Developing Custom Knowledge Scripts

NetIQ AppManager

Version 6.0
Legal Notice

THIS DOCUMENT AND THE SOFTWARE DESCRIBED IN THIS DOCUMENT ARE FURNISHED UNDER AND ARE SUBJECT TO THE TERMS OF A LICENSE AGREEMENT OR A NON-DISCLOSURE AGREEMENT. EXCEPT AS EXPRESSLY SET FORTH IN SUCH LICENSE AGREEMENT OR NON-DISCLOSURE AGREEMENT, NETIQ CORPORATION PROVIDES THIS DOCUMENT AND THE SOFTWARE DESCRIBED IN THIS DOCUMENT “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW DISCLAIMERS OF EXPRESS OR IMPLIED WARRANTIES IN CERTAIN TRANSACTIONS; THEREFORE, THIS STATEMENT MAY NOT APPLY TO YOU.

This document and the software described in this document may not be lent, sold, or given away without the prior written permission of NetIQ Corporation, except as otherwise permitted by law. Except as expressly set forth in such license agreement or non-disclosure agreement, no part of this document or the software described in this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, or otherwise, without the prior written consent of NetIQ Corporation. Some companies, names, and data in this document are used for illustration purposes and may not represent real companies, individuals, or data.

This document could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein. These changes may be incorporated in new editions of this document. NetIQ Corporation may make improvements in or changes to the software described in this document at any time.

Copyright © 1995-2004 NetIQ Corporation, all rights reserved.

U.S. Government Restricted Rights: If the software and documentation are being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), in accordance with 48 C.F.R. 227.7202-4 (for Department of Defense (DOD) acquisitions) and 48 C.F.R. 2.101 and 12.212 (for non-DOD acquisitions), the government’s rights in the software and documentation, including its rights to use, modify, reproduce, release, perform, display or disclose the software or documentation, will be subject in all respects to the commercial license rights and restrictions provided in the license agreement. (1 jl)


All other company and product names mentioned are used only for identification purposes and may be trademarks or registered trademarks of their respective companies.
Contents

About this guide .......................................................... 9
Intended audience ......................................................... 9
What’s changed? .......................................................... 10
Using this guide ........................................................... 10
Conventions used in this guide ................................. 11
Where to go for more information .......................... 12
Learning more about NetIQ products ...................... 13
Questions or suggestions? Contact us .................... 14

Chapter 1  AppManager, Knowledge Scripts, and the
Developer’s Console ...................................................... 17

Configuring a Knowledge Script job in the
AppManager Operator Console ...................................... 17
How AppManager processes the Knowledge Script ........ 23
The components of a Knowledge Script .................... 23
Developer’s tools ......................................................... 30
Editing Knowledge Scripts in the Developer’s Console . 31
Different views in the Developer’s Console ................. 35
Testing the sample script ............................................. 38

Chapter 2  AppManager Architecture ............................ 39

A completed Knowledge Script ..................................... 39
AppManager architecture ............................................ 40
Running Knowledge Scripts ....................................... 43
<table>
<thead>
<tr>
<th><strong>Chapter 7</strong></th>
<th><strong>Modifying an action script written in VBScript</strong></th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preliminary discussion</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>Syntax of the Callback functions</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>The program logic</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>The modified script, Samples_HTTPHealthEx.qml</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>Setting up to perform actions</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>Invoking actions</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>Events without actions</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>Ending actions</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>XML messages</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>Listing of the Action_WriteToFile.qml script</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>User-set Script Parameters</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>Parameters supplied by AppManager</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>Functions called in the code</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>Syntax of the Callback functions</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>The program logic</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>The modified script, Action-writeToFileEx.qml</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 8</strong></td>
<td><strong>Modifying an action script written in Summit BasicScript</strong></td>
<td>161</td>
</tr>
<tr>
<td>Listing of the Action_Messenger.qml script</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>User-set Script Parameters</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>Parameters supplied by AppManager</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>Functions called in the code</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>Syntax of the Callback functions</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>The program logic</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>The modified script, Action_MessengerEx.qml</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>DynaDataLog</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>GetAgentInfo</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td>GetContextEx</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>GetJobID</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>GetKPInterval</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>GetMachName</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>GetProgID</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>GetSecurityContext</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>GetTempFileName (VBScript only)</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>GetVersion</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>Item (VBScript only)</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>ItemCount (VBScript only)</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>IterationCount</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>LongDataHeader</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>LongDataLog</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>LongDynaDataLog</td>
<td>268</td>
<td></td>
</tr>
<tr>
<td>MCAbort</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>MCEnterCS</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>MCExitCS</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>MCGetMOID</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>MCVersion</td>
<td>275</td>
<td></td>
</tr>
<tr>
<td>MCWaitForObject (Summit BasicScript only)</td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>MCWaitForObjectEx (Summit BasicScript only)</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>MSActions</td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>MSLongActions</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>NQSleep</td>
<td>285</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 12 \textbf{AppManager Callbacks for Perl} \hfill 291

- AbortScript() \hfill 292
- CounterValue() \hfill 294
- CreateData() \hfill 295
- CreateEvent() \hfill 298
- ExecCmd() \hfill 301
- ExportData() \hfill 303
- ExportHugeData.pl() \hfill 305
- GetJobID() \hfill 306
- GetMachName() \hfill 307
- GetScriptInterval() \hfill 308
- GetTempFileName() \hfill 309
- ImportData() \hfill 310
- ImportHugeData.pl() \hfill 312
- IterationCount() \hfill 313

Chapter 13 \textbf{Testing and debugging} \hfill 315

- Debugging Knowledge Scripts \hfill 315
- Where to debug scripts \hfill 316
- Setting debuggers for VBScript and BasicScript \hfill 316
- The prepend and append files \hfill 318
- Debugging Summit BasicScript scripts \hfill 320
- Debugging VBScript scripts \hfill 320
- Debugging Perl scripts \hfill 321
Chapter 14  Glossary ...........................................323

Appendix A  Dialog Boxes......................................327

Appendix B  Perl Development .................................351
   Compiling your Perl modules ..........................351
   Perl best practices ..................................352

Index ..............................................................361
About this guide

The NetIQ® AppManager Suite (AppManager®) is a comprehensive solution managing and monitoring the performance, availability, and server health for a broad spectrum of operating environments, applications, and server hardware.

AppManager enables system administrators to view all of their servers and workstations from a central, easy-to-use console, providing complete visibility of critical server and application resources across the enterprise. With AppManager, administrative staffs can monitor computer and application resources, check for potential problems, initiate responsive actions, and gather performance data for real-time and historical reporting and analysis.

Intended audience

Developing Custom Knowledge Scripts is intended for system administrators and expert users interested in modifying existing Knowledge Scripts® to provide different or additional information.

Knowledge Scripts referenced in the AppManager documentation and packaged with the product can be used as a basis for building your own Knowledge Scripts, provided that you are an authorized Beta site or licensed customer and you are not engaged in any competitive activities against NetIQ Corporation.

This guide assumes you are at least somewhat familiar with Visual Basic or Perl programming and common programming practices as well as system or operation management. All of the Knowledge Scripts discussed in this guide and used as examples are written in Summit BasicScript, VBScript, or Perl.
Developing Custom Knowledge Scripts

What’s changed?

This book replaces the Developer Guide that was delivered with AppManager 5.0 and earlier versions. Developing Custom Knowledge Scripts covers monitoring, action, and reporting scripts, with detailed examples written in VBScript, Summit BasicScript, and Perl.

Using this guide

Depending on your interests and level of AppManager experience, you may want to read portions of this guide selectively. The following topics are covered:

- Chapter 1, “AppManager, Knowledge Scripts, and the Developer’s Console,” provides an overview of how Knowledge Scripts are used in AppManager and introduces the Developer’s Console.
- Chapter 2, “AppManager Architecture,” discusses the process by which AppManager turns a Knowledge Script in XML format into an executable script that an agent can run.
- Chapter 3, “Knowledge Script basics,” covers the basics of creating a Knowledge Script, with the exception of writing the code.
- Chapter 4, “Modifying a monitoring script written in VBScript,” dissects the code in a sample monitoring script and shows how to modify it to obtain different information. This example is quite simple.
- Chapter 5, “Modifying a monitoring script written in Summit BasicScript,” dissects the code in a sample monitoring script and shows how to modify it to obtain additional information. This example is more complex than the one in the previous chapter.
- Chapter 6, “Modifying a monitoring script written in Perl,” dissects the code in a sample Perl monitoring script and shows how to modify it to obtain different information.
Chapter 7, “Modifying an action script written in VBScript,” dissects the code in a sample action script and shows how to modify it to obtain different behavior.

Chapter 8, “Modifying an action script written in Summit BasicScript,” dissects the code in a sample action script and shows how to modify it to obtain different behavior.

Chapter 9, “Modifying an action script written in Perl,” dissects the code in a sample action script and shows how to modify it to obtain different behavior.

Chapter 10, “Modifying a report script written in VBScript,” discusses the structure of a report script.

Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript,” provides reference information for the Callback functions used in BasicScript and VBScript Knowledge Scripts.

Chapter 12, “AppManager Callbacks for Perl,” provides reference information for the Callbacks used when writing Knowledge Scripts in Perl.


Chapter 14, “Glossary,” defines the terms used in this book.

Appendix A, “Dialog Boxes,” lists and discusses all the fields in the various dialog boxes you can open in the Developer’s Console.


In addition to these chapters an index is provided for your reference.

Conventions used in this guide

The following conventions are used in this guide:

- Fixed-width font is used for source code, program names or output, file names, and commands that you enter.
Developing Custom Knowledge Scripts

- An italicized fixed-width font is used to indicate variables.
- Bold text is used to emphasize commands, buttons, or user interface text, and to introduce new terms.
- Italics are used for book titles.

Where to go for more information

The AppManager documentation set includes several sources of information. These sources are available both as printed books and in Adobe Acrobat (PDF) format:

- **Installation Guide** for complete instructions on installing and configuring AppManager.
- **User Guide** for complete information about running jobs, responding to events, creating reports, and working with all of the AppManager consoles.
- **Administrator Guide** for complete information about managing an AppManager site, setting security, and maintaining the AppManager repository.
- **Knowledge Script Guide** for a brief description of what each Knowledge Script does.

Additional documentation is available in Adobe Acrobat (PDF) format only and includes:

- **Upgrade and Migration Guide** for complete information on how to upgrade from a previous version of AppManager.
- **Knowledge Script Reference Guide** for complete information about each Knowledge Script, including details about setting job parameters.
- **Managed Objects Reference Guide** for technical information about the most commonly used AppManager managed objects. (This guide does not document all AppManager managed objects.)
- **Reporting Guide** for complete information about working with AppManager reporting components, including the NetIQ

The basic AppManager documentation set is available on the AppManager CD-ROM. Additional resources are available on the NetIQ Online Support Web site. In many cases, supplemental, application-specific documentation may be available on the Web. For example:

- NetIQ Work Smarter guides provide tips, advice, and recommendations on special topics, such as improving the performance of the AppManager Operator Console. We recommend you periodically check the NetIQ Online Support site for updated and new NetIQ Work Smarter guides.
- Up-to-date information regarding the versions of products that AppManager supports.

