *one* and will be incremented once for every book title we capture to Output. The reason this index starts at *one* instead of *zero* is that DOM nodes use one-based indexing. We will be using `bookNumber` to index our nodes.

We also will use an ECMAScript expression (in a Function Action) to create a blank ECMAScript object:

```
var bookTable = new Object();
```

By storing book titles as property names on this object, we can keep an *unduplicated* list of records, as explained further above (see "Dealing with Redundant Data").

To create the loop, we place a Repeat While action in the Action Model. (Right-mouse-click, then select **New Action > Repeat > Repeat While**.) The dialog settings for this look like:



By setting the While condition to *true*, we are—in effect—creating an infinite loop. The exit conditions for this loop are twofold:

- If a blank record (all space characters) is encountered, the loop is terminated.
- If the current screen is identical to the previous one, the loop is terminated.

The latter condition provides a suitably robust way to break out of our infinite loop.What's more, it's generally applicable to a wide range of terminal applications—not just the library-query application.

The index variable `i`, which cycles from zero to `booksPerScreen - 1`, serves two roles:

- It lets us know when it's time to fetch a new screen (namely, when the value reaches `booksPerScreen - 1`), and
- It serves as the basis for our *row offset* when fetching records.

Screen Caching

One additional bit of pre-loop setup code involves caching the current screen. We include the following Function Action statement immediately before beginning the loop:

```
previousScreen = Screen.getTextAt(1,1,Screen.getColumnCount() *
Screen.getRowCount());
```

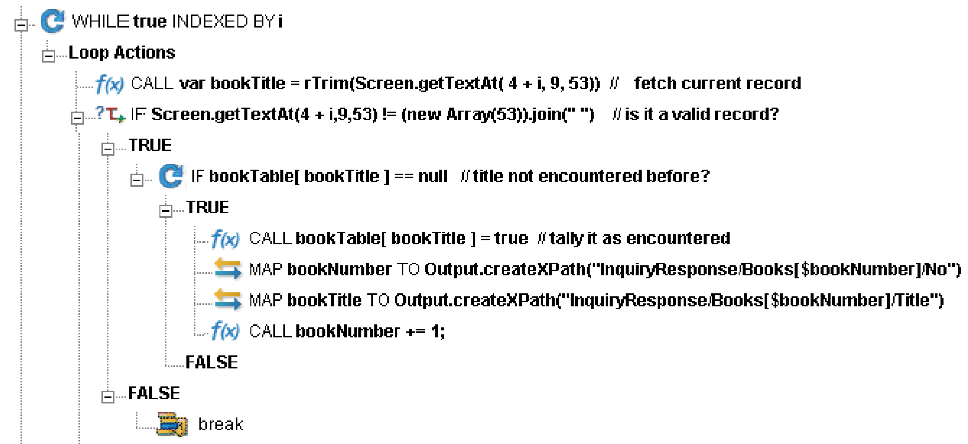
The variable `previousScreen` caches the contents of the last-looked-at screen so that we can check newly obtained screens against it. If a newly obtained screen has exactly the same content as the screen we just processed, this is a hint that we have reached the final screen (and we should therefore terminate the loop).

The Main Loop

We're now in a position to look at what our Action Model's main loop actually does.

First Half

Consider the first portion of the loop as shown below. This is where most of the real work takes place.



The first action inside the loop is a Function Action, which fetches the 53 characters beginning at column 9 of row `4 + i`. The rows we're interested in include rows 4 through 11, inclusive; this is the zone in which the host reports our line items. Since `i` cycles from zero to 7, we can use "4 + i" as a row offset in our code.

Once we've obtained a record, we do a validation check before proceeding. Only if the zone that the record came from is non-empty will we continue with the loop. We use a Decision Action with a decision expression of:

```
Screen.getTextAt( 4 + i, 9, 53) != (new Array(53) ).join(" ")
```

The statement on the right side of the expression means "create a new, empty array of length 53, and convert it to a String by joining the array members together, using a single space character as the delimiter." Since each array member is null, this essentially forms a String consisting of 53 space characters in a row. We can compare this String with the onscreen string to determine if a blank record was encountered.

In the TRUE branch of our Decision Action, we immediately check to see if the book title we just fetched has already been encountered. (We don't want duplicates.) Since we've been using the tactic of keeping book titles as property names on the `bookTable` object (see discussion further above), all we have to do to check for prior existence of the book is execute a Decision Action against the expression:

```
bookTable[ bookTitle ] == null
```

If this statement is true, it means the `bookTable` object has no property whose name matches the String in `bookTitle`. When this is the case, it means we can go ahead and do our mapping operations. (Otherwise, we fall through and keep iterating.)

In the TRUE branch of this decision, we mark `bookTable[ bookTitle ]` as *true*; this assigns a new, non-null property to `bookTable`. We then map an index number as well as the book title to new nodes in our Output DOM. By applying a target expression of

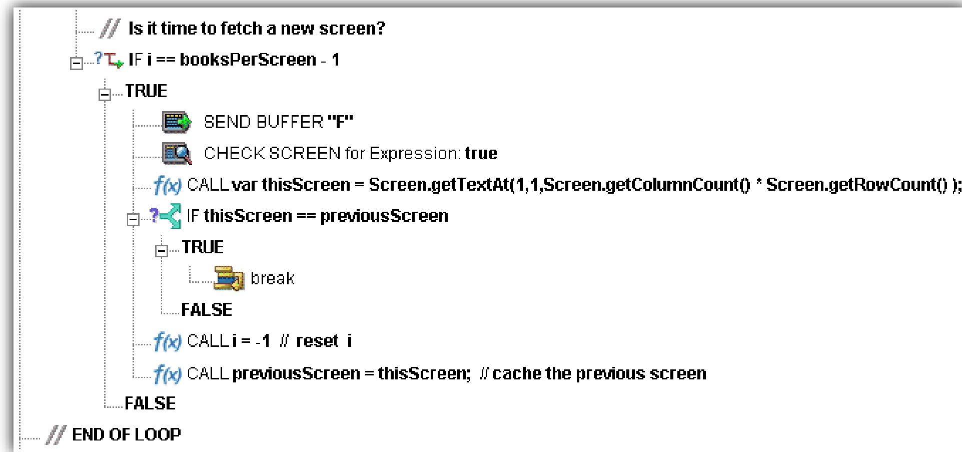
```
Output.createXPath("InquiryResponse/Books[$bookNumber]/Title")
```

for mapping, we are able to use the running index in `bookNumber` to create a new node instance under **InquiryResponse/Books** with element name **Title**.

Finally, we increment `bookNumber`.

Second Half

In the final portion of our loop, we check to see if it's time to fetch a new screen. If so, we execute the necessary Send Buffer command to tell the host we want to page forward to the next screen.



Notice that as soon as we've fetched the new screen, we capture its contents into a String variable, `thisScreen`. Then we execute a Decision Action in which we simply compare `thisScreen` to `previousScreen`. If the two are equal, we use a Break Action to break out of the loop. Otherwise we fall through and continue executing.

NOTE: Use care when deciding a Min Wait time for the Check Screen action shown above. If the Min Wait is short and the go-ahead condition is true, it's possible you could unintentionally skip a screen and break out of the loop prematurely.

If we're still executing, we reset `i` (the row index variable) and stuff `thisScreen` into `previousScreen` in preparation for the next round.

The Output DOM resulting from our loop ends up looking something like this:

Output	Data
InquiryResponse	
TotalTitles	43
Books	
No	1
Title	An Aquinas reader
Books	
No	2
Title	Aquinas Scripture series
Books	
No	3
Title	Aquinas: selected politic
Books	
No	4
Title	Commentary on Aristotle
Books	
No	5
Title	Commentary on the De a
Books	
No	6
Title	Concerning being and e

The DOM lists all the titles found for this author, numbered sequentially. And even though the final screen's worth of data contains a significant amount of information duplicated from the preceding screen, our DOM contains no duplicate titles.

Performance Considerations

You can perform millisecond-based timing of your Action Model's actions by wrapping individual actions (or block of actions) in timing calls.

➢ **To time an Action:**

1. Click into the Action Model and place a new **Function Action** immediately before the action you wish to time. (Right-mouse-click, then **New Action** > **Function**.)
2. In the Function Action, enter an ECMAScript expression of the form:
   ```
   startTime = Number(new Date)
   ```
3. Insert a new **Function Action** immediately *after* the action you wish to time.
4. In the Function Action, enter an ECMAScript expression of the form:
   ```
   endTime = Number(new Date)
   ```
5. Create a **Map Action** that maps `endTime - startTime` to a temporary DOM element. (Right-mouse-click, **New Action** > **Map**.)
6. Run the Component. (Click the **Execute** button in the main toolbar.)

If you do extensive profiling of your Action Model, you will probably find that the overwhelming majority of execution time is spent in Check Screen actions. (You will seldom, if ever, encounter a Check Screen that executes in less than 150 milliseconds.) Two implications of this worth considering are:

- ECMAScript expressions (in Map and/or Function actions) will seldom, if ever, be a performance consideration for the component as a whole.
- Overall component performance rests on careful tuning of Min Wait and Timeout values in Check Screen actions.

Finally, remember that testing is not truly complete until the deployed service has been tested (and proven reliable) on the app server.

For additional performance optimization through the use of shared connections, be sure to read the next chapter, on Logon Components.

6 Logon Components, Connections, and Connection Pools

This section discusses certain features available in the HP3000 Connect designed to maximize performance of deployed services.

HP3000 Session Performance

The overall performance of any service that uses back-end connectivity is usually dependent on the time it takes to establish a connection and begin interacting with the host. Obtaining the connection is "expensive" in terms of wait time. One strategy for dealing with this is *connection pooling,* a scheme whereby an intermediary process (whether the app server itself, or some memory-resident background process not associated with the server) maintains a set number of preestablished, pre-authenticated connections, and oversees the "sharing out" of these connections among client apps or end users.