**Note** To access the NetIQ Online Support site, you must be a registered AppManager user.

You may also find supplemental technical documentation for your applications useful. For example, you may want to refer to various Microsoft Resource Kits and Microsoft VBA manuals.

---

**Learning more about NetIQ products**

NetIQ Corporation is a leading provider of intelligent, e-business management software solutions for all components of your corporate infrastructure. These components include servers, networks, directories, Web servers, and various applications.

NetIQ Corporation provides integrated products that simplify and unify directory, security, operations, and network performance management in your extended enterprise. NetIQ Corporation provides the following categories of products:

- **Windows and Exchange Management** The NetIQ Windows and Exchange Management products provide tools for managing, migrating, administering and analyzing your Windows and
Exchange environments. These products include tools for setting and enforcing policies that govern user accounts, groups, resources, services, events, files, and folders, and products that automate time-consuming administration tasks.

- **Performance and Availability Monitoring** The NetIQ Performance and Availability products provide control and automation for monitoring the performance and service availability for your critical servers, applications, and devices, and extensive network monitoring capabilities to provide a complete, end-to-end management solution for e-business infrastructures. These products enable you to pinpoint existing and potential server and network problems and resolve those problems quickly and effectively.

- **Security Management and Administration** The NetIQ Security Management and Administration products enable you to administer, assess, enforce, and protect all aspects of security within your Windows environment. These products provide incident management and intrusion detection, vulnerability assessment, firewall reporting and analysis, and Windows security administration.

- **Web Analytics** The NetIQ Web Analytics products enable you to evaluate and analyze your Web site traffic and performance, as well as manage your visitor relationships.

Questions or suggestions? Contact us...

NetIQ Corporation is a Microsoft Premier Independent Software Vendor, a Microsoft Certified Solution Provider, ADSI Partner, and Microsoft Security Partner and is headquartered in San Jose, California, with offices throughout the United States, Canada, Europe, and Asia.

If you have questions or comments, we look forward to hearing from you. For information about contacting NetIQ, visit our Web site at www.netiq.com/About_NetIQ/ContactUs.asp. From the Web
site, you can get the latest news and information from Technical Support, Public Relations, Investor Relations, and Sales. In addition, you can find our office locations and a list of our current partners.

To fill out an online Technical Support Request form, go to www.netiq.com/Support or e-mail Technical Support directly at support@netiq.com.

For comments or suggestions regarding the documentation or online help, send an e-mail to documentation@netiq.com.
Chapter 1

AppManager, Knowledge Scripts, and the Developer’s Console

This chapter provides an overview of the way in which AppManager uses Knowledge Scripts and an introduction to the Developer’s Console. The following topics are covered:

- Configuring a Knowledge Script job in the AppManager Operator Console
- How AppManager processes the Knowledge Script
- The components of a Knowledge Script
- Developer's tools
- Editing Knowledge Scripts in the Developer's Console
- Different views in the Developer’s Console
- Testing the sample script

Configuring a Knowledge Script job in the AppManager Operator Console

If you are going to develop your own Knowledge Scripts, you should already be familiar with the use of Knowledge Scripts in the AppManager environment. This section provides a brief review of the process, covering the steps that will be important for you to understand as a developer.

An AppManager agent (software developed by NetIQ) on a managed client (a managed computer) runs Knowledge Script jobs on that computer. These jobs are requested by an Operator Console user who:
- chooses the Knowledge Script that performs the task or tasks that the user wants performed, and
- sets the properties of the job—such as the frequency with which the job should run, thresholds that should not be exceeded, whether or not to raise events, and so forth.

The user accomplishes these tasks in the AppManager Operator Console by:

1. selecting the desired script in the **Knowledge Script** pane,

2. dragging the script to the target object on which it should operate (a computer, a hardware component like a disk drive, an application, and so forth) in the **TreeView** pane,

3. dropping the script on the target object, which opens a **Properties** dialog box for the script, and

4. setting the job properties in the **Properties** dialog box.

When the user clicks **OK** to close the **Properties** dialog box, the Knowledge Script becomes a “job” and is run by the AppManager agent on the target computer.

**Note** In this book, the term **target computer** refers either to the computer that is itself the target object for a script, or to the computer that contains the target object (when the target object itself is a hardware device like a CPU, or a software application or service). Refer to Chapter 14, “Glossary,” for more definitions of terms used in this book.
Visually, the process occurs like this:

**Step 1.** The user selects the Knowledge Script to be run and drags it to the target object in the Operator Console’s TreeView pane. In the case below, the script is a sample script (Samples_HelloWorld.qml, in the Samples Knowledge Script Group) and the target object is a computer.

Note that, during the drag, the target object icon has changed from its normal state to a green disc. This indicates that it is legal to drop this particular script on this target object. At the same time, the icon for the CPU immediately below the target computer in the TreeView pane has changed from its normal state to a left-pointing green arrow,
indicating that it is legal to drop this script on a target object higher up in the **TreeView** pane.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Normal State</th>
<th>During Drag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target object (a computer).</td>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>CPU in the target computer, a component lower in the <strong>TreeView</strong> pane.</td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
</tbody>
</table>

**Note**  The changing of icons to green, indicating where it is legal to drop this script, is a manifestation of “object type checking.” Briefly, every Knowledge Script contains an “object definition” that determines which target objects are legal for this script. The Operator Console software will not permit you to drop a script on the wrong type of object.

**Step 2.** Given the green disc, the user drops the Knowledge Script on the target object. The Knowledge Script **Properties** dialog box then opens.
Chapter 1: AppManager, Knowledge Scripts, and the Developer’s Console

The Schedule tab allows the user to set the frequency with which the job is to be run. In this case the default is Run once. The user can set a different schedule, or accept the default.

Note The default schedule is not always the same. You, the script developer, choose the default for your script.

Perhaps the most important tab in the Properties dialog box is the Values tab. The Script Parameters in your script that you have chosen to be user-definable are listed in this tab. The user may elect to accept the default values or to change them.

Caution When an Operator Console user enters values for Script Parameters, the Operator Console does not do any input validation. Your code must always be written so that it can handle user input errors, including no input.

You, the developer, create the Script Parameters that users can give values to, and you choose the defaults for those Script Parameters as well—when you create your script. You also define the range of possible values for the Script Parameters. For example, in Samples_Helloworld.qml, DO_EVENT can only take on two values, “y” or “n”.

![Properties for Samples_Helloworld](image)
In the case of this script, there is only one Script Parameter that the user can set:

<table>
<thead>
<tr>
<th>Script Parameter</th>
<th>Description</th>
<th>Possible Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raise events?</td>
<td>Should the script raise events?</td>
<td>y (yes) or n (no)</td>
<td>y</td>
</tr>
</tbody>
</table>

In general, the user is not shown the name of the Script Parameter he or she is setting. When you create the Knowledge Script, you will associate a variable name with each Script Parameter so that you can access the user-specified value. The user will be shown a user-friendly description. For example, in `Samples_Helloworld.qml`, the user will see "Raise events? [y/n]" while the variable name associated with this parameter is `DO_EVENT`.

**Step 3.** When the user clicks **OK** and closes the **Properties** dialog box, the job begins to run.
Recall the green disc that appeared when the script was being dragged and dropped. Once the job begins to run, the original icon reappears, with a blue capital J (“J” for job) superimposed. A job is, by definition, a Knowledge Script that has begun to run.

**Note** If an event has been raised by a script, the target object’s icon will blink alternately with an error icon—a disc whose color indicates the severity of the error. Your script defines what that severity is.

### How AppManager processes the Knowledge Script

In the process described above, the Knowledge Script is transformed by the AppManager infrastructure to generate a final script that the AppManager agents can run. In this transformation, AppManager:

1. Parses the XML elements of the Knowledge Script.
2. Leaves the code section as is.
3. Adds constants for the AppManager and Knowledge Script version numbers at the beginning of the code.
4. Gives the object type variable a value and adds it at the beginning of the code.
5. Converts the Script Parameters to variables with defined values (VBScript, Perl) or constants (Summit BasicScript), and adds them at the beginning of the code.

This transformation generates an executable script that is sent to the AppManager agent, along with scheduling information (not part of the generated script), as a job to be run.

### The components of a Knowledge Script

Knowledge Script code is written in:

- Summit BasicScript (older scripts for managing Windows computers),
● VBScript (more recent scripts for managing Windows computers),
or
● Perl (scripts for managing UNIX computers).

Each Knowledge Script written by you or others (as checked into the AppManager repository), is an XML file that consists of two qualitatively different components:

1. numerous non-code XML elements at the beginning of the script
2. an XML element that contains the code (the last element in the file).

Such scripts have a “.qml” extension (for “NetIQ XML”).

Note Many older scripts have an “.ebs” extension. These scripts are not written in XML. However, if you open and then save such a script in the Developer’s Console (see “Developer’s tools” on page 30, below), it will be converted to an XML file with a .qml extension.

**The non-code XML elements of the Knowledge Script**

The non-code XML section that precedes the code element contains:

- All of the Script Parameters (thresholds, DO_DATA, DO_EVENT, etc.) for which the user can set values, along with their variable type, range (if any), and default value.
- The schedule for running the script, with a default value.
- The resource object type or types for the script.
- The names of action scripts to be executed, if any (usually chosen by the user).
- The name of the scripting language used.
- Several other elements, as discussed later.

**Note** You should not edit the non-code XML section directly. The Developer’s Console includes a user interface (the Script Properties dialog box, opened from the View menu) for entering and modifying these non-code XML elements.
The code component of the Knowledge Script

This code component of the script is written by you. It is itself a large XML element, although you do not need to concern yourself with the XML tags. It will interact with the non-code XML elements in the script, so you must be aware of them. For example, some of the constants or variables in your script can have their values set by the user. These values will replace the defaults in the non-code XML <parameter> element if a user chooses to set them.

**Note** The Script Parameters that can be set by the user will become variables (or constants, in the case of Summit BasicScript) in your code.

Your Knowledge Script code can contain the following:

- Any logic allowed by the language you have chosen to use.
- Any built-in functions of the language you have chosen to use.
- Script Parameters to which a user can give values, for example `DO_EVENT`. You use these Script Parameters as constants or variables in your code *just as if you had declared them and had assigned values to them*. When the final agent-runnable script is generated, these Script Parameters will be included as defined constants in Summit BasicScript and as variables with values assigned in VBScript or Perl scripts.
- Any other variables or constants of your choice.
- Managed object methods (see the Managed Object Reference Manual). These methods are the “workhorses” of the scripts—you use them to get system information about managed hardware or process information about services and applications. In VBScript and Summit BasicScript, managed objects are COM objects that contain methods that you can call. In Perl, managed objects are Perl modules.
- Callback functions (see Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript” and Chapter 12, “AppManager Callbacks for Perl”). These functions are called by your script to request information or action *from the AppManager agent that is*
running the script. For example, you use a Callback function to raise an event.

- Exception handling.