Connection pooling overcomes the latency involved in opening a connection and authenticating to a host. But in terminal-based applications, a considerable amount of time can be spent "drilling down" through menu selections and navigating setup screens in order to get to the first bonafide application screen of the session. So even when connections are reused through pooling, session-prolog overhead can be a serious obstacle to performance.

Composer addresses these issues by providing connection pooling, managed by a special kind of component (called a *logon component*) that can maintain an open connection at a particular "drill-down" point in a terminal session, so that clients can begin transactions *immediately* at the proper point in the session.

When Will I Need Logon Components?

Logon Components are useful in several types of situations:

- When you have a need for multiple tiers of pooling based on multiple security challenges within your system. (For example, users may need one set of logon credentials to get into the network, another to get into the mainframe, and another to get into database.) Serial log-in requirements may dictate the use of multiple logon components.
- When your service needs stateful "session-based" connections.
- When you need the performance advantages available through connection pooling.

If performance under load is not a high-priority issue and your connectivity needs are relatively uncomplicated, you may not need to use Logon Components at all. But there is no way to know if performance is adequate merely by testing services at design time, on a desktop machine.

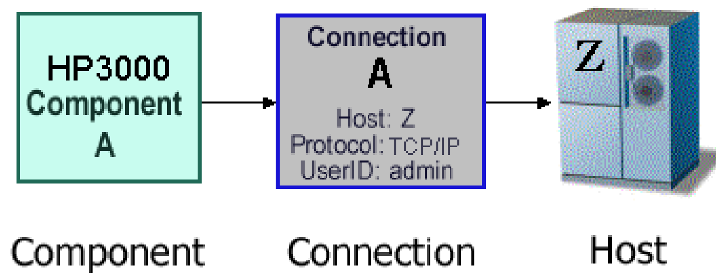
Components and services built with the HP3000 Component Editor may appear to execute quickly at design time (in Animation Mode, for example). But in real-world conditions—which is to say under load, with dozens or even hundreds of requests per second arriving at the server—session overhead can be a significant factor in overall transaction time. *The only way to know whether you need to use the special performance enhancement features described in this chapter is to do load testing on a server, under test conditions that mimic real-world "worst case" conditions.*

Connection Pool Architecture

When you install the Connect for HP3000, two types of Connection Resources are added to the Connection creation wizard:

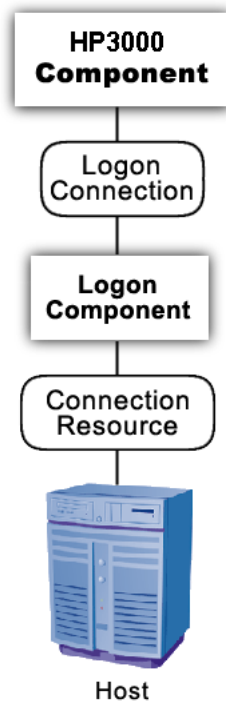
- HP3000 Connection
- HP3000 Logon Connection (henceforth referred to as a Logon Connection)

The HP3000 Connection is a true terminal *connection* and (when used by an HP3000 component) can establish a session with a host system. This is the connection-type we have been using throughout this Guide.



The HP3000 connection resource is designed to make an individual connection to the host on an as-needed basis. The connection is made just-in-time and discarded as soon as the client is done. It is not reused in any way.

The Logon Connection, on the other hand, is different. It defines a pool of User IDs and passwords, each of which can make its own connection. The Logon Connection also serves as an indirection layer to allow clients to connect to the host at exactly the point in the host program (exactly the screen) where the client needs to start. This entry-point-location behavior is made possible by the Logon *Component*. (A Logon Connection always uses a Logon Component to get to the actual connection.) The architecture is shown in the graphic below.



A *Connection Resource* is always required in order to get to the host. (This is true for any Composer service that uses HP3000 components.) For simplicity, this diagram shows the Connection Resource going directly to the host; in the real world, there may be intervening delegation layers for security purposes.

The *Logon Component* contains Actions (an action model) designed to find a particular screen of interest in the host program. This drill-down location is the effective entry point of the transaction for any upstream process that uses this Logon Component. You can think of the Logon Component as a go-between between the *physical* connection (represented by the Connection Resource) and the logic layer (represented by the HP3000 Component itself.

In order for an HP3000 Component (at the top of the diagram) to use a Logon Component, it needs to enlist the aid of a *Logon Connection* resource. The Logon Connection is the bridge between the HP3000 Component and the Logon Component.

The kinds and responsibilities of the various objects discussed above are summarized in the following table.

Object	Role
HP3000 Connection Resource	Allows a connection to be established with a host system.
Logon Component	Specialized type of component in which the action model contains Logon, Keep Alive, and Logoff action blocks. This component can maintain a connection at a particular launch screen in a host program.
Logon Connection	Specialized type of Connection Resource that associates a pool of UserIDs and passwords with a given Logon Component type. At runtime, connections are established for client processes on demand (and reused), with one Logon Component instance per connection. Every connection in the pool provides ready access to a given point (a particular launch screen) in the host program, thanks to the associated Logon Component (see above).
HP3000 Terminal Component	Contains the action model that comprises the business logic for a particular HP3000 interaction (or transaction).

The Logon Connection's Role in Pooling

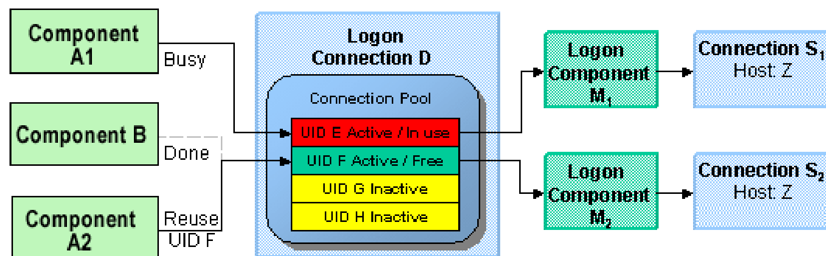
The Logon Connection differs from the ordinary "host-direct" connection resource in that it manages *pooling* (the sharing of connection instances and Logon Component instances at runtime).

In the context of a Composer service, pooling not only allows reuse of (open) connections at runtime, it also increases the effective bandwidth of a deployed service. Consider the simple case where you've designed an HP3000 component that uses a regular connection resource. In creating the connection resource, you will have specified a UserID and password for the resource to use so that at runtime, the component can log in to the host. When an actual running instance of your component is using that connection, no other instance of the component can log in to the host using that same set of credentials. The bandwidth of your service is limited to one connected instance at a time.

With a Logon Connection, on the other hand, numerous host connections can be maintained in a "live" state so that multiple instances of your component can access the host (each on its own connection) without waiting. Throughput is dramatically increased.

The diagram below shows one possible runtime case where three component instances (two instance of HP3000 Terminal Component A and one instance of HP3000 Terminal Component B) are executing on the server. Instance 1 of Component A is using UserID 'E' to obtain a connection. This component has its own dedicated instances of Logon Component M and Connection S.

Terminal Component B has just finished executing and is relinquishing its connection (established through credentials defined by UID 'F'). Note that because connection pooling is in effect, Component B's downstream resources (its Logon Component instance, M2, and its Connection instance, S2) are not simply discarded; they remain live. As a result, Terminal Component A2 is able to obtain (reuse) the M2/S2 resource instances that were previously held by Terminal Component B.



In this diagram, Logon Connection D is associated with four connections based on four UIDs (user IDs or credentials: A-thru-F). One is in use; another (UID 'F') is alive but not being used; and two are inactive but available (i.e., valid UIDs have been assigned, so these two connections can be made live at any time).

How Many Pools Do I Need?

It's possible for several different HP3000 components to draw from the same connection pool. It's also possible for different components to draw from different pools. This means different Logon Connections.

An important factor in deciding how many Logon Connection resources (in effect, how many pools) your service needs is the number of different start screens (or entry point screens) needed by the various components in your project. Suppose Terminal Component A needs to begin its work at a particular starting screen in a host application, but you've also designed another component—Terminal Component B—that needs to start on a different screen. Components A and B will need separate Logon Connections, and the separate Logon Connections will point to separate Logon Components. (In any given connection pool, Composer objects are shared in such a way that every user of the pool *must* start at the same screen.)

Pieces Required for Pooling

The combination of a Logon Connection, a Logon Component, and its Connection Resource form the basis of a connection pool. Starting from the host layer and working up the chain:

- The *Connection Resource* defines the most basic parameters necessary for establishing a connection with the host. When connection pooling is in effect, runtime instances of this object are kept alive and reused.
- The *Logon Component* defines the set of steps (actions) necessary to get to a particular entry point in the host program. (At runtime, an instance of this component will actually carry out those steps in order to arrive at, and maintain ready-to-use, a particular screen location in the host program.) When connection pooling is in effect, instances of this object are kept alive and reused.
- The *Logon Connection* is a special type of resource that contains all the information needed to define a connection *pool*. This resource is designed to encapsulate pool-management info and does not establish host connections directly; instead, it delegates those responsibilities to the Logon Connection (which delegates them, in turn, to the appropriate Connection Resource).

How Do I Implement Pooling?

To create the various pieces required for pooling, you'll go through the following basic steps (each of which will be discussed in greater detail in the sections to follow):

1. First, you'll create a basic HP3000 connection resource, as demonstrated in "To create an HP3000 Connection Resource:" on page -14 of this Guide.
2. Next, you'll create a Logon Component that uses the connection resource defined in Step 1. As part of this process, you'll create an action model designed to navigate to a certain point in the host program.
3. You will create a Logon Connection resource, which is a specialized type of connection resource that relies on a Logon Component (from Step 2) to make the basic connection (through the resource defined in Step 1).
4. Finally, you'll create an HP3000 Terminal Component and associate it with the Logon Connection resource of Step 3.

These steps are described in detail starting with the discussion in "Creating a Connection Pool" further below. Before going to that section, however, you should become familiar with the design principles behind the Logon Component (and also the Logon Connection). We'll start with the Logon Component, since it's impossible to create a Logon Connection without using a Logon Component.

The HP3000 Logon Component

The Logon Component is a special type of component: it has an Action Model, yet can be used as a connection resource as well. The Action Model of this type of component is designed to manage a connection that will be used by multiple HP3000 terminal components. In most respects, the Logon Component is the same as an HP3000 Terminal component. The differences are:

1. In a Logon Component, the Action Model is organized around connection-management tasks. Those tasks are implemented via special actions: the Logon Action, Keep Alive Action, and Logoff Action.
2. A Logon Component is not invoked directly by another component or service. Its invocation is under the control of a Logon Connection.

   NOTE: A Logon Component must and can only be used in conjunction with a Logon Connection.

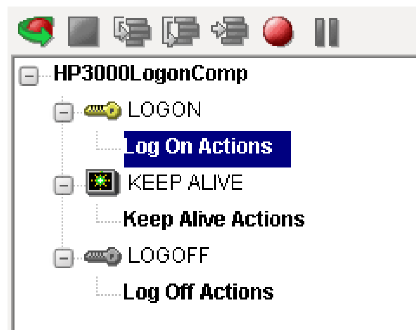
Instead of calling the Logon Component directly, using (for example) a Component Action, you will associate the Logon Component with a special connection resource called a Logon Connection. When your HP3000 Terminal Component executes, it executes via the Logon Connection, which in turn executes the Logon Component.

Logon, Keep Alive, and Logoff Actions

The Logon Component provides several screen-management capabilities that are important factors in overall performance. These capabilities are implemented in terms of Logon, Keep Alive, and Logoff actions:

- **Logon Actions** - These actions navigate through the host environment and park at a desired launch screen in the host system. The connection is activated using User IDs from the pool. The HP3000 components that subsequently reuse the connection have the performance benefit of already being at the launch screen and won't incur the overhead of navigating to the launch screen as if they had come in under their own new session.
- **Keep Alive Actions** - These actions do two important tasks. First, they prevent the host from dropping a connection if it is not used within a standard timeout period defined by the host. Second, these actions must insure that the connection is always positioned at the "launch screen in the host, even after performing the Keep Alive actions needed to prevent the connection from dropping (the first important task).
- **Logoff Actions** - These actions exit the host environment in a manner you prescribe for all the connections made by User IDs from the pool, when a connection is being terminated.

These actions and their meanings will be discussed in greater detail below. For now, it's enough to know that these three action groupings are created for you automatically when you first create a Logon Component. Note the (empty) Logon, Keep Alive, and Logoff action blocks in the action model shown below:



Logon Actions

Actions you place in the Logon group are primarily concerned with signing into the host security screen and then navigating through the host menu system to a launch screen where each HP3000 component's Action Model will start. It is important that any HP3000 component using a Logon component be able to start execution at the same common screen. Otherwise, the performance gains of avoiding navigation overhead won't be realized and more importantly, the odd HP3000 component won't work.

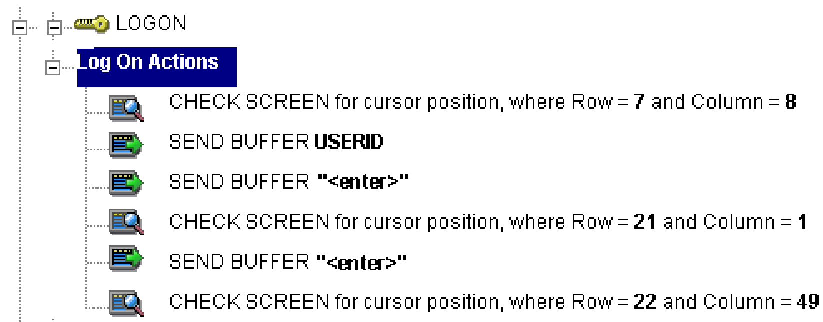
You can create actions under the Logon Actions block the same way as you would in an ordinary HP3000 Terminal Component—namely by using the Record feature to create (in real time) whatever actions are necessary in order to enter sign-on info such as User ID and Password (as well as your initial menu choices to arrive at the launch screen).

NOTE: Remember to use the User IDs and Passwords from the Logon Connection Pool. (See the discussion in “Creating a Logon Connection using a Pool Connection” on page -84.) To do this, you need to map the two special system variables called USERID and PASSWORD to the appropriate fields on the screen. By specifying these two variables, you make it possible for exteNd Composer to automatically locate and use values from the next active and free Pool slot.

The launch screen is a common point of execution for all the HP3000 Terminal Components that use the User ID pool provided by a Logon Connection. The Logon actions in a Logon Component (which are executed only once when a new connection is established) let the calling component—your HP3000 Terminal Component—begin execution at a given screen in the host program.

Maximizing Performance with the Logon Component

The Logon Actions must be structured properly and therefore always begin and end with a Check Screen Action as shown in the screen below.



The final Check Screen action in the Logon block guarantees that control is not turned over to the HP3000 Component before the screen of interest has arrived in the connection. Without this, the HP3000 Component could start at an invalid screen, throw an exception, and possibly corrupt a transaction.

NOTE: You may notice when animating a Logon Component that the ending Check Screen is skipped. This is normal design-time behavior. In a production environment , the actions in a Logon Component always execute in an interleaved manner with an HP3000 Terminal Component. Animating a Logon Component from start to finish actually creates an abnormal sequence of events that would result in two Check Screens being processed in succession, which is not allowed.

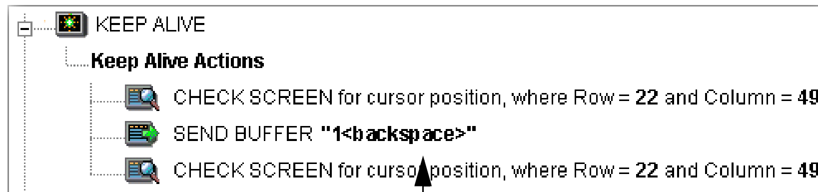
The performance benefit comes into play as a result not only of connection reuse but launch-screen reuse. For example, if a User ID pool of three entries is fully used and (ultimately) reused by the execution of a component fifteen times, the overhead of navigating to a menu item that executes the transaction of interest will occur only three times. Likewise, there will only be three logons to the host because the Logon actions at the top of a Logon Component are executed only once—when a new connection is activated (not when it is reused). This is key to obtaining maximum performance in a high-transaction-volume production settings.

NOTE: When possible, use the Try/On Error action to trap potential logon errors that may be recoverable. Otherwise, the UserID trying to establish the failed logon will be discarded from the pool, decreasing the potential pool size. The pool size will remain smaller until you manually reset the discarded connections using the exteNd Composer Enterprise Server Console for HP3000. *See the Managing Pools Sections in this Chapter for more details.*

Keep Alive Actions

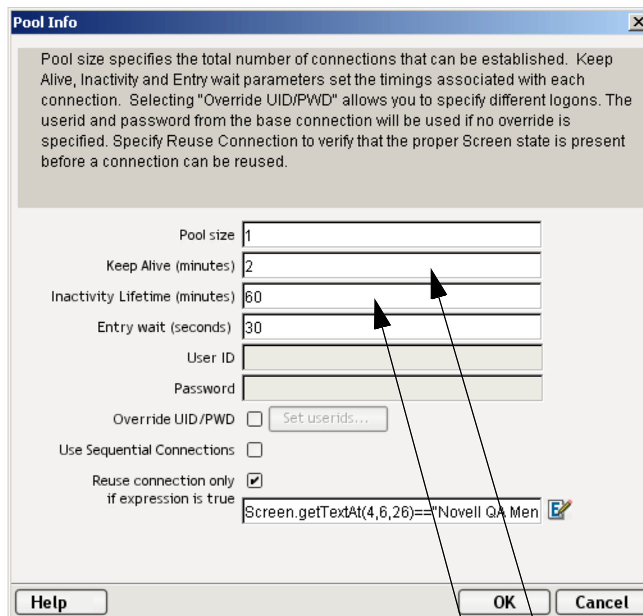
The Keep Alive block is where you will place actions that “ping the host” in whatever way necessary to keep the connection alive so that it can be reused.

Keep Alive actions usually involve sending a key like <ENTER>, to the host at some specified interval. However, if after sending the key the screen changes to some screen that is different than the launch screen, you must be sure to return the Logon Component to the launch screen in the Keep Alive section. Failure to do so will leave the next component at an incorrect screen, causing it to fail.



This escape sequence corresponds to “1” and “Backspace.”

The Pool Info dialog of a Logon Connection (see discussion in “Creating a Logon Connection using a Pool Connection” on page -84) is where you control how often the Keep Alive actions will execute. If you specify in your Logon Connection pool that you would like to keep a free connection active for 5 minutes, but the host will normally drop a connection after two minutes of activity, you can specify keyboard actions to let the host know the connection is still active such as sending an <ENTER>key.



interaction every 2 minutes

active connection for 5 minutes

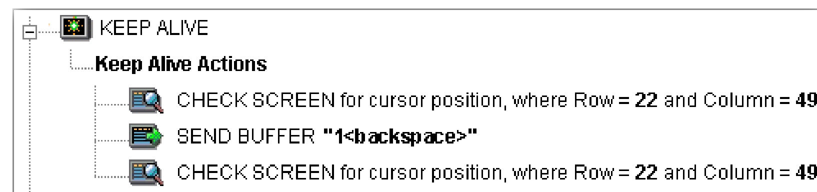
Keep Alive actions may be executed multiple times, but after the Keep Alive Time Period defined on the Pool Info dialog of the Logon Connection.

NOTE: The execution of the Keep Alive actions does not cause the Inactivity Lifetime clock to reset in the Logon Connection. Only an HP3000 Component's execution will reset the Inactivity Lifetime.

The last action inside a Keep Alive block should be an empty but "enabled" navigation action. If a user disables this last action, animation will not work properly due to two consecutive empty navigation actions occurring. For example, if an action in Logon and the first action in Keep Alive are disabled, an error occurs.