**A sample Knowledge Script**

Here is a listing of the entirety of a very simple Knowledge Script called `Samples_HelloWorld.qml`, which can be found on your AppManager CD in `appmanager\documentation\development_tools\developer_guide\scripts`.

`Samples_HelloWorld.qml` is written in VBScript (the default language). It raises an event with a message of “Hello World!” every time the script executes (default = every 2 minutes).

The lines of asterisks (for example, `*******comment**********`) are not part of the script. They have been added to show you the boundaries of the non-code XML elements and the code element.

**Note** It is not necessary to try to understand this script at this time. It is here just so you can see what a complete Knowledge Script contains.

```
beginning of the XML file*****
the non-code XML elements**********

<PROLOGUE>
<![CDATA[
'### Copyright (c) 1995-2002 NetIQ Corp. All rights reserved.
'###
'### Samples_HelloWorld.qml
'### This script, that illustrates sending events,
'### is used as an example in the Developer Guide.
]]>
</PROLOGUE>

<KSID>
<Type>Regular</Type>
<Name>Samples_HelloWorld</Name>
<Desc>Sample Knowledge Script for raising events.</Desc>
-Version>
  <AppManID>4.0.15.1</AppManID>
  <KSVerID>1.0</KSVerID>
</Version>
```
<NeedPWD>0</NeedPWD>
<AdminOnly>0</AdminOnly>
<UnixOnly>0</UnixOnly>
<DataSrcID>0</DataSrcID>
<Platform>-1</Platform>
</KSID>

<ObjType v3style="1" fullpath="0" dropfolderlist="0">
  <Type name="NT_MachineFolder"></Type>
</ObjType>

<Schedule>
  <Default type="interval" runmode="sched">
    <Interval>
      <Hour>0</Hour>
      <Minute>0</Minute>
      <Second>120</Second>
    </Interval>
  </Default>
  <Allowed>
    <RunOnce>1</RunOnce>
    <IntervalIter>1</IntervalIter>
    <Daily>1</Daily>
    <Weekly>1</Weekly>
    <Monthly>1</Monthly>
  </Allowed>
</Schedule>

<DataSrc></DataSrc>

<Parameter>
  <Desc>Set the Event property to y to generate events.</Desc>
  <Param name="DO_EVENT">
    <Desc>Raise events? (y/n)</Desc>
    <Type>String</Type>
    <Size>1</Size>
    <Range>ynYN</Range>
    <Value>y</Value>
    <ReqInput>0</ReqInput>
    <Folder>0</Folder>
    <NoQuote>0</NoQuote>
  </Param>
  <Param name="AKPID">
    <Desc>Action, if any</Desc>
    <Value>AKP_NULL</Value>
  </Param>
</Parameter>
Developing Custom Knowledge Scripts

When you look over the above script in its entirety, it is evident that this Knowledge Script file cannot be executed as it is because the non-code XML portions are not syntactically correct. Even the syntactically correct code cannot be executed because **DO_EVENT** is neither declared nor assigned a value.
After the Operator Console user has created a job by dragging and dropping a script on a target object (or objects) and then setting its properties, and before the script is executed, the AppManager infrastructure will generate a final script that the AppManager agent can run. In the generation process, some of the non-code XML elements in the Knowledge Script will be converted into actual code. For example, the user-set Script Parameters are converted and prepended to the beginning of the code section. The result is a generated script, which is a complete and executable script that is sent to the AppManager Agent (or Agents) to be run as a job. This was briefly described in “How AppManager processes the Knowledge Script” on page 23. You will see, in the next chapter, examples of each stage in the process of converting a Knowledge Script into a running job.
Developer’s tools

When you purchase a Developer license, you receive a suite of tools and help files that you can use to create custom Knowledge Scripts. You install these tools and help files when you select the Developer’s Console Utilities component in the setup program.

![Developer’s tools image]

The main utilities installed are the Developer’s Console, the Knowledge Script Editor, and the Icon Manager.

**Developer’s Console**

Using this console, you can:

- Automatically generate values for non-code XML elements of your Knowledge Script, using the **Script Properties** dialog box.
- Enter the code in the executable part of the script.
- Check Knowledge Scripts out of and into the AppManager repository.
- Debug scripts written in VBScript.
The Developer's Console also allows you to convert older existing Knowledge Scripts (with a .ebs extension) to the newer .qml format. Simply open the .ebs script in the Developer's Console and then save it. It will be automatically converted to the newer XML format and saved with a .qml extension. Alternatively, you can convert entire directories of 3.x scripts by using the Migrate command on the Tools menu.

**Caution** After you migrate an .ebs file to a .qml file, you must be sure that the `<AppManID>` element in the new .qml file is set to version 4.0 or later (for example, `<AppManID>4.0</AppManID>`). You will need to edit the .qml file in a text editor to accomplish this—the `<AppManID>` element is not accessible through the Script Properties dialog box.

**Knowledge Script Editor**

You can use the Knowledge Script Editor to write and debug Summit BasicScript code directly (just the code, not the rest of the file).

**Icon Manager**

Use this utility to add custom icons and object types to the AppManager repository for customized discovery scripts (not covered in this book).

**Editing Knowledge Scripts in the Developer’s Console**

**Opening the Developer’s Console**

To open the Developer's Console, choose Program Files > NetIQ > AppManager > Developer’s Console > Developer Console from the Windows Start menu.

You have several options for editing Knowledge Scripts:

- Create a new Knowledge Script.
- Edit an existing Knowledge Script.
Developing Custom Knowledge Scripts

- Copy an existing Knowledge Script, rename it, and then edit it to create a new script with modified behavior.

In all cases, the Knowledge Script you are editing must be checked out of the AppManager repository. After completing your edits, you must check the script in to the AppManager repository before you can run it as a job.

Opening Files

Choose Open from the File menu to open a .qml or .ebs file. The Developer's Console will automatically sense the language that the script is written in (from the XML <Script> element) and will open with the script in edit mode.

Warning You must save any script you are editing in the Console before you open another. If you use the Open command to open a different file, the current file will be closed automatically and you will lose your edits if you haven’t saved the file.

Alternatively you can double-click a Knowledge Script file in Windows Explorer:

- If you double-click a file with a .qml extension in Windows Explorer, the Developer's Console will open automatically with that file in edit mode.
- If you double-click on a file with a .qml extension when the Developer's Console is already open, a new instance of the Console will open.
- If you double-click a file with an .ebs extension, the Developer's Console will not open. The Knowledge Script Editor, which is used uniquely for .ebs scripts, will open instead.

Checking out scripts for editing

If the Knowledge Script already exists in the AppManager repository and you want to edit it, you should check it out (use the right-click menu in the AppManager Operator Console) before editing. Checking
it out will automatically open the Developer’s Console with the script in edit mode. Then, choose Check In Knowledge Script from the Tools menu to check your script back in when you are finished with your edits.

**Note** If you have difficulty checking a script in from the Developer’s Console, you can check it in from the AppManager Operator Console using the Check In Knowledge Script... command on the KS menu (or on the right-click menu). This will overwrite the previously checked-in version.

### Copying, renaming, and checking in scripts

If you want to use an existing Knowledge Script as the basis of a new (modified) script, you should copy it and rename it before you do your modifications.

To copy an existing Knowledge Script and check it in with a new name, do the following:

1. In the Knowledge Script pane of the AppManager Operator Console, highlight the file you want to copy.
2 With the cursor on the highlighted file, open the right-click pop-up menu and choose *Copy Knowledge Script...*. The *Copy Knowledge Script* dialog box will open:

![Copy Knowledge Script Dialog](image)

3 Rename your file as desired, change the description, and click **OK**.

4 The new Knowledge Script appears in the *Knowledge Script* pane of the AppManager Operator Console—it is *automatically checked in* to the AppManager repository.

To edit the new file, you will need to check it out to a directory of your choice. Then you must check it back in when you are finished with your modifications.

**Saving and checking in scripts**

To save and check in a script:

1 Choose **Save** or **Save As** from the *File* menu to save your script to any directory.

2 Choose **Check In Knowledge Script** from the *Tools* menu to check your script in to the AppManager repository. If the script has
never been checked in before, you will see it appear in the appropriate category in the AppManager Operator Console.

**Different views in the Developer’s Console**

The screen below shows the View menu of the Developer’s Console when the Console is opened to begin development of a new script in the default scripting language, VBScript.

In the central section of this menu are the views that you will use during development. The views are:

- **Edit**—This is the default view where you see only your code, which you can edit.
- **XML (Read-only)**—In this view, you see the entire XML file, the same way you would see it if you opened it in a text editor. You cannot edit the file in this view. (You are not supposed to edit the non-code XML portion—you change that through the Script Properties dialog box that opens when you select Properties from the View menu.)
- **VB Script/BasicScript/Perl (Read-only)**—In this view, you see the script as it will appear when it is run by the AppManager agent,
except that the values that the user can choose are still set to the
script defaults and the object type values are not yet assigned.

The generated script that gets executed in the AppManager agent will
look exactly like the script shown in this view except that the object
type will be filled in.

**Note** Not all of the information in the non-code XML elements goes
into the running script. For example, the information about the
schedule is sent to the AppManager agent along with the script, but it
is not part of the script.

Here is the sample script, `Samples_HelloWorld.qml`, that was listed
earlier, as it appears in the different views:

**Edit view**

This is what you see in the **Edit** view:

```vbnet
Sub Main()
    Dim strShortMsg

    If DO_EVENT = "y" Then
        ' Event message displayed in the List pane
        strShortMsg = "Hello World! "
        ' raise an event
        NQEXT.CreateEvent 2, strShortMsg, AKPID, ",", 0, ",", ",", 0, 0
    End If
End Sub
```

**XML (Read-only) view**

The complete listing that you saw in “A sample Knowledge Script” on
page 26 is what appears in the **XML (Read-only)** view (except that
the comment lines, such as `****the non-code XML elements******`,
will not be present).

**VB Script (Read-only) view**

In the **VB Script (Read-only)** view, this is what you see:
Note that this is a script that the AppManager agent can run. It does not yet have the values that a user might choose. However, it does have the default values and you can run it as a job. Also note that the lengthy section of non-code XML elements has been replaced with a much shorter section of executable script.

**Note** The “KPV Section” that contains `Sub KS_INIT()` does nothing—it is reserved for future use.
Testing the sample script

If you want to run Samples_Helloworld.qml to test what you have learned, you must first check it in to the AppManager repository, as follows:

1. Open the script in the Developer's Console. This script, and other sample scripts, are located on your AppManager CD, in appmanager\documentation\development_tools\developer_guide\scripts.

2. Check in the script by choosing Check In Knowledge Script from the Tools menu. The script should appear in the Samples tab of the Knowledge Script pane of the Operator Console.

Note: If check-in fails using the Tools command, you can check in the file directly by using the right-click menu in the Knowledge Script pane of the Operator Console.

Once the Knowledge Script has been checked in, you can run it. You might find it interesting to explore the effect of changing schedules or adding actions to the script. You can do this either of two ways:

- with the Script Properties dialog box in the Developer's Console (in this case, you must check the script out and back in each time you modify it) or
- using the Properties dialog box that opens in the Operator Console after you have dropped the script on its target object.
Chapter 2

AppManager Architecture

The discussion in this chapter describes the AppManager architectural elements used in processing and running Knowledge Scripts. It should be helpful in understanding the more subtle aspects of writing scripts. Complete mastery of this material is, however, not essential for modifying existing scripts.

This chapter provides information about the life history of a Knowledge Script—from the time it was checked in to the AppManager repository as a completed script to the time it begins to run as a job.

The following topics are covered:

- A completed Knowledge Script
- AppManager architecture
- Running Knowledge Scripts
- Example
- Where each part of the running script came from

A completed Knowledge Script

After you have finished creating or modifying a script, you check it in to the AppManager repository. As you know (see “The final, generated script,” on page 28), your checked-in Knowledge Scripts are not yet executable.

All pre-existing scripts are stored in various tables of the AppManager repository. You can find any script in your \AppManager\qdb\kp
directory, where they are in subdirectories according to type. For example, WinNT scripts can be found in `\AppManager\qdb\kp\nt`.

**Note** The `qdb` in `\AppManager\qdb` is just the name of a directory—it does not reflect the contents of the AppManager repository. During the AppManager installation, scripts in the `kp` directory tree were checked in to the AppManager repository. Subsequently, any changes made (by checking scripts out, altering them, and checking them back in) affect the Knowledge Scripts stored in the AppManager repository, but those changes are *not* reflected in the `.qml` files in the `\AppManager\qdb\kp` directory tree *unless* you checked them out to that directory.

When a user creates a job from your Knowledge Script, AppManager retrieves the script from the repository and processes it through a series of steps to generate an executable script that the AppManager agent can run on the target computer. To understand this series of steps, you need to know a little about the AppManager architecture.

**AppManager architecture**

As discussed in Chapter 1, the Operator Console creates Knowledge Script jobs to be run by the AppManager agent on the target computer.
The following drawing shows a more detailed view of the AppManager architecture to explain how Knowledge Scripts are processed and run. The drawing does not represent the only possible AppManager configuration—for example, the three components shown as AppManager can be on the same server, as shown, but they do not need to be. Also, the components on the Managed computer have been simplified somewhat to facilitate discussion.
Developing Custom Knowledge Scripts

AppManager components

1 The Operator Console is the user interface for AppManager, and connects to the AppManager repository.  

Note The AppManager repository is called “QDB” by default, although it can be given any name during installation.

2 The repository is very important—it is the center of the AppManager world. The repository server provides a central store of information including Knowledge Scripts, events, graphs, and jobs (instances of running Knowledge Scripts). The job tables include the various pieces of your scripts, and other information such as scheduling.
3 The Management Service is responsible for transferring jobs created by the user to the AppManager agents on managed systems. It is also responsible for forwarding the events and data generated by jobs from the agents back into the AppManager repository.

**Managed computer components**

1 The AppManager agent performs a variety of tasks:
   - It runs scripts (jobs).
   - It has a local repository where it stores scripts, schedules, and actions.
   - It communicates with the AppManager management server.

2 The managed objects are installed on the managed computer along with the AppManager agent, and are called by the scripts being run by the agent. They are COM objects or Perl modules containing methods that are specific to particular applications and are used to retrieve information about the monitored system or application that the script cannot obtain for itself.

**Running Knowledge Scripts**

These are the steps that the script undergoes when it is converted from a Knowledge Script stored in the AppManager repository to a job running on a target computer:

1 In the Operator Console, a user chooses a Knowledge Script, drags it, and drops it on the target object.

2 The Properties dialog box opens.

3 The user sets Script Parameters, execution schedule, actions, and so forth—or accepts the defaults. Then the user clicks **OK** to close the dialog box.

4 The Operator Console fills in values for the object types.
The Operator Console creates an instance of the script (a job) in the repository. The job is an instance of the script that includes the user defined Script Parameter values, the schedule, the object types, and so forth. This final script has all Script Parameters (including AKPID) and object types defined as constants in BasicScript and variables in VBScript and Perl.

The job is forwarded to the AppManager agent on the target computer to be run as a job. Scheduling information (not part of the script) is also sent to the agent, as is information about Actions to perform.

All of the information about the job is also held in the AppManager repository, along with pointers to any action scripts that are to be run on the AppManager management server.

Once the job starts to run, you can see the entire running script by double-clicking on the job's child. This will open the Properties dialog box, where there is now a button you can click to see the running script:
Example

As an example, let’s run the Samples_Helloworld.qml script discussed in the previous chapter, accepting the defaults in the Properties dialog box: Here is the final script as it will run:

```qml
Sub KS_INIT()
End Sub

Sub Main()
Dim strShortMsg
If DO_EVENT = "y" Then
    "Event message displayed in the List pane
    strShortMsg = "Hello World!"
    " raise an event
    NQEXT.CreateEvent 2, strShortMsg, AKPID, "", 0, "", "", 0, 0
End If
End Sub
```

Compare this script with the VB Script (Read-only) view in the Developer's console:

```vb
Const AppManID = "4.0.15.1"
Const KSVerID = "1.0"

Sub KS_INIT()
End Sub
```

```vb
NT_MachineFolder = "SJCRISSERT01"
```

```vb
Sub Main()
Dim strShortMsg
If DO_EVENT = "y" Then
    "Event message displayed in the List pane
    strShortMsg = "Hello world!"
    " raise an event
    NQEXT.CreateEvent 2, strShortMsg, AKPID, "", 0, "", "", 0, 0
End If
End Sub
```
Comparing the two scripts, you will see only one difference in the code—a value has been filled in for the object type, `NT_MachineFolder = "SJCRISSERT01"` (the name of the target computer) in the running script.

If we had used the Properties dialog box to change the value of the `DO_EVENT` Script Parameter, rather than accepting the default, we would have seen the changed value in the “KPP section” of the running script as well.

Changes to the schedule will not appear in the running script, as scheduling information is not part of the final script. AppManager sends the scheduling information to the AppManager agent independently of the script.
Where each part of the running script came from

Apart from Sub Main(), everything in the running script was generated by AppManager from the other non-code XML elements of the Knowledge Script. Here is a brief view of where each section came from:

<table>
<thead>
<tr>
<th>Running Script Section</th>
<th>Origin</th>
</tr>
</thead>
</table>
| '### Begin KSID Section  
Const AppManID = "4.0.15.1"  
Const KSVerID = "1.0"  
'### End KSID Section | The two constants came from the <KSID></KSID> non-code XML element. |
| '### Begin Type Section  
NT_MachineFolder = "SJCRISSERT01"  
'### End Type Section | The name "NT_MachineFolder" came from the <ObjType> non-code XML element. The value ("SJCRISSERT01", which is the actual name of the target computer) was filled in by the Operator Console program. |
| '### Begin KPV Section  
Sub KS_INIT ()  
End Sub  
'### End KPV Section | This section is reserved for future use. At the present time, Sub KS_INIT () does nothing. |
| '### Begin KPP Section  
DO_EVENT="y"  
AKPID="AKP_NULL"  
'### End KPP Section | These values came from the <Parameter> non-code XML elements. Any changes by the user during job creation would appear here. See the note below about AKPID. |

Sub Main()()
Dim strShortMsg
If DO_EVENT = "y" Then
    strShortMsg = "Hello World! "
'Event message displayed in the List pane  
'raise an event
    NQEXT.CreateEvent 2, strShortMsg, AKPID,"", 0, ",", 0, 0
End If
End Sub

Anything that was in non-code XML elements that does not appear in the running script is used by AppManager in some other way. For
example, all of the information about scheduling is forwarded to the AppManager agent but is not part of the script.

**Note** If a user added one action script during job creation, `AKPID` would have the value `AKPID= "1"`. If two jobs were added, it would be `AKPID= "1,2"`. These values are IDs for the action scripts to be run. AppManager can determine which action scripts need to be run from the IDs, although the running script itself has no knowledge of what the action scripts are.

As with schedules, you do not need to write code to handle actions. You do, however, need to define `AKPID` in the **Parameters** section of the Developer's Console **Script Properties** dialog box and give it a default value of "AKP_NULL". If you want to define actions yourself, you must do it in this dialog box (in which case the Operator Console program will over-ride "AKP_NULL" as the value of `AKPID` when a job is created).
Chapter 3

Knowledge Script basics

This chapter covers the basics of creating a Knowledge Script, with the exception of writing the code—that will be discussed in detail in subsequent chapters.

Even if you are interested only in modifying or extending pre-existing scripts—in such cases you will find much of this work already done for you—this chapter will provide you with a basis for understanding how to use the Developer’s Console to work with Knowledge Scripts.

The following topics are covered.

- Script elements
- Starting creation of a new script
- Setting default properties
- Where to go from here

Script elements

Apart from the code itself, there are a number of things that go into Knowledge Scripts. For example, you must name your script and assign an object type to it in ways that are consistent with the AppManager application framework. You must also choose values for a number of the non-code XML elements of the script.

Naming scripts

Each Knowledge Script name is composed of two parts: a prefix that determines its Knowledge Script category and a name (as self-explanatory as possible) that will be displayed in the Knowledge
Developing Custom Knowledge Scripts

Script pane of the AppManager Operator Console. An underscore character separates the two parts.

For example, if you name a Knowledge Script \texttt{NT\_DiskSpace}, the Knowledge Script is displayed in the Knowledge Script pane under the \texttt{NT} tab along with other NT-related Knowledge Scripts.

You can use an existing Knowledge Script category, or you create a new one. If you check in a Knowledge Script with a prefix that does not correspond to an existing category, a new category will be created. For example, if you create and check in a Knowledge Script named \texttt{Samples\_HelloWorld.qml}, and there is no \texttt{Samples} category, it will be created with your script in it.

Assigning an object type

Each Knowledge Script is associated with one or more resource object types. An object type is used to determine which resource objects—such as computers, disk drives, databases, or network cards to which
the Knowledge Script can be applied. Internally, AppManager uses a type checking mechanism to ensure that each Knowledge Script is applied only to the resource objects it can manage.

Creating new resource object types, and the discovery scripts that must go with them, is a complex activity that should not be undertaken by anyone who is not an experienced programmer. It is not covered in this book.

You should be able to find an existing resource object type that you can use. For example, \texttt{NT\_MachineFolder} is quite versatile for Windows. Similarly, for UNIX, \texttt{UNIX\_MachineFolder} is widely applicable.

To see what object types already exist, do the following:

1. Choose \textbf{Properties} from the \textbf{View} menu in the Developer’s Console.

2. Choose the \textbf{Object Types} tab in the \textbf{Script Properties} dialog box.

3. Click \textbf{Add} to open the \textbf{Add New Object Type} dialog box.

   Between the \textbf{Object group:} and \textbf{Objects:} lists, you can see all the existing object types.

This book is oriented towards script developers who are primarily interested in modifying or extending \textit{existing} Knowledge Scripts. In such cases, the object type will already be defined and you do not have to worry about it.