Maximizing Performance with Keep AliveKeep Alive Actions

Check Screens must occur at the beginning and end of the Keep Alive section. Not only does the Keep Alive section prevent the connection from closing, but it must make sure that the launch screen is present when the execution is completed. The beginning Check Screen checks to make sure that during the time the connection was available but not in use, that an unexpected screen didn't arrive from the host. And again, the ending Check Screen prevents releasing the connection to the next HP3000 Component prematurely after executing the Keep Alive actions. See the following screen.



Logoff Actions

Logoff actions essentially navigate the User ID properly out of the host system after a timeout.

Logoff actions execute only once for a connection and *only* when a connection times out (i.e. the Inactivity Lifetime expires) or screen expression criteria is not met, or the connection is closed via the HP3000 Server Console.

In a "best practices" sense, it's vitally important to make Logoff Actions bulletproof. If an exception occurs during execution of the Logoff actions, exteNd Composer will break its connection with the host, freeing the UserID in the pool. *But the UserID may still be active on the host*. Until the host kills the UserID (from inactivity), a subsequent attempt by the pool to log on with that UserID may fail, unless you've coded your logon to handle the situation. Logon failures cause the UserID to be discarded from the pool, reducing the potential pool size and performance overall. As with Logon and Keep Alive actions, the way to guarantee you are on the proper screen at the end of the logoff is to end with a Check Screen.

Logon Component Life Cycle

Each time a User ID is activated from the Logon Connection Pool, an instance of the corresponding Logon Component is created and associated with that User ID. Then the Logon actions are executed until the desired launch screen is reached. At this point the HP3000 Terminal component execution begins. When it is finished another HP3000 Terminal component using the same Logon Connection may begin executing, starting at the same launch screen.

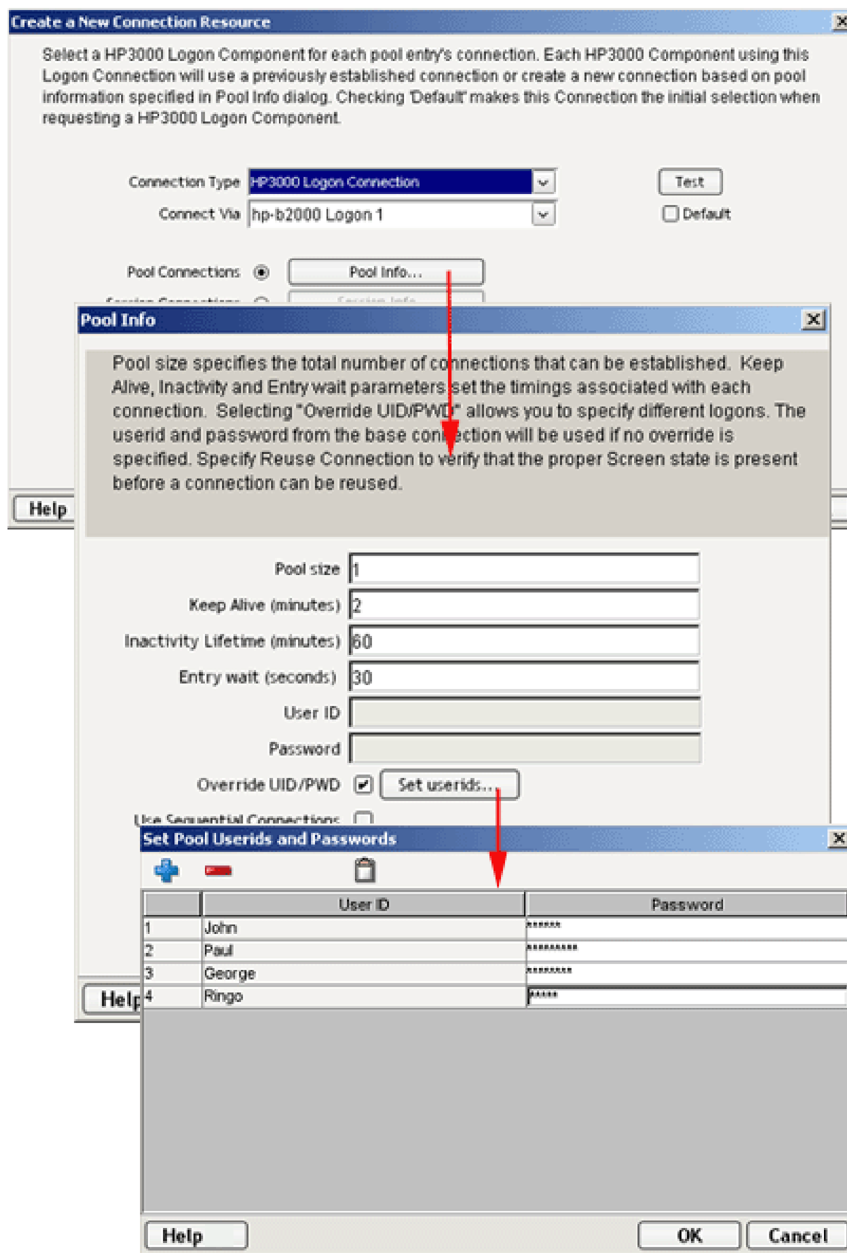
If no other component requests the connection, then the connection-instance in question enters an active but free state (an “idle state”) defined by the Inactivity Lifetime and Keep Alive settings on the Pool Info dialog of the Logon Connection. If the Keep Alive period (e.g., 2 minutes) is shorter than the Inactivity Lifetime (e.g., 60 minutes), then at appropriate (2-minute) intervals, the Keep Alive actions will be executed, preventing a host timeout and dropped connection; and the Keep Alive Period begins anew.

A Logon Component’s execution lifetime is dependent on the activity of the Logon Connection that uses it. As long as one entry in the Logon Connection pool is active, then one instance of the Logon Component will be in memory in a live state. A Logon Component instance will go out of scope (cease executing) when the last remaining pool entry expires due to inactivity. The only other way to stop execution of a Logon Component is through the HP3000 Console on the Server.

The HP3000 Connection

The Logon Connection is not a true connection object like an HP3000 Connection Resource, but a pointer to a Logon Component (which in turn connects to a host either through a conventional Connection Resource or yet more intervening Logon Connection/Logon Component pairs). The Logon Connection encapsulates information needed to describe a *pool of connections.* That includes User IDs and passwords, plus pool settings involving the time interval between retries on discarded connections, etc. Another function of the Logon Connection is that it ensures the use of different instances of the same Logon Component for all the User IDs for which connections are made.

The dialogs you’ll use in setting up a pool of User IDs for a Logon Connection are shown in the following set of illustrations. Arrows denote the buttons that lead to continuation dialogs.



Every Logon Connection is associated with a given Logon Component. In addition, the Logon Connection provides the following User ID pool functionality:

1. It allows the specification of multiple User IDs in advance ensuring that clients are able to secure a connection when one is needed
2. It allows the reuse of a User ID/connection once it is established to eliminate repeated user authentications and disconnects
3. It allows a single User ID to use multiple connections if this is supported by the host system
4. It keeps a connection active to prevent host timeouts during inactive periods
5. It lets you specify when to remove a connection from the active pool
6. It sets a timeout period to use for a fully active pool to provide a free connection
7. It lets you specify error handling dependent on the state of the Logon Component used by the Logon Connection

Many-to-One Mapping of Components to Logons

In order for multiple instances of an HP3000 component or different HP3000 components to use the same Logon Connection, the following conditions must be met:

1. All the HP3000 components must use the same Connection Resource (thereby sharing the HP3000 Host, Port and Terminal type).
2. All the HP3000 components must have a common launch screen in the host system from which they can begin execution (see “The HP3000 Logon Component” above for more detail).

Connection Pooling with a Single Sign-On

If your host system security supports multiple logins from a single user ID, you may have circumstances where you wish to pool the single User ID. This can be accomplished by performing the following steps:

- Specify a User ID/Password in the Connection Resource used by the Logon Component
- On the Pool Info dialog of the Logon Connection, specify a Pool Size greater than one
- Do NOT check the **Override the UID/PWD** setting in the Pool Info dialog of the Logon Connection.

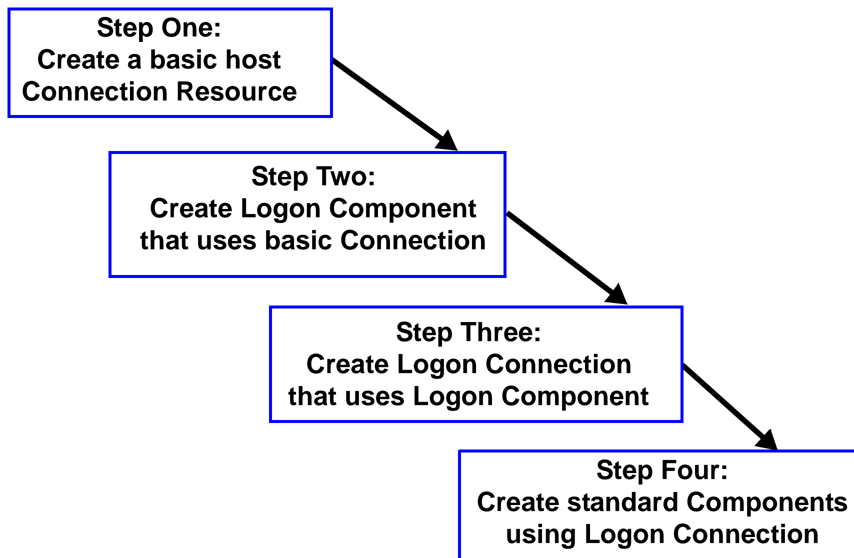
These steps will cause each pool slot to use the User ID and Password contained in the Connection object and not use and User IDs from the pool.

Creating a Connection Pool

Overview

When creating an HP3000 component, you normally create the Connection object it needs first. Similarly, when creating the object comprising a Connection Pool, you must create certain objects first, starting (in essence) at the host and working your way backwards to the HP3000 Terminal Component that will access the host.