Object types are associated with icons. If you look in the \textbf{Knowledge Script} pane of the Operator Console, you will see that each script exhibits its own icon— the object type for the script. At the same time, each object in the \textbf{TreeView} pane is also represented by the icon for its object type. In general (a few icons are used for more than one object type), when the script icon matches the \textbf{TreeView} object icon, you can drop the script on that object.
Deciding on user-definable Script Parameters

You do not need to decide on user-definable Script Parameters before you write your script, although you can do so. For example, DO_DATA and DO_EVENT are very commonly used to allow users to decide whether to collect data or raise events.

You should always define a Script Parameter named AKPID, used by event messages to call for execution of action scripts. In most scripts, you will leave it to the user to determine actions so you set AKPID to a default of “AKP_NULL” (no actions). Users do not see AKPID as a Script Parameter, even though that’s where you defined it—they use the Actions tab of the Operator Console Properties dialog box to choose the actions. The AppManager program alters the AKPID Script Parameter in accordance with the user’s choices.
Other non-code XML elements

You will need to choose the scripting language before you begin writing code. You must also choose the type of Knowledge Script. There are four possibilities:

- Normal scripts, which can be either monitoring scripts (see Chapters 4 through 6 for more information) or report scripts (Chapter 10).
- Action scripts (see Chapters 7 through 9).
- Discovery scripts (not covered in this book).
- Install scripts (not covered in this book).

Starting creation of a new script

When you open the Developer's Console and choose New from the File menu, this is what you will see in the default Edit View:
You can begin to write your code immediately. Note, in the title bar of the Console window, that the default language is VBScript.

It appears that the new Knowledge Script is empty except for the two lines opening and closing the Main subroutine, where you will place your code. However, it is not empty—the non-code XML elements are there, even though they are not visible in the Edit View. Some of the non-code XML elements already have default values filled in, while others are empty, waiting for you to populate them through the Script Properties dialog box.

Choose XML (Read-only) from the View menu to see the entirety of the new Knowledge Script, including the header. If you have not yet opened the Script Properties dialog box and made any changes, this is what you will see:

The 6.0 agent can handle action scripts up to 256k bytes in length. Agents from previous versions of AppManager can only run action scripts up to 32k in length.
### Listing of the new (empty) script

The table below shows the entire contents of the window above, with explanations. You will populate the non-code XML elements (all but the last three rows in the table) yourself through the **Script Properties** dialog box.

<table>
<thead>
<tr>
<th>Script section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;PROLOGUE&gt;</code></td>
<td>You can enter comments here, such as the author’s name, the date, copyright statements, and a brief description of the script.</td>
</tr>
</tbody>
</table>

Here you edit the `.qml` file directly in a text editor, rather than through the **Script Properties** dialog box. This is the only place you should edit a non-code XML element without using the **Script Properties** dialog box.

| `<KSID>` | This section is already populated with some default values. The `<Type>` element is set to the default of “Regular”, which means that the script is of “Normal” type, as opposed to “Discovery” or “Action” or “Install”. `<AppManID>` is 4.0, meaning that this script is consistent with AppManager 4.0 and later. The `<KSVerID>` element is incremented automatically every time you check a script out and then back in. It begins at 1.0 and is incremented to 1.1, 1.2, ..... **Note**: To see the version number of any Knowledge Script selected in the **TreeView** pane of the Operator Console, open the right-click menu and choose **Version History**. A zero value for elements means “no” or “not required.” |

| `<ObjType v3style="1" fullpath="0" dropfolderlist="0">` | This element will contain the “resource object type” or types, when you add them with the **Script Properties** dialog box. |

---

---
Developing Custom Knowledge Scripts

**Setting default properties**

You use the Script Properties dialog box to set default values for the job properties. The properties for a running script will be set by whoever creates a job with this Knowledge Script using the Operator Console. They will use a different dialog box to do this.

The different tabs of the Script Properties dialog box are shown below. The Help file for the Developer's Console describes their use in detail. They are shown here to get you thinking about what goes into the non-code XML elements of your Knowledge Scripts. See the Appendix, “Dialog Boxes,” for a more detailed discussion.
The Header tab

You use this tab to set general values such as a tool-tip description of the Knowledge Script, its type (Normal, Action, Discovery, Install), the operating system, the scripting language, and the AppManager version.

Refer to the Help file for the Developer’s Console (choose Contents from the Help menu) for further information on the fields in the Script Properties dialog box.
The Object Types tab

In this tab you choose the resource object types for this script.

To add an object type, click the Add button. This will open the Add New Object Type dialog box.
The Default Schedule tab

Every Knowledge Script job must run according to a set schedule. Use this tab to set the default schedule. If someone creates a job with this Knowledge Script and does not choose a different schedule, this default schedule will be used.

The Advanced tab allows you to place restrictions on the allowed schedules.
Developing Custom Knowledge Scripts

The Action tab

![Image of the Action tab dialog box]

There are no actions at the outset. You add them with the **Add New Action** dialog box that opens when you click the **New** button.

Users can add actions themselves when they define the properties of a Knowledge Script job, with a nearly identical dialog box. Usually, you will leave it to users to define the actions, if any.
The Parameters tab

You select variables in your code whose values can be modified by an Operator Console user to change the behavior of the script. These variables are called “Script Parameters.” You write your code as if these variables are already defined. However, you do not explicitly add them to your code—AppManager will do that for you when it generates the final script.

Examples of Script Parameters are user-selected thresholds and limits. You can also use Script Parameters to specify behavior in your script. For example, you can add a Script Parameter called DO_EVENT that can have the values y or n. If the user sets the value to y (yes), then your script will raise events.

You use the Parameters tab of the Script Properties dialog box to:

- Create a Script Parameter.
- Assign a variable name to the Script Parameter so that your code may use the user-defined value and act accordingly.
- Assign a description to the Script Parameter that the Operator Console user will see.
- Assign default values for the Script Parameters.
- Define the allowed values for the Script Parameter. For example, you can allow a user to give `DO_EVENT` the values `y` or `n`, but no other values.

![Script Properties dialog box with parameters list]

The variables assigned to the Script Parameters that you define with the Parameters tab will become constants (Summit BasicScript) or variables (VBScript and Perl) in your script.
In a new script there are no Script Parameters at all. You use the **Add New Parameters** dialog box to add Script Parameters one at a time.

Each Script Parameter has two names:

- The name of the constant (Summit BasicScript) or variable (VBScript and Perl) that you use in your code. Enter this in the **Variable to use:** field. The Operator Console user will not see this name.

- The name (really more of a description) that is visible to the user and that the user can set a value for. Enter this in the **Description:** field.

The script developer should always define a parameter with a variable name of **AKPID** (for VBScript and Summit BasicScript) or **$Akpid** (for Perl). The Operator Console user will never see this parameter.

**Example of defining a Script Parameter**

Assume that you have decided to use a variable called **CPU_THRESHOLD**, nominally set to **50**, in your script, and also assume that you want the
AppManager Operator Console user to have the ability to change the value of this variable if they want to.

If this constant were not user-definable, you would just define it in your code like this:

- **CPU_THRESHOLD** = "50" (VBScript variable)
- **Const CPU_THRESHOLD = "50"** (Summit BasicScript constant)
- **$CPU_THRESHOLD = 50;** (Perl variable)

Since **CPU_THRESHOLD** is going to be user-definable in your code, you do not define it at all—you leave the definition to AppManager—but you write your code as if you had defined it. Here is an example of the process for a Knowledge Script called **Samples_Test** written in VBScript.

**Step 1, What the script developer does:** In the Developer's Console, you open the **Script Properties** dialog box, select the **Parameters** tab, click **Add**, and add your variable:
Returning to the **Parameters** tab, this is what you will see:

![Script Properties](image)

**Step 2, What the Operator Console user does:** When the Operator Console user drags the script to a target object (in this case a CPU), the **Properties for Samples_Test** dialog box opens and the user selects the **Values** tab.

**Note** The **Properties for Samples_Test** dialog box that the user sees in the AppManager Operator Console is similar to the **Script Properties** dialog box used by the script developer in the Developer's Console, but the two are *not* the same.
The user does not need to change the default value of 50, but in this case it is changed to 75.
After making this change in the value for CPU usage not-to-exceed (%), the user clicks OK in the Properties for Samples_Test dialog box and a job is started.

**Step 3, What the AppManager infrastructure does:** In the process of starting the job, AppManager adds a definition for CPU_THRESHOLD to the beginning of the samples_Test script. The running script now begins like this:

![Script for Job ID = 146](image)

In summary:

1. The script developer used a variable called CPU_THRESHOLD in the script and used the Developer's Console Script Properties dialog box to:
   - Create a Script Parameter.
   - Assign the variable named CPU_THRESHOLD to the Script Parameter.
   - Assign a description of “CPU usage not-to-exceed (%).” to the Script Parameter for the Operator Console user to see.
   - Assign a default value of 50 the Script Parameter.
2. The Operator Console user created a job from the script after changing the value of CPU usage not-to-exceed (%) to 75.

3. AppManager started the job after adding `CPU_THRESHOLD="75"` at the beginning of the script.

Where to go from here

This and prior chapters have provided an overview of the entire Knowledge Script. Beginning with the next chapter, the book will concentrate on the code portion of scripts. Examples will be given of scripts written in Summit BasicScript, VBScript, and Perl. If you plan to modify existing scripts, you will need to work in the scripting language in which the file was originally written. For the Windows environment, this could be either Summit BasicScript or VBScript. Therefore, if you will be developing scripts for Windows, you should study the chapters on both Summit BasicScript and VBScript.

Check in the sample scripts

All sample scripts used in this book can be found in your AppManager installation, or on your AppManager CD, in `documentation\development_tools\developer_guide\scripts`. It would be a good idea at this point to copy these files to a directory of your own choice and then check them into the AppManager repository (see “Editing Knowledge Scripts in the Developer’s Console,” on page 31.)

Note A Knowledge Script must be checked into the AppManager repository if it is to be visible in the Knowledge Script pane of the AppManager Operator Console. This does not mean that you will find it in your AppManager installation in the `NetIQ\AppManager\qdb\kp` directory. The files in this directory reflect only the files that were present when you first installed AppManager. Any new scripts, or alterations to existing scripts, that have been checked in will not be
copied to \NetIQ\AppManager\qdb\kp, unless you put them there yourself.

**Which scripting language to use**

For scripts that are to be run by an AppManager UNIX agent, you have only one choice—Perl.

For scripts that are to be run by an AppManager Windows agent, you may use either Summit BasicScript or VBScript. The latter is recommended for new scripts, except in these situations:

- There are some managed objects (e.g., Active Directory) that cannot be called from VBScript because they require type declarations that are not available. For example, VBScript supports the Variant data type, but not the String data type.

  Such managed objects are being rewritten so that they use the Variant data type, but the process is not yet complete. You can determine which managed objects have this type problem in VBScript by writing a short script and using the debugger.

- Not all managed objects are “thread safe.” If an AppManager agent is simultaneously running both BasicScript scripts and VBScript scripts that call the same managed objects, the different scripts can corrupt each other’s data. This is discussed in detail in the *Managed Objects Reference Guide*.