A typical sequence of steps for creating a Connection Pool is outlined in the diagram below:



Creating a Basic Connection

This step is simple. Create a new Connection Resource as described in “To create an HP3000 Connection Resource:” on page -14 of this Guide. Even though you will be using User IDs and Passwords defined in the Logon Connection later, you should still define one in the Connection as well. This will be needed when you define the Logon Component in the next step. Alternatively, you can simply use an existing Connection Resource.

Creating a Logon Component

➤ **To create an HP3000 Logon Component:**

1. From the Composer **File** menu, select **New> xObject,** then open the **Component** tab and select **HP3000 Logon**.

   The Header Info panel of the New xObject Wizard appears.

2. Type a **Name** for the connection object.
3. Optionally, type **Description** text.
4. Click **Next** and the Connection Info panel appears.

5. Select a **Connection** from the drop down list. (This will be the basic connection, not the logon connection.)

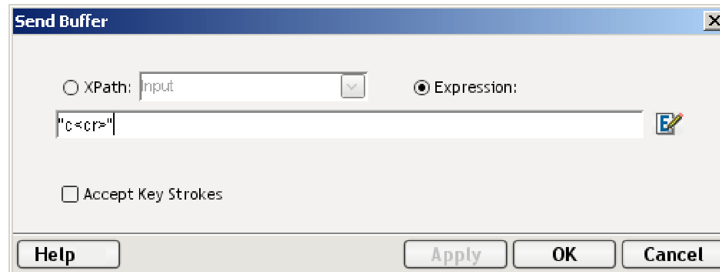
6 Click **Finish** and the Logon Component Editor appears.

NOTE: Recording actions follows a series of steps. The cursor must be positioned over LOGON; then turn Record on, and when you are done, turn Record off. Position the cursor to Keep Alive, turn Record on, and when you are done, turn Record off. Position the cursor to LOGOFF, turn Record on, then when you are done, turn Record off.

7 Record Logon Actions for logging into the host and navigating to the launch screen using the same Recording techniques described in Chapter 4 of this Guide.

8 Edit the Logon Map actions that enter a User ID and Password to instead use the special USERID and PASSWORD variables described in the section titled “HP3000-Specific Expression Builder Extensions” in Chapter 4 of this Guide.

9 Create the needed Send Buffer actions in the Keep Alive section of the Action Model (a quick way is to copy an existing SEND key action, Paste it, and then modify the key code sent).



10 Record Logoff actions for properly exiting the host

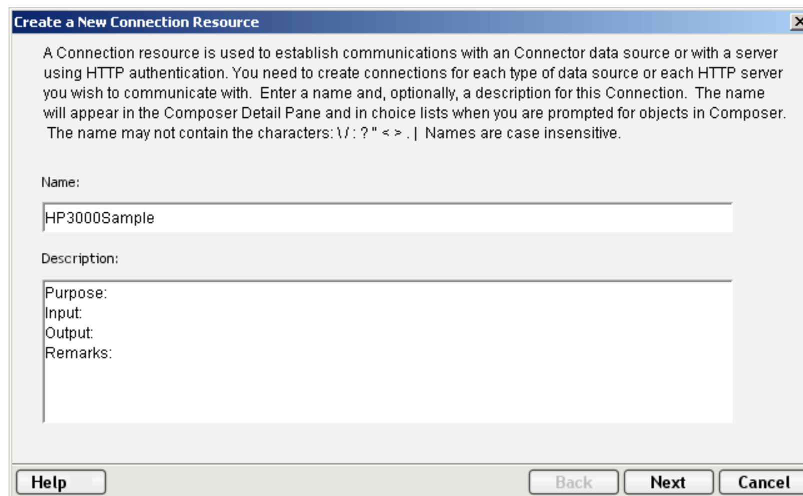
11 Save and close the logon Component.

Creating a Logon Connection using a Pool Connection

➢ To create an HP3000 Logon Connection:

1 From the Composer File menu, select **New> xObject**, then open the **Resource** tab and select **Connection**, or you can click on the icon.

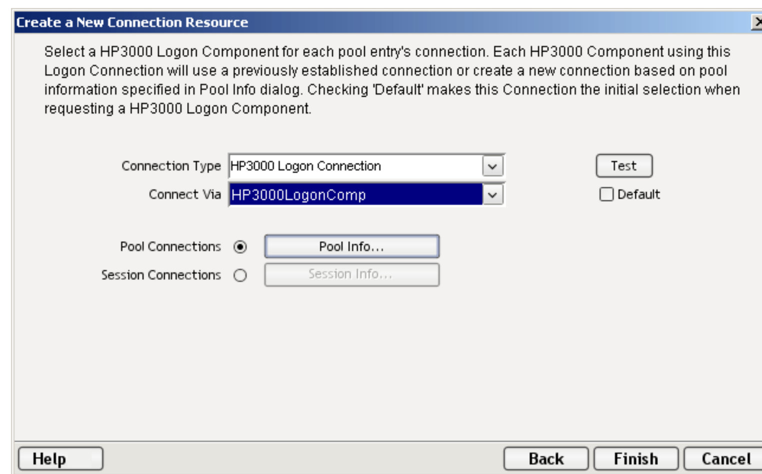
The Header Info panel of the New xObject Wizard appears.



2 Type a **Name** for the connection object.

3 Optionally, type **Description** text.

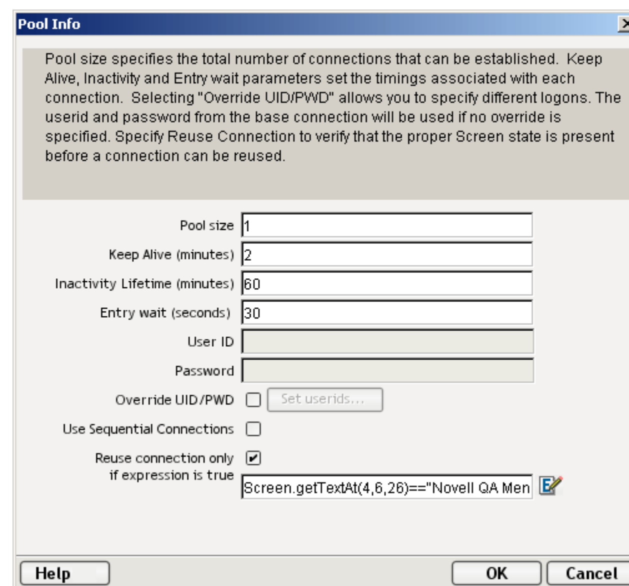
4 Click **Next** and the **Connection Info** panel appears.



5 For the **Connection Type** select "HP3000 Logon Connection" from the drop down list.

6 In the **Logon Via** control, select the Logon Component you just created.

7 Click on the **Pool Info** button and the Pool Info dialog appears.



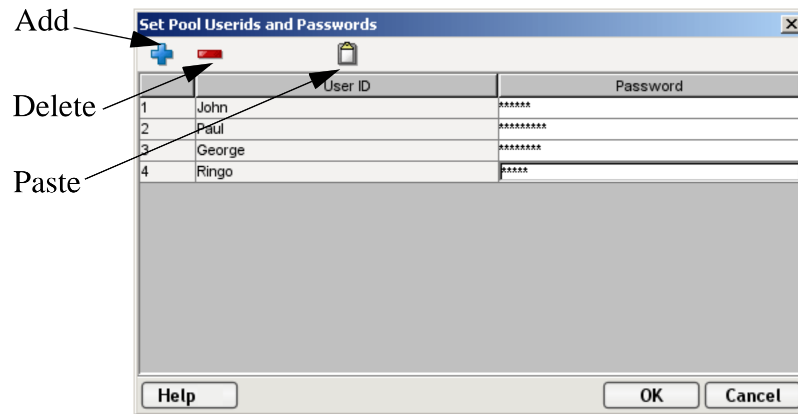
8 Enter a **Pool Size** number. This represents the total number of connections you wish to make available in this pool. For each connection, you will be expected to supply a UserID/Password combination later.

9 Enter a **Keep Alive** time period. This number represents (in minutes) how often you wish to execute the Keep Alive actions in the associated Logon Component whenever the connection is active but free (i.e. not being used by an HP3000 component). The number you enter here should be less than the Timeout period defined on the host for an inactive connection.

10 Enter an **Inactivity Lifetime**. This number represents (in minutes) how long you wish to keep an active free connection available before closing out the connection and returning it to the inactive portion of the connection pool. Remember, that once the connection is returned to its inactive state in the pool, it will incur the overhead of logging in and navigating host screens when it is re-activated.

11 Enter an **Entry Wait** time in seconds. This time represents how long an HP3000 component will wait for a free connection when all the pool entries are active and in use. If this time period is reached, an Exception will be thrown to the Application Server.

12 Checking **Override UID/PWD** means you wish to specify User ID/Password combinations for use in the connection pool. When checked, this activates the Set USERID/PASSWORD button. Click on the button to display the Set USERIDs and PASSWORDS dialog.



On the Toolbar there are three icons: **Add** which adds an empty row, **Delete**, which deletes a highlighted row and **Paste** which allows you to copy/paste information from a spreadsheet into the table. For more on this, see the following Note.

NOTE: Alternate and faster ways to enter data are to copy data from a spread sheet and paste it into the table. Make sure your selection contains at least two columns, UserID and Password. The first and second column must contain data, all other columns will be disregarded. The first number column you see in the screen is automatically generated. Open the spreadsheet, copy the two columns and as many rows as needed. Open the table and immediately press the Paste button. You can also copy data from tables in a Microsoft Word® document using the same technique.

13 Enter as many **USERID/PASSWORD** combinations until you reach the size of the pool you specified and click **OK**. Pool size will be adjusted depending upon how many rows you entered.

14 Click **OK** to dismiss the “Set User IDs and Passwords” dialog and return to the Pool Info dialog.

15 Optionally check the **Use Sequential Connections** control if you want Composer to establish connections in the same order that User IDs were listed in the “Set User IDs and Passwords” dialog. Connections will be made in numerical sequence.