If you will be modifying scripts in the UNIX environment, you should read the chapters on Perl, but not necessarily those on Summit BasicScript and VBScript.

Report scripts are always written in VBScript because they are run on a Windows computer, irrespective of whether the data they report on comes from Windows or UNIX computers.
Chapter 4

Modifying a monitoring script written in VBScript

This chapter dissects the code in a sample Knowledge Script called Samples_FilesOpen.qml. This script is then modified to become Samples_FilesOpenEx.qml. As this is your first introduction to the code portion of scripts, the code is relatively simple.

You should open each sample Knowledge Script in your Developer’s Console where you can look at it in the various views and open its Script Properties dialog box.

You will also benefit from running the scripts in the AppManager Operator Console and experimenting with various Properties choices.

The following topics are covered in this chapter:

- Listing of the Samples_FilesOpen.qml script
- Preliminary discussion
- Syntax of the managed object methods
- Syntax of the Callback functions
- The program logic
- The modified script, Samples_FilesOpenEx.qml
- Performance Monitor counters

Listing of the Samples_FilesOpen.qml script

This sample Knowledge Script, Samples_FilesOpen.qml, is a complete script. You can check it in and run it as a job.
Samples_FilesOpen.qml checks for the number of files currently opened in the server, an indication of server activity. The script compares the result to the user-defined threshold. If the threshold is exceeded, the script generates an event and initiates any actions defined by the user.

After analyzing this script, you will learn how to modify it to return different information.

See “The modified script, Samples_FilesOpenEx.qml” on page 86.

Here is a listing of Samples_FilesOpen.qml running as a job. The sections at the beginning that are added by AppManager are included. Note that the Script Parameters are declared as variables in VBScript.

```vb
'### Begin KP-Version Section
Const AppManID = "4.5.78.0.8"
Const KSVerID = "1.0"
'### End KP-Version Section

'### Begin Type Section
NT_MachineFolder = "$JCRISSERT01"
'### End Type Section

'### Begin KPV Section
Sub KS_INIT ()
End Sub

'### End KPV Section

'### Begin KPP Section
DO_EVENT=y
DO_DATA=n
TH_FILES=10
Severity=5
AKPID="AKP_NULL"
'### End KPP Section

Dim NT
Dim SYSTEM
Const UNITNUMBER = "^^#"
Const ErrorSeverity = 35

Sub Main()
Dim dblValue
Dim strProgID
```
If NQEXT.IterationCount() = 1 Then
    strProgID = NQEXT.GetProgID ("NetiQAgent.NT", AppManID)
    Set NT = CreateObject (strProgID)
    Set SYSTEM = NT.System
    End If

    ' Retrieve the counter value for the Server/Files Open counter
    dblValue = SYSTEM.CounterValue("Server", "Files Open", "")
    If dblValue = -1 Then
        NQEXT.CreateEvent ErrorSeverity, _
        "Failed to retrieve the counter for Server/Files Open.", "AKP_NULL", "", 0, "", 0, 0
        Exit Sub
    End If
    End If

    ' Check threshold and raise an event if the threshold is exceeded
    If DO_EVENT = "y" Then
        Dim strDetailMsg
        If dblValue > TH_FILES Then
            strDetailMsg = "# of files open is " & dblValue & _
            strDetailMsg = "\" & dblValue & _
            NQEXT.CreateEvent Severity, "High number of files opened.", AKPID, "", 0, _
            strDetailMsg, "", 0, 0
        End If
    End If

    ' Collect data
    If DO_DATA = "y" Then
        NQEXT.CreateData 1, "Files Opened" & UNITNUMBER, 
        dblValue, "# of files open = " & _
        dblValue, 0
    End If
End Sub
Preliminary discussion

Recall from Chapter 2 the steps that the script undergoes when it is run:

1. A user chooses a script and drags it to the target object.
2. The Properties dialog box opens.
3. The user sets Script Parameters, the schedule, actions, etc.—or accepts the defaults—and closes the dialog box.
4. The Operator Console creates a job (an instance of the script along with the user configured Script Parameters, schedule, actions, etc.) in the AppManager repository.
5. The AppManager management server retrieves the job, the schedule, any action scripts, and so forth from the AppManager repository and forwards it all to the AppManager agent which will run the job. The final script has all Script Parameters and object types defined as constants with assigned values.

User-set Script Parameters

There are four Script Parameters that the user can alter when launching this script. These Script Parameters will become variables in the running script. The Script Parameters are:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO_DATA</td>
<td>Collect data? (y/n)</td>
<td>If “y”, data will be collected.</td>
</tr>
<tr>
<td>DO_EVENT</td>
<td>event? (y/n)</td>
<td>If “y”, an event will be raised when threshold is exceeded.</td>
</tr>
<tr>
<td>TH_FILES</td>
<td>Number of files open maximum threshold</td>
<td>Threshold for maximum number of files open.</td>
</tr>
</tbody>
</table>
Object types

The object type for this script is:

```
<Type name="NT_MachineFolder"></Type>
```

When the script is dragged onto the target object the Operator Console will assign the appropriate value. AppManager will assign the machine name of the target computer to the variable `NT_MachineFolder` and will insert it in the code like this:

```
NT_MachineFolder = "SJCRISSERT01"
```

Here, `SJCRISSERT01` is the machine name of the target computer.

Actions

`AKPID` determines what action scripts, if any, are run. If there are to be action scripts, they will be run when an event is raised—`AKPID` is a parameter of the Callback function `CreateEvent()`. If no events are raised, no action scripts will be run.

**Note** Raising events is the mechanism used to launch action scripts. Other than calling an event with `AKPID` as a (required) parameter, you do not write code to run action scripts.
The default for AKPID in this script is “AKP_NULL” (no action), which is the default for AKPID in most scripts. If the user adds actions with the Properties dialog box when setting up the job, the value of AKPID will be changed to “1, 2, 3, 4, .... n” when the user adds n actions (n >= 1).

Functions called in the code

The code calls two types of functions:

- Callback functions, by which the script requests information or action from the AppManager agent running the job. See Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript.”
- NetIQ managed object methods. Managed objects are COM objects whose methods are used to get basic information about hardware, applications, processes, and services on the managed computer. See the Managed Object Reference Guide.

Here are the functions called in the script, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQEXT.IterationCount</td>
<td>Callback function that determines the number of times that the calling Knowledge Script has run since it was last started or restarted.</td>
</tr>
<tr>
<td>NQEXT.GetProgID</td>
<td>Callback function that retrieves the versioned Prog ID of the NetIQagent.NT COM object that is required by this Knowledge Script.</td>
</tr>
<tr>
<td>SYSTEM.CounterValue()</td>
<td>Method call to retrieve a Windows Performance Monitor counter.</td>
</tr>
<tr>
<td>NQEXT.CreateEvent</td>
<td>Callback function that raises an event.</td>
</tr>
<tr>
<td>NQEXT.CreateData</td>
<td>Callback function that sends data points back for logging and graphing.</td>
</tr>
</tbody>
</table>
Syntax of the managed object methods

Refer to the *Managed Objects Reference Guide* for more details.

**System.CounterValue**

The *CounterValue* function returns the current value of a specified Performance Monitor counter and (if applicable) instance.

**Syntax**

```
System.CounterValue ObjectName, CounterName, InstanceName
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectName</td>
<td>String</td>
<td>Object name as it appears in the Performance Monitor <em>Add Counters</em> dialog box.</td>
</tr>
<tr>
<td>CounterName</td>
<td>String</td>
<td>Counter name as it appears in the Performance Monitor <em>Add Counters</em> dialog box.</td>
</tr>
<tr>
<td>InstanceName</td>
<td>String</td>
<td>Instance name as it appears in the Performance Monitor <em>Add Counters</em> dialog box. Use &quot;&quot; if the counter does not require an instance name.</td>
</tr>
</tbody>
</table>

*CounterValue* returns a double that is the current value of the counter specified by the input parameters. A return value of -1 indicates an error condition.

**Syntax of the Callback functions**

Refer to *Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript,”* for more details.

**Long IterationCount**

Returns the current iteration count.

**Syntax**

```
IterationCount
```

```
Developing Custom Knowledge Scripts

GetProgID

Retrieves version information for the managed object installed on the computer where the Knowledge Script is running. This is used to ensure that a particular version of a Knowledge Script calls a suitable version of a managed object.

Syntax

GetProgID progid, scriptver

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>progid</td>
<td>String</td>
<td>Version independent MO COM progid</td>
</tr>
<tr>
<td>scriptver</td>
<td>String</td>
<td>The associated KS script version string</td>
</tr>
</tbody>
</table>

GetProgID returns the Prog ID as a string.

CreateEvent

Used by a Knowledge Script to send an event to the AppManager agent. The AppManager agent will apply additional rule processing and will determine whether to send a new event or a duplicated (collapsed) event to the AppManager management server.

Syntax

CreateEvent sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [, deletefile]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>The event severity. A value from 1 to 40.</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>The message to be displayed under the Message column in the Events tab.</td>
</tr>
<tr>
<td>akp</td>
<td>String</td>
<td>Name of the action script to launch as a response to this event. You would normally create an AKPID parameter as part of your script. When the job is dropped and you select an action, the UI will fill in the AKPID variable with the action name. You will just need to pass in the AKPID variable to the script.</td>
</tr>
</tbody>
</table>
CreateEvent returns nothing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised. This value will determine which object in the <strong>TreeView</strong> pane to blink. Format of the value passed in should be &quot;ObjectName = Objectvalue&quot;, e.g. &quot;UNIX_DiskObject = /mnt/cdrom&quot;. The Objectvalue can normally be obtained by the drop object variable, e.g. UNIX_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value to raise the event. This parameter is currently not used. Set to 0.0.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the detail message or a file name that contains the detail message. The detailed message is displayed in the Message tab of the Event Property dialog box. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Not used. Should always be empty.</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Not used. Should always be 0.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg parameter is a file name or the detailed message itself. If it is a file name, then the contents of the file are read and passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype (!= 1). Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
Developing Custom Knowledge Scripts

CreateData

Sends data points for dynamic data streams. This function allows you to collect data for data streams that may be instantiated at each iteration.

**Syntax**

```plaintext
CreateData streamId, legend, dynaleg, objlist, val, agentmsg, msgtype [,schema] [,loglimit] [,lowWM] [,hiWM] [,deletefile]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>String</td>
<td>The data stream ID. For each unique stream ID in a script, it will generate a Data Source in the AppManager database. Subsequent calls to <code>CreateData</code> using the same stream ID will insert data points to the same Data Source.</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>The data stream legend. This value will show up under the Legend column and in the graphs. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>dynaleg</td>
<td>String</td>
<td>The data stream dynamic legend. Contains the dynamic information that can be used for reporting. If a portion of your legend changes often, then pass that text into this parameter. Otherwise leave it blank.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Corresponding object name where the data is collected on. This value is used for graphing and reporting. Format of the value passed in should be &quot;ObjectTypeName = ObjectValue&quot;, e.g. &quot;NT_DiskObject = D:&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. <code>NT_MachineFolder</code>.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The data point value.</td>
</tr>
</tbody>
</table>
CreateData returns nothing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the data detail or a file name that contains the data detail. The data detail is basically an annotation of each data point, giving more information about the data point since the data point is just a numeric value. For example, the data point value may be 5 for the number of processes running, while the data detail may list the processes that are running. The detailed message is displayed in the Graph Data Detail dialog box for each data point. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg is a file name or the detailed message itself. If it is a file name, then the contents of the file are passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Optional. XML schema for dynamic table creation in RDB. Default is an empty string.</td>
</tr>
<tr>
<td>loglimit</td>
<td>Long</td>
<td>Optional. The number of days to keep this data point in the database. Default is -1, keep forever. The data points can be removed from the database by other means.</td>
</tr>
<tr>
<td>lowWM</td>
<td>Double</td>
<td>Optional. Low watermark. Default is -1.0.</td>
</tr>
<tr>
<td>hiWM</td>
<td>Double</td>
<td>Optional. High watermark. Default is -1.0.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype ≠ 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
The program logic

`Samples_FilesOpen.qml` raises an event whenever the number of files that are open exceeds a threshold, `TH_FILES`, set by the user. The default is `TH_FILES = 200`.

The Windows NT/2000 managed object method `SYSTEM.CounterValue()` is used to obtain the number of open files.

**Sub Main()**

`IterationCount()` returns the number of times that the Knowledge Script job has run. If this is the first time that the script has been run, `IterationCount()` will return 1. In that case, the body of the `If NQEXT.IterationCount() = 1 Then` block will be executed in order to:

- Obtain the ID of the COM Object that contains the managed object method, `SYSTEM.CounterValue()`, that will be used in the Knowledge Script.
- Create the `NT.SYSTEM` object so that this method can be called.

```
If NQEXT.IterationCount() = 1 Then
    strProgID = NQEXT.GetProgID("NetiQAgent.NT", AppManID)
    Set NT = CreateObject(strProgID)
    Set SYSTEM = NT.System
End If
```

The Callback function `NQEXT.GetProgID()` constructs the COM object ID from the string “`NetiQAgent.NT`” and the value `AppManID`. `AppManID` is the AppManager build number that is appropriate for this Knowledge Script. It is defined in the non-code XML section of the Knowledge Script, and appears in the header section of the final, generated script:

```
'### Begin KSID Section
Const AppManID = "4.0.15.1"
Const KSVerID = "1.0"
'### End KSID Section.
```
Note The COM ID will be of the form NetIQAgent.NT.n, where n is an integer. In general, n will not be equal to the actual value of AppManID. For example, in this script AppManID=4.0.15.1, while the ID of the COM object for AppManager 5 is NetIQAgent.NT.4. The Callback function NQEXT.GetProgID() translates the AppManID into the proper value for the COM ID.

The next section of the code calls SYSTEM.CounterValue() to get the number of open files. When the ObjectName parameter is set to “Server” and the CounterName parameter is set to “Files Open,” SYSTEM.CounterValue returns the number of files that are currently open.

If the call to SYSTEM.CounterValue fails, it will return -1. In this case, the If dblValue = -1 Then block is executed. An event is raised indicating failure to obtain the counter value and the code exits.

' Retrieve the counter value for the Server/Files Open counter
    dblValue = SYSTEM.CounterValue("Server", "Files Open", ")
    If dblValue = -1 Then
        NQEXT.CreateEvent ErrorSeverity, 
            "Failed to retrieve the counter for Server/Files Open.", "AKP_NULL", ",", 0, ",", ",", 0, 0
        Exit Sub
    End If

If the call to SYSTEM.CounterValue( ) succeeds, then an event will be raised only if DO_EVENT = “Y” and the TH_FILES threshold is exceeded.

' Check threshold and raise an event if the threshold is exceeded
    If DO_EVENT = "y" Then
        Dim strDetailMsg
        If dblValue > TH_FILES Then
            strDetailMsg = "# of files open is " & dblValue & "; \text{>TH = "} & TH_FILES
            NQEXT.CreateEvent Severity, "High number of files opened.", AKPID, "", 0, ",", ",", 0, 0
            strDetailMsg, "", 0, 0
        End If
    End If
In this case, the event message will report the number of open files and the threshold that was exceeded under the **Message** heading in the **Event** pane of the Operator Console. If a user double-clicks on the event in the **Event** pane to open the **Event Properties** dialog box, and then chooses the **Message** tab, they will see the **strDetailMsg**.

The **CreateEvent()** parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>Event severity.</td>
</tr>
<tr>
<td>&quot;High number of files opened.&quot;</td>
<td>Event message.</td>
</tr>
<tr>
<td>AKPID</td>
<td>Actions to execute, if any. Value of AKPID depends on whether the user requested actions. If not, it will have the default value of &quot;AKP_NULL&quot;, which means no action.</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Corresponding object name where the event is raised. An empty string is used here means to blink the NT_MachineFolder level.</td>
</tr>
<tr>
<td>0</td>
<td>The current value (to raise the event). Not used here.</td>
</tr>
<tr>
<td>strDetailMsg</td>
<td>The plain text message, defined in the line of code just before call to CreateEvent().</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Event source. Should always be empty.</td>
</tr>
<tr>
<td>0</td>
<td>Event ID. Should always be 0.</td>
</tr>
<tr>
<td>0</td>
<td>Indicates that strDetailMsg is plain text, as opposed to a file.</td>
</tr>
</tbody>
</table>

The last block of code in the script handles data collection, provided that the user has set **DO_DATA** to “y” (the default is “n”).

```'
' Collect data
    If DO_DATA = "y" Then
        NQEXT.CreateData 1, "Files Opened" & UNITNUMBER, "", "", 
        "", 
        dblValue, 
        # of files open = " & _ 
        dblValue, 0
    End If
```
The `CreateData()` parameters are:

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The ID of the data stream—becomes important when there is more than one stream.</td>
</tr>
<tr>
<td>&quot;Files Opened&quot; &amp; UNITNUMBER</td>
<td>Text for the <strong>Legend</strong> heading in the <strong>Graph Data</strong> tab of the <strong>List pane</strong> = &quot;Files Opened &quot; # UNITNUMBER is a defined constant: const UNITNUMBER = &quot;^^#&quot; where the caret represents spaces</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Dynamic legend, contains dynamic information that can be used for reporting. Not used here.</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Corresponding object name where the data is collected. Not used here.</td>
</tr>
<tr>
<td>dblValue</td>
<td>Current data point value.</td>
</tr>
<tr>
<td>&quot;# of files open = &quot; &amp; dblValue</td>
<td>This is a message created (agentmsg) by the developer that appears in the data detail dialog box.</td>
</tr>
<tr>
<td>0</td>
<td>0 means that agentmsg is plain text (as opposed to a file).</td>
</tr>
</tbody>
</table>
The modified script, Samples_FilesOpenEx.qml

Here is the listing for Samples_FilesOpenEx.qml, a modified version of Samples_FilesOpen.qml. The modification, shown in bold, is very simple. The Performance Monitor counter value returned is changed from “Server”, “Files Open” to “Server”, “Files Opened Total.”

**Note** There is no reason whatsoever why you could not modify this script to obtain both measures of served files. Go ahead and try it.

Microsoft defines “Files Opened Total” as “The number of successful open attempts performed by the server on behalf of clients (since the last reboot). Useful in determining the amount of file I/O, determining overhead for path-based operations, and for determining the effectiveness of open locks.”

By comparison, “Files Open” is defined as “The number of files currently opened in the server. Indicates current server activity.”

In other words, Samples_FilesOpen.qml checks for the number of files opened in the server that are *still open*. Samples_FilesOpenEx.qml checks for the number of files that have been opened in the server since your computer was last rebooted, *whether or not* they are still open.

Dim NT
Dim SYSTEM
Const UNITNUMBER = "^^#"
Const ErrorSeverity = 35

Sub Main()
    Dim dblValue
    Dim strProgID

    If NQEXT.IterationCount() = 1 Then
        strProgID = NQEXT.GetProgID ("NetiQAgent.NT", AppManID)
        Set NT = CreateObject (strProgID)
        Set SYSTEM = NT.System
    End If

    ' Retrieve the counter value for the Server/Files Open counter
dblValue = SYSTEM.CounterValue("Server", "Files Opened Total", "")

If dblValue = -1 Then
    NQEXT.CreateEvent ErrorSeverity, "Failed to retrieve the counter for Server/Files Open.", "AKP_NULL", "", 0, "", "", 0, 0
    Exit Sub
End If

' Check threshold and raise an event if the threshold is exceeded
If DO_EVENT = "y" Then
    Dim strDetailMsg
    If dblValue > TH_FILES Then
        strDetailMsg = "# of files open is " & dblValue & "; >TH = " & TH_FILES
        NQEXT.CreateEvent Severity, "High number of files opened.", AKPID, "", 0, 
            strDetailMsg, "", 0, 0
    End If
End If

' Collect data
If DO_DATA = "y" Then
    NQEXT.CreateData 1, "Files Opened" & UNITNUMBER, "", "", 
        dblValue, "# of files open = " & 
        dblValue, 0
End If
End Sub

Performance Monitor counters

The managed object method SYSTEM.CounterValue(), used in this script to obtain Performance Monitor data, is a very powerful general method. You can use it in your own scripts to obtain a wide range of performance metrics.

To get an idea of the possibilities, do the following:
1. Choose Programs > Administrative Tools > Performance from the Windows Start menu to open the Performance window. (Alternatively, you can run perfmon from the command line.)

2. When the Performance window opens, right-click in the right-hand pane and choose Add Counters... from the pop-up menu.

The Add Counters dialog will open.

The counters are identified by three things:

- The Performance Object, which you choose from the Performance Object drop-down list.
- The counter itself, which you choose from the Select counters from list text box.
- The Instance (lower right in the picture).

When you call SYSTEM.CounterValue() from your scripts, the Performance Object name is the first parameter and the counter name is the second parameter. The third parameter is the instance, if any.
3 When you have highlighted a counter, click the **Explain** button to learn about the counter. For example:
Chapter 5  

Modifying a monitoring script written in Summit BasicScript

This chapter dissects the code in a sample Knowledge Script called `Samples_CpuLoaded.qml`. This script is then modified to become `Samples_CpuLoadedEx.qml`.

You should open each sample Knowledge Script in your Developer's Console where you can look at it in the various views and open its Script Properties dialog box.

You will also benefit from running the scripts in the AppManager Operator Console and experimenting with various Properties choices.

The following topics are covered in this chapter:
- Listing of the NT_CpuLoaded.qml script
- Preliminary discussion
- Syntax of the managed object methods
- Syntax of the Callback functions
- The program logic
- The modified script, NT_CpuLoadedEx.qml

Listing of the NT_CpuLoaded.qml script

This sample Knowledge Script, `Samples_CpuLoaded.qml`, is the same as `NT_CpuLoaded.qml`, except for several minor changes and the addition of numerous comments. `Samples_CpuLoaded.qml` is a complete script. You can check it in and run it as a job.
Samples_CpuLoaded.qml checks the current values for CPU total processor time, CPU user time, and CPU queue length against the user-defined thresholds. If the thresholds are exceeded, the subroutine CpuCheck() generates events and initiates any actions defined by the user.

After analyzing this script, you will learn how to modify it to return more information. See “The modified script, NT_CpuLoadedEx.qml” on page 111.

Here is a listing of Samples_CpuLoaded.qml running as a job. The sections at the beginning that are added by AppManager are included. Note that the Script Parameters are declared as constants in Summit BasicScript.