16 Optionally check the **Reuse connection only if expression is true** control. This control allows you to enter an ECMAScript expression that evaluates to true or false based on some test of the launch screen. The purpose of the expression is to check to make sure the launch screen is the proper one each time a new HP3000 Component is about to reuse an active free connection. Under circumstances unrelated to your Composer service, it’s possible that the launch screen will be replaced by the host with a different screen. For instance, if there is a system ABEND on the host, the launch screen in the Logon Component may be replaced by a System Message screen.

The following a is a sample Custom Script used to see if a particular screen is present. If it is not, the script writes a message to the console stating that the screen is bad and the logon connection is being released. This function is called from the “Reuse connect only if expression is true” control on the Pool Info dialog.

```
function checkValidLaunchScreen(ScreenDoc)
{
     var screenText = ScreenDoc.XPath("SCREEN").item(0).text
     if((screenText.indexOf( "MENU") != -1 || screenText.indexOf("APLS") != -1) &&
       (screenText.indexOf("COMMAND UNRECOGNIZED") == -1 ||
         screenText.indexOf("UNSUPPORTED FUNCTION") == -1))
     {
         return true;
     }
     else
     {
         java.lang.System.out.println("Warning - Releasing logon connection at bad screen");
         java.lang.System.err.println("Warning - Releasing logon connection at bad screen");
         return false;
     }
}
```

17 Click **OK** to return to the Connection Info panel.

18 Click on **Finish** and the Logon Connection is saved.

Maximizing Performance of HP3000 Logon Connection

To prevent HP3000 Components from beginning execution on a connection that may have been left on an invalid screen by a previous HP3000 component, the Logon Connection Resource allows the connection itself to check for the presence of the launch screen. This is accomplished by using the option titled “Reuse connection only if expression is true” on the Pool Info dialog of the Logon Connection. The screen test you specify here is executed each time an HP3000 Component completes execution. If the test fails, exteNd Composer will immediately disconnect from the host, possibly leaving a dangling UserID on the host. As noted before, the host will eventually kill the user, but the UserID may be discarded from the pool if it is accessed again before being killed, thereby reducing the pool size and consequently overall performance.

Another reason to use the “Reuse connection only if true” option is that you can perform very detailed tests against the screen to make sure it is your launch screen. While Map Screen actions do perform a screen check, they only look at the number of fields in the terminal data stream. In most cases, this is sufficient. However, it is possible two different screens can have the same number of fields in which case the expression based test that examines the content of the screen will produce more rigorous results. A best practices approach mandates that you use this feature all the time.

Static versus Dynamically Created Documents/Elements

In some Composer applications, users have a need to place various control, auditing, and/or meta-data in an XML document. This document may or may not be in addition to the actual elements/documents being processed (i.e. created from an information source). If this document structure and data is dynamically created by multiple Map actions (i.e. over 100) performance of the component and therefore the entire service may suffer. To boost performance, create the portion of the document structure without the dynamic content ahead of time, then load it into the Service at runtime via an XML Interchange action and retain the Map actions for dynamic content. This can boost performance as much as 30% in some cases.

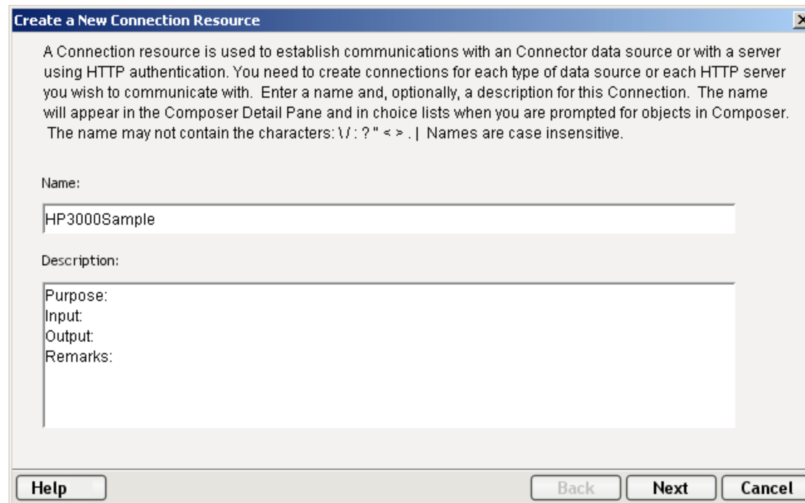
Creating a Logon Connection using a Session Connection

Sometimes, you may want the extra level of control over session parameters that a Logon Connection affords, without necessarily wanting to use pooling. In this case, you can follow the procedure outlined below.

➤ To create an HP3000 Session Connection:

1 From the Composer **File** menu, select **New> xObject,** then open the **Resource** tab and select **Connection**, or you can click on the icon.

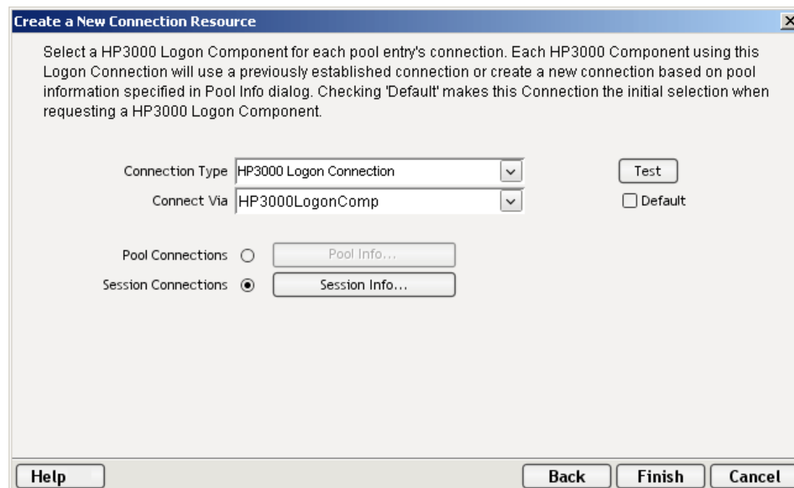
The Header Info panel of the New xObject Wizard appears.



2 Type a **Name** for the connection object.

3 Optionally, type **Description** text.

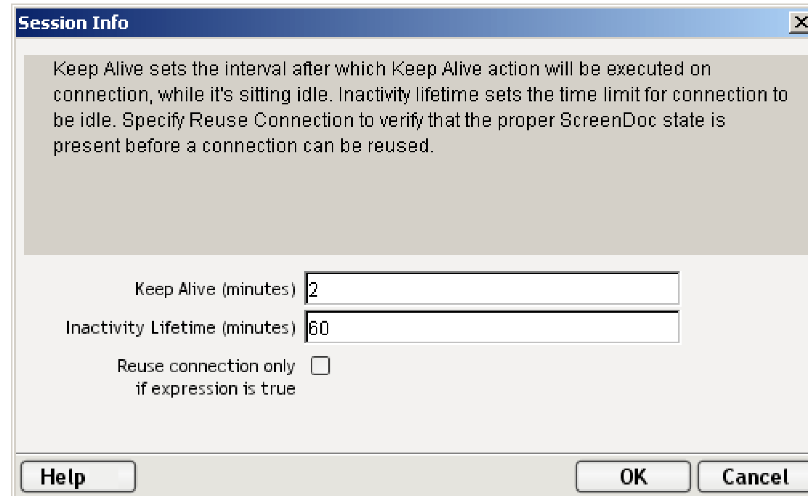
4 Click **Next** and the **Connection Info** panel appears.



5 For the Connection Type select "HP3000 Logon Connection" from the drop down list.

6 In the **Connect Via** control, select the Logon Component you just created.

7 Click the Session Connections radio button and then on **Session Info**.



8 The Keep Alive (minutes) number represents (in minutes) how often you wish to execute the Keep Alive actions in the associated Logon Component whenever the connection is active but free (i.e. not being used by an HP3000 Terminal component). The number you enter here should be less than the Timeout period defined on the host for an inactive connection.

9 The Inactivity Lifetime (minutes) number represents (in minutes) how long you wish to keep an active free connection available before closing out the connection and returning it to the inactive portion of the connection pool. Remember, that once the connection is returned to its inactive state in the pool, it will incur the overhead of logging in and navigating host screens when it is re-activated.

10 Click in the checkmark box if you want to Reuse connection only if expression is true. If you choose to do so, the expression field automatically displays and you can click on the expression icon to display the if the expression is true dialog.

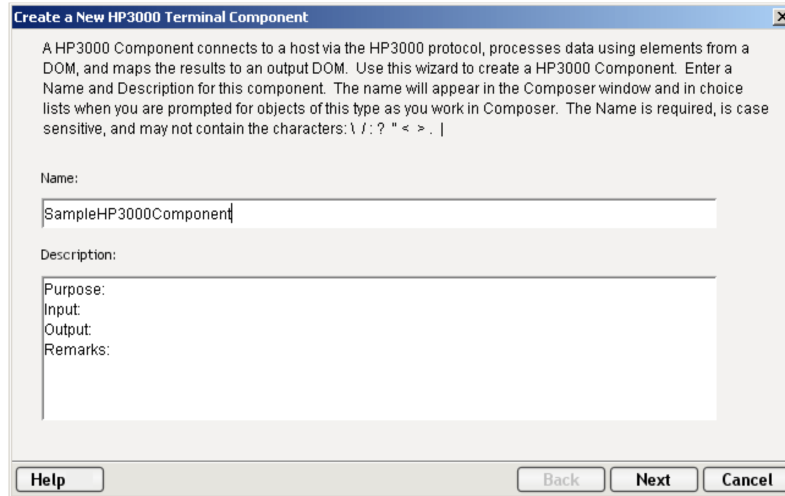
Creating an HP3000 Component That Uses Pooled Connections

At this point, you are ready to create an HP3000 Component that can use the Connection Pool. For the most part, you will build the component as you would a normal HP3000 component, the only difference being the Connection you specify on the New xObject Wizard. (You’ll specify a Logon Connection instead of a regular HP3000 Connection.)

➢ To create an HP3000 Component:

1 From the Composer **File** menu, select **New> xObject**, then open the **Component** tab and select **HP3000**.

The Header Info panel of the New xObject Wizard appears.



2 Type a **Name** for the component.

3 Optionally, type **Description** text.

4 Click **Next** and the XML Property Info panel appears.

5 Select the necessary **Input and Output Templates** for your component.

6 Click **Next**. The Connection Info panel appears.

7 Select the **Logon Connection** you created and click on **Next**. The Component editor appears.

8 Build the component as described “To create a new HP3000 Component:” on page -19 of this Guide.

Maximizing Performance of HP3000 Terminal Components

Once the launch screen is obtained by the logon Component’s logon actions, it is handed to the HP3000 Terminal Component that uses the connection. Then the HP3000 Terminal component (when finished executing) leaves the screen handler back at the launch screen. If the HP3000 Component finishes without being on the launch screen,(i.e. it releases the connection back to the pool with an invalid screen) then it is possible that all subsequent HP3000 Components that use the connection may throw exceptions rendering the connection useless. It also will degrade overall performance and possibly cause data integrity problems within the component processing.

Once again, ensure that the launch screen is present, *the last action to execute in an HP3000 Component must be a Check Screen that checks for the launch screen.* This can be tricky if your component has many decision paths that may independently end component execution. You must be sure that each path ends with a Check Screen action.

Managing Pools

Connections pools can be managed through the HP3000 Console Screen.

➢ How to Access the Console

1 If you are using the Novell exteNd Application Server, log on to your Server via your web browser using **http://localhost/SilverMaster50** (or whatever is appropriate for the version in use). In this example, Novell exteNd App Server 5.0 is used.

SilverMaster50

exteNdComposer
robots.txt
SilverMaster50
SilverStream

NOTE: If you are not using the exteNd app server, enter a URL of this form:

http://<hostname>:<port>/exteNdComposer/Console

2 Click on the **exteNd Composer** link and a list of installed Connects displays to the left of the main console page.

exteNd Composer
Server Console

General Properties
About Products:
3270
5250
CICSRPC
DG
EDI
HP3000
HTML
JDBC
JMS
PROCESS

Log Level: 5
Apply Log Level

Cache Status

Expressions Cached:	2
XPath Nodes Cached:	0
Functions/Code Tables Cached:	0
Component Types Cached:	1
Total Components Cached:	2

Clear Cache

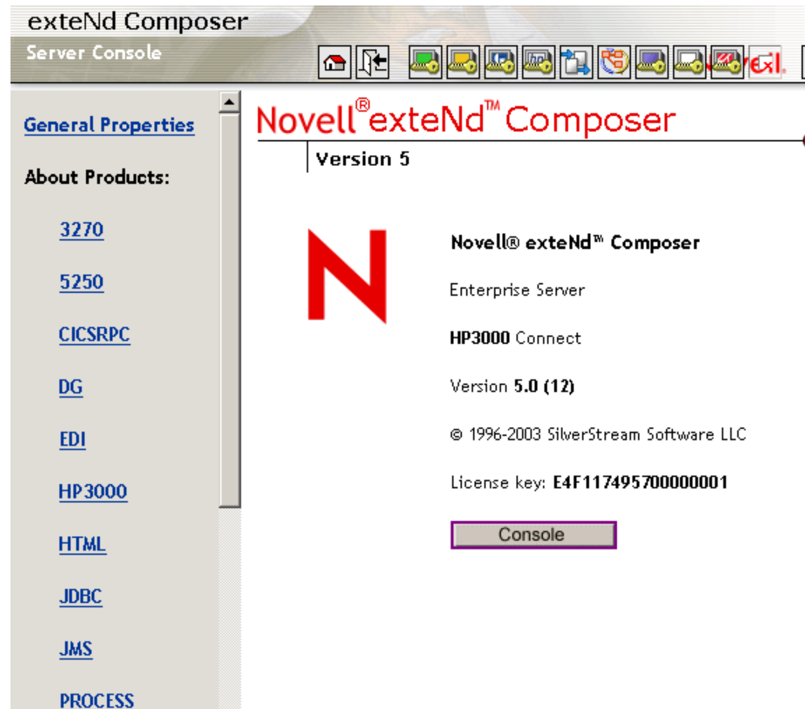
Cache Tuning

Expression / XPath Caching:	On / Off
Component Cache Expiry:	720
Total Component Cache Size:	250

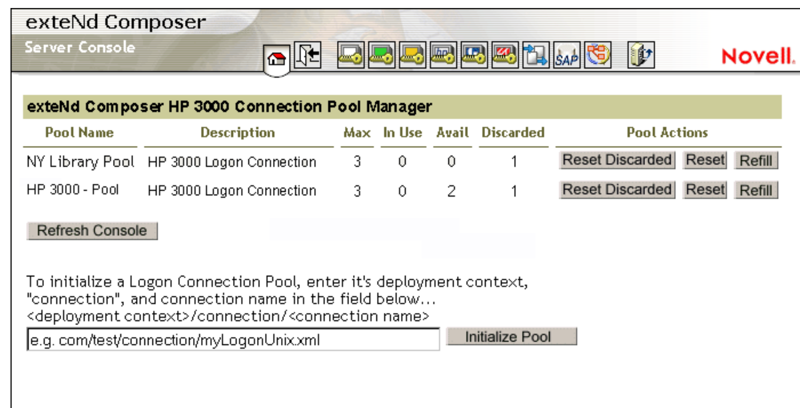
Apply Cache Tuning

©1996-2003 SilverStream Software LLC 2003/06/10 11:02:30

3 Click on the **HP3000** link in the left (nav) frame and the HP3000 General Properties Screen will come into view.



4 Click on **Console**. displays. A browser popup window (the HP3000 Connection Pool Management Screen) should appear.



5 To initialize a Logon Connection Pool, enter its deployment context, the word "connection", and the actual connection name in the text field near the bottom of the screen. (See illustration above.) Then click the **Initialize Pool** button.

NOTE: Refer to the appropriate Composer Enterprise Server guide for more information.

6 Optionally click the **Refresh Console** button to update the view.

Connection Pool Management and Deployed Services

The Connection Pool Management Screen displays the current state of the connection(s) with the HP3000 Connect. The screen contains a table listing the Pool Name, Description of the connection, the maximum number of connections in the pool, the number of connections in use, the number of connections available, the number of connections discarded. It also contains several buttons allowing you to perform various actions related to connection pooling, which are outlined in the table below.

Button Name	Action
Reset Discarded	Resets the Discarded connections which are then reflected in the table
Reset (Pool)	Resets the Available and Discarded connections which are then reflected in the table
Refill (Pool)	Refills the pool with the maximum number of connections
Additional Buttons on HP3000 Connection Pool Manager Console	
Refresh Console	Shows the current status of the connection pool
Initialize Pool	Initializes a Logon Connection Pool by entering a relative path to the deployed lib directory. This will not work unless the deployed jar is extracted. Click on the SUBMIT button when finished.

Connection Discard Behavior

The performance benefits of connection pooling are based on the ability of more than one user to access a resource, or set of resources, at once. The way a connection is established begins with the logon component picking the User ID and Password from the table. If the connection fails, then it is discarded for this User ID and Password and tries another until a connection is established. The failure of one connection doesn't prevent a successful connection to be established.

The Connect for HP3000 addresses the "one bad apple" problem by discarding any connection that can't be established (for whatever reason: bad user ID, timed-out password, etc.) and reusing the others. When a connection is determined to be unusable, the Connect for HP3000 will write a message to the system log that says: "Logon connection in pool <Pool name> was discarded for User ID <User ID>."

Screen Synchronization

Screen synchronization has special ramifications for users of pools. If a situation arises in which a user leaves a connection without the screen returning to its original state, the next user will begin a session with the screen in an unexpected state and an error will occur. To prevent this, we have a screen expression which the user can specify in the connection pool. It is important that the last action in an HP3000 Component be a correct Send Attention Key action that will result in the session ending with the correct logon screen active.

NOTE: The last action should be an empty Check Screen action so that the HP3000 Terminal component waits until the launch screen arrives before giving up the connection.

If you want to check, at runtime, for the presence of a bad screen at the end of a user session, include a Function action at the end of your component's action model that executes a function similar to the one shown below:

```
function checkValidReleaseScreen(ScreenDoc)
{
       var screenText = ScreenDoc.XPath("SCREEN").item(0).text
       if((screenText.indexOf( "MENU") != -1 || screenText.indexOf("APLS") != -1)
&&
          (screenText.indexOf("COMMAND UNRECOGNIZED") == -1 ||
screenText.indexOf("UNSUPPORTED FUNCTION") == -1))
        {
            return true;
        }
        else // Write error messages to
                // System.out and System.err:
        {
    java.lang.System.out.println("Warning - Releasing logon connection at bad
screen");
    java.lang.System.err.println("Warning - Releasing logon connection at bad
screen");
            return false;
        }
}
```

This function checks the screen text and returns false if the final screen is not correct. The check returns true if the screen contains "MENU" or "APLS" and does *not* contain "COMMAND UNRECOGNIZED" nor "UNSUPPORTED FUNCTION."

A Glossary

ANSI American National Standards Institute.

Check Screen An action that action signals the component that execution must not proceed until the screen is in a particular state, subject to a user-specified timeout value.

Dumb Terminal A computer terminal that has no onboard CPU, memory, or storage devices, beyond the minimum necessary to communicate with a more powerful host machine.

ECMAScript Any JavaScript-like language that conforms to European Computer Manufacturers Association standard No. 262.

HP3000 The HP3000 is midrange computer product of the Hewlett-Packard Company.

Native Environment Pane A pane in the HP3000 Component Editor that provides an emulation of an actual HP3000 terminal session.