```qlim
### Begin KP-Version Section
Const AppManID = "4.5.78.0.8"
Const KSVerID = "1.0"
### End KP-Version Section

### Begin Type Section
Const NT_CPUFolder = "CPU"
Const NT_CPUNumber = "0"
### End Type Section

### Begin KPV Section
Sub KS_INIT ()
End Sub

### End KPV Section

### Begin KPP Section
Const DO_EVENT="y"
Const DO_DATA="n"
Const DO_OVERALL="n"
Const TH_UTIL=90
Const TH_QLEN=2
Const Severity=5
Const PRM_KSERR=35
Const AKPID="AKP_NULL"
### End KPP Section

Dim NT As Object
Dim CPU As Object
Const UNITPERCENT = "%%"```
'This sub routine checks for the processor time, user time, 'and queue length to see if they exceed the given 'thresholds for a given cpu, or the overall cpu

Sub CpuCheck(sCPUName As String)
    Dim dUserTime#, dPrivilegeTime#, dTotalTime#, dQueueLen#
    Dim sDetailMsg$, sObjectList$, sCPUMsg$
    Dim lStreamID As Long
    If (sCPUName = "") Then
        ' Set the machine object as the resource. This will cause
        ' the machine object to blink if there is an event.
        sObjectList = "NT_CPUFolder = " + NT_CPUFolder
        lStreamID = 0
        sCPUMsg = "Overall CPU"
    Else
        ' Set the individual cpu name as the resource. This will
        ' cause the individual cpu object to blink if there is
        ' an event for each individual cpu
        sObjectList = "NT_CPUNumber = " + sCPUName
        lStreamID = Val(sCPUName)
        sCPUMsg = "CPU# " + sCPUName
    End If
    dTotalTime = CPU.UtilValue("PROCESSOR", sCPUName)
    dUserTime  = CPU.UtilValue("USER", sCPUName)
    If dTotalTime = -1 Or dUserTime = -1 Then
        ' A return value of -1 indicates a failure to
        ' retrieve the value of the counter
        MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
        sObjectList,"Processor or User counter not found_
        (Proc: " & Cstr(dTotalTime) & ", User: " & _
        Cstr(dUserTime) & ")"
        Exit Sub
    End If
    If dTotalTime > dUserTime Then
        dPrivilegeTime = dTotalTime - dUserTime
    Else
        dPrivilegeTime = 0
    End If
    If IterationCount() = 1 Then
        If DO_DATA = "y" Then
            DataHeader "PROCESSOR Utilization - " & sCPUMsg _
Developing Custom Knowledge Scripts

If DO_EVENT = "y" And dTotalTime > TH_UTIL Then
    dQueueLen = CPU.QueueLengthValue
    If dQueueLen = -1 Then
        MSActions PRM_KSERR, "Counter not found", "AKP_NULL",
        sObjectList, "The queue length counter could not be found"
        Exit Sub
    End If

    ' if TH_QLEN = -1 ignore query length value and raise event
    ' else if query length value exceeds threshold value then raise event
    If TH_QLEN = -1 Then
        sDetailMsg = sCPUMsg + " utilization% is " & _
        Format$(dTotalTime, "0.00") & " >TH = " & Cstr(TH_UTIL)
        MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
        sObjectList, sDetailMsg
    Elseif dQueueLen > TH_QLEN Then
        sDetailMsg = sCPUMsg + " utilization% is " & _
        Format$(dTotalTime, "0.00") & " >TH = " & _
        Cstr(THUTIL) & " AND" & chr$(10) & "CPU queue length is " & Cstr(dQueueLen) & " >TH = " & _
        Cstr(TH_QLEN)
        MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
        sObjectList, sDetailMsg
    End If
End If

If DO_DATA = "y" Then
    sDetailMsg = sCPUMsg + " utilization% is: " & chr$(10)
    & "Privileged " & Format$(dPrivilegeTime, "0.00") & _
    chr$(10) & "User " & Format$(dUserTime, "0.00") & _
    chr$(10) & "Total " & Format$(dTotalTime, "0.00")
    DataLog lStreamID, dTotalTime, sDetailMsg
End If
Sub Main()
    Dim sCPUName$, sProgID$
    Dim iNumberCPU As Integer

    If IterationCount() = 1 Then
        ' Retrieve the prog id of the NetIQ NT MO COM object
        sProgID = MCGetMOID ("NetiQAgent.NT", AppManID)
        Set NT = CreateObject (sProgID)
        Set CPU = NT.CPU
    End If

    iNumberCPU = ItemCount(NT_CPUNumber, ",")
    If iNumberCPU = 1 Or DO_OVERALL = "n" Then
        ' Check each individual CPU in the object list
        For I = 1 To iNumberCPU
            sCPUName = Item$(NT_CPUNumber, I,, ",")
            CpuCheck sCPUName
        Next I
    Else
        ' Just check the overall CPU usage
        CpuCheck ""
    End If
End Sub

Preliminary discussion

Recall from Chapter 2 the steps that the script undergoes when it is run:

1. A user chooses a script and drags it to the target object.
2. The Properties dialog box opens.
3. The user sets Script Parameters, the schedule, actions, etc.—or accepts the defaults—and closes the dialog box.
4. The Operator Console creates a job (an instance of the script along with the user configured Script Parameters, schedule, actions, etc.) in the AppManager repository.
Developing Custom Knowledge Scripts

The AppManager management server retrieves the job, the schedule, any action scripts, and so forth from the AppManager repository and forwards it all to the AppManager agent which will run the job. The final script has all Script Parameters and object types defined as constants with assigned values.

### User-set Script Parameters

There are seven Script Parameters that the user can alter when launching this script. These Script Parameters will become constants in the running script. The Script Parameters are:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO_DATA</td>
<td>Collect data? (y/n)</td>
<td>If = “y”, event will be fired when thresholds are exceeded.</td>
</tr>
<tr>
<td>DO_EVENT</td>
<td>Event? (y/n)</td>
<td>If = “y”, data will be collected.</td>
</tr>
<tr>
<td>DO_OVERALL</td>
<td>Overall load? (y/n)</td>
<td>If = “y”, script will be run only to obtain the % usage of all CPUs aggregated together. If = “n”, script will be run for each individual CPU.</td>
</tr>
<tr>
<td>TH_UTIL</td>
<td>%CPU maximum threshold</td>
<td>Threshold for maximum CPU % usage.</td>
</tr>
<tr>
<td>TH_QLEN</td>
<td>CPU queue length maximum threshold</td>
<td>Threshold for maximum CPU queue length.</td>
</tr>
<tr>
<td>Severity</td>
<td>Event severity level</td>
<td>Severity level of event fired (if DO_EVENT = “y”).</td>
</tr>
</tbody>
</table>
Object types

The object types for this script are:
<Type name="NT_CPUFolder"></Type>
<Type name="NT_CPUNumber"></Type>

When the script is dragged onto the target object, AppManager will assign the appropriate values:

- **NT_CPUFolder** will be assigned the name of the top-level folder.
- **NT_CPUNumber** will be assigned a comma-delimited string listing all the individual CPUs.

AppManager will insert these constants in the code of the final, generated script like this:

```vbnet
Const NT_CPUFolder = "CPU"
Const NT_CPUNumber = "0"
```

In this case, there is only one CPU in the CPU folder.
Actions

**Akpid** determines what action scripts, if any, are run. If there are to be action scripts, they will be run when an event is raised—**Akpid** is a parameter of the Callback function **CreateEvent()**. If no events are raised, no action scripts will be run.

**Note** Raising events is the mechanism used to launch action scripts. Other than calling an event with **Akpid** as a (required) parameter, you do not write code to run action scripts.

The default for **Akpid** in this script is "**AKP_NULL**" (no action), which is the default for **Akpid** in most scripts. If the user adds actions with the **Properties** dialog box when setting up the job, the value of **Akpid** will be changed to “1,2,3,4,...,n” when the user adds n actions (n >= 1).

Functions called in the code

The code calls three types of functions:

- Summit BasicScript built-in functions. See the BasicScript documentation in `appmanager\documentation\development_tools\summit_basicscript\documentation`.
- Callback functions, by which the script requests information or action from the AppManager agent running the job. See Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript.”
- NetIQ managed object methods. Managed objects are COM objects whose methods are used to get basic information about hardware, applications, processes, and services on the managed computer. See the Managed Object Reference Guide.
Here are the functions called in the script, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU.UtilValue</td>
<td>Windows NT managed object that obtains information about CPU utilization.</td>
</tr>
<tr>
<td>MSActions</td>
<td>Callback function that reports events and initiates actions.</td>
</tr>
<tr>
<td>IterationCount</td>
<td>Callback function that determines the number of times that the calling Knowledge Script has run.</td>
</tr>
<tr>
<td>DataHeader</td>
<td>Callback function that sends the data header for logging and graphing data streams.</td>
</tr>
<tr>
<td>CPU.QueueLengthValue</td>
<td>Windows NT managed object that obtains information about the CPU queue length.</td>
</tr>
<tr>
<td>CStr</td>
<td>Summit BasicScript built-in function that converts an expression to a string.</td>
</tr>
<tr>
<td>Chr$</td>
<td>Summit BasicScript built-in function that returns the character whose value is its argument. In this script, chr$(10) returns a carriage return.</td>
</tr>
<tr>
<td>Format$</td>
<td>Summit BasicScript built-in function that formats a string to a user’s specification.</td>
</tr>
<tr>
<td>DataLog</td>
<td>Callback function that sends points back for logging and graphing.</td>
</tr>
<tr>
<td>MCGetMOID</td>
<td>Callback function that retrieves the versioned ProgID of the COM object that is required by this Knowledge Script.</td>
</tr>
<tr>
<td>ItemCount</td>
<td>Summit BasicScript built-in function that returns the number of items in a delimited text string list.</td>
</tr>
<tr>
<td>Item$</td>
<td>Summit BasicScript built-in function that returns a discrete item in a delimited text string list.</td>
</tr>
</tbody>
</table>
**Syntax of the managed object methods**

Refer to the *Managed Objects Reference Guide* for more details.

**CPU.UtilValue**

This function reports the percentage of CPU utilization for the entire system. You specify the type of CPU utilization to return (total, privileged time, or user time). On a multiprocessor, the value returned is the average CPU utilization for all system processors.

**Syntax**

```plaintext
CPU.UtilValue what, CpuInstance
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>what</td>
<td>String</td>
<td>Type of CPU usage to monitor. Valid settings are (case sensitive):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PROCESSOR for utilization of both privileged and user CPU modes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PRIVILEGED for utilization in privileged mode by the NT operating system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• USER for utilization