Screen Object A special DOM in the HP3000 component editor window representing the current HP3000 screen display as an XML document.

Send Buffer An action that appears in the Action Model whenever a there is map to the screen or keys entered on the screen.

TCP/IP Abbreviation for Transmission Control Protocol/Internet Protocol

Type-ahead A technique for preloading a keyboard buffer with more than one screen’s worth of commands.

Terminal Emulation A technique for imitating the runtime behavior of a “dumb terminal” on a desktop (or other) machine.

B HP3000 Keyboard Equivalents

HP3000 Common Keys	Keyboard Equivalent
Arrow Down	Arrow Down
Arrow Left	Arrow Left
Arrow Right	Arrow Right
Arrow Up	Arrow Up
BackSpace	BackSpace
Back Tab	Back Tab
Delete	Delete
Escape	Escape
Linefeed	Linefeed
Return	Return
Tab	Tab
F1	F1
F2	F2
F3	F3
F4	F4
F5	F5
F6	F6
F7	F7
F8	F8

HP3000 NumPad Keys	Keyboard Equivalent
0	Numpad 0
1	Numpad 1
2	Numpad 2
3	Numpad 3
4	Numpad 4
5	Numpad 5
6	Numpad 6
7	Numpad 7
8	Numpad 8
9	Numpad 9
Minus	Numpad -
Comma	Numpad ,
Period	Numpad .
Enter	Numpad Enter

HP3000 Control Keys	Keyboard Equivalent
BS	CTRL+H
CR	CTRL+M
ESC	CTRL+[
HT	CTRL+I
LF	CTRL+J

HP3000 Other Keys	
MENU	No Keyboard Equivalent
SYS	No Keyboard Equivalent
Roll Up	No Keyboard Equivalent
Insert	Insert
Roll Down	No Keyboard Equivalent
Home	Home
Next	Page Down
Prev	Page Up
Number Pad	No Keyboard Equivalent
Clear	No Keyboard Equivalent
Back	No Keyboard Equivalent
New Line	No Keyboard Equivalent
Insert Line	No Keyboard Equivalent
Delete Line	No Keyboard Equivalent
Insert Char	No Keyboard Equivalent
Delete Char	No Keyboard Equivalent
Left Extd Char	No Keyboard Equivalent
Right Extd Char	No Keyboard Equivalent
BREAK	No Keyboard Equivalent
PRINT	No Keyboard Equivalent
Return	No Keyboard Equivalent

C HP3000 Display Attributes

The `Screen.getAttribute()` method will return one of the values shown below, representing the current attribute state of the onscreen character at the given location.

Number	Attribute
0	normal display
1	bold on
2	faint
3	standout
4	underline (mono only)
5	blink on
7	reverse video on
8	nondisplayed (invisible)

Viewing All Character Attributes at Once

Using the `Screen.getAttribute()` method, you can easily write a function that captures all attributes (at all screen locations) at once. The following ECMAScript function, for example, can be used at design time to display screen attributes in an alert dialog.

```
function showAttributes( myScreen )
{
         var attribs = new String(); // create empty string

     // Iterate over all rows and columns:
     for (var i = 1; i <= myScreen.getRowCount(); i++, attribs += "\n")
          for (var k = 1; k <= myScreen.getColumnCount(); k++)
               attribs += myScreen.getAttribute(i,k);
     // display the results:
     alert( attribs );
}
```

The following illustrations show an HP3000 screen and the result of applying the `showAttributes()` function to the screen:

```
Welcome to The New York Public Library's LEO

Library

Entrance

Online

For assistance, press "?" and the key labeled "Return."
To use the catalog, press the key labeled "Return."
```

EcmaScript Alert

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111100000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
11111111111111111111111111114444444444444444444444411000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111110000000000000
11111111111111111111111111111111111111111111111111111111111111111000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

OK

D Reserved Words

The following terms are reserved words in exteNd Composer for HP3000 Connect and should not be used as labels for any user-created variables, methods, or objects.

- USERID
- PASSWORD
- PROJECT
- Screen
- getAttribute
- getCursorColumn
- getColumnCount
- getPrompt
- getRowCount
- getText
- getTextAt
- getTextFromRectangle
- setText

E Java Code Pages

About Encodings

exteNd Composer’s ability to perform character encoding conversions is tied directly to the Java VM in use. The supported encodings vary between different implementations of the Java 2 platform. Sun's Java 2 Software Development Kit, Standard Edition, v. 1.2.2 for Windows or Solaris and the Java 2 Runtime Environment, Standard Edition, v. 1.2.2 for Solaris support. The encodings can be found at the Sun web page:

http://java.sun.com/products//jdk/1.2/docs/guide/internat/encoding.doc.html

Sun's Java 2 Runtime Environment, Standard Edition, v. 1.2.2 for Windows comes in two different versions: US-only and international. The international version (which includes the lib\i18n.jar file) supports all encodings in both tables.

Index

Symbols

$PASSWORD 34

A

About Adding Alias Actions 53
About the HP 3000 Connection 80
Accept Key Strokes 30
Action
  Check Screen 28
Action pane context menu 28
Actions
  Set Screen Text 28
actions, editing 47
Adding A New Action 51
Animation 49, 52
  tools 56
animation 56
applications 11
array, unduplicating an 63
attributes, screen 101

B

binary search technique 59
blank record 66
breakpoint 59
breakpoints 50, 52, 56
buttons, toolbar 26

C

caching screens 66
calculated Timeout 33
Changing an Existing Action 48
Check Screen Action 31
  purpose of 32
Check Screen Actions 28
Clancy, Tom 41
Code Pages
  encodings 105
  support 16
Common Keys 23
comparing screens 66
Connection Discard Behavior 93
Connection Pool Console, refreshing 93
Connection Pool Management and Deployed Services 93
Connection Pooling with a Single Sign-On 82
Connection Pools
  implementing 75
Connection Resource 13, 72
  constant-driven 16
  expression-based 16
  how to create 17
ConnectionPools
  status 93
Connections
  resetting discarded 93
context menus 27
Control Keys
  24
control keys (also see Appendix B) 31
coordinates, onscreen 42
Create Check Screen button 26
createXPath() method 67
Creating a Connection 83
Creating a Connection Pool 82
Creating a Logon Connection 84
Cursor Position 32
Cutting/Copying actions 48

D

debugging 58
Decision Action 44
default Min Wait time 33
default Timeout value 33
Deleting an Action 54
DOM 22
drag and drop 38, 48
dumb terminal 22

E

ECMAScript
  HP 3000-specific methods 34
  unduplicating data with 62
editing an Action Model 47
errors and error messages 58
escape values 31
exceptions 33, 58
Expression Builder
  picklists in 35
Expression Editor 17

F

final screen, detecting 62
Floating Keypad 23
Function Action 64

G

getAttribute() 35
getColumnCount() 36
getCursorColumn() 35
getCursorRow() 35
getPrompt() 36
getRowCount() 36
getText() 36
getTextAt() 36, 38
getTextFromRectangle() 37, 39

H

hard-coded values 65
Host or IP Address 15
hover-help box, escape codes and 31

I

Inactivity Lifetime 79, 85, 89
index variables 65
indexOf() 44, 45
infinite loop 66
Initialize Pool 93
Insert Char and Delete Char 22
ISBN 40, 43, 45
iterating through screens 62

J

join() method 67

K

Keep Alive 89
KeepAlive Actions 75
keepalive actions 76

L

latency 33
LOGOFF Actions 79
logoff actions 76
logon actions 76
Logon Component 75
Logon Connections 14, 72
looping over multiple screens 63
lTrim() 45

M

Managing Pools 91
Maximizing Performance of HP 3000 Logon Connection 87
Maximizing Performance of HP 3000 Terminal Components 90
Maximizing Performance with KEEP ALIVE Actions 79
Maximizing Performance with the Logon Component 77
millisecond timing 69
Min Wait 32
  default of 50ms 33
multiple screens, grabbing data from 61

N

Native Environment Pane 21
newlines, in rectangular screen selections 39
non-printing characters 37
non-printing keys 24
NumPad Keys 22, 23

O

Other Keys 24
Output DOM notes, creating 64
Override the UID/PWD 82
Override UID/PWD 86

P

padded screens 62
PASSWORD global 34
performance tuning 69
picklists 35
Pool Info dialog 78
pool size 85
pools
  checking status 93
  implementing 75
  initializing 93
  refilling 93
  resetting 93
Port 15
profiling 69
PROJECT Variables 16
Prompt 33
prompt string 57
property names 65
pseudocode 64

R

recording 27, 39
rectangular onscreen selections 38
redundant data, dealing with 62
Refill Pool 93
Refresh Consolel 93
RegExp constructor 44
RegExp() 44

regular expressions 44
rejection of duplicates 62
Repeat While action 65
Reset Discarded 93
Reset Pool 93
Reuse connection only if expression is true 89

S

scraping data 61
scraping data from multiple screens 64
screen caching 66
Screen Object 25
  API for all methods 35
screen scraping 11
Screen Selections 37
Screen Synchronization 93
screens, comparing 66
selecting onscreen data 37
Send Buffer
  PASSWORD 27
  USERID 27
Send Buffer Action 30
  creating 30
  exiting 34
  hover help in dialog 31
Set Screen Text Action 28
setText() 37
Shift key down, dragging with 38
shift-drag selection technique 38
split() 45
spoofing, logon 15
Static versus Dynamically Created Documents/Elements 87
Step to Breakpoint 50, 52
strategies for loop termination 61

T

Telnet Connection types 72
Temp XML Document 20
Terminal Type 15
testing 54
Thomas Aquinas 64
Timeout 32, 33
Tips for building HP 3000 Components 56
To create an HP 3000 Component 90
To create an HP 3000 Logon Connection 84
Toggle Breakpoint 49, 52
toolbar buttons 26
troubleshooting 58
type-ahead 41, 58

U

unduplicating records 62
Unicode 16
USERID global 34

W

While (Repeat-While action) 43

X

XPath 30
XSL 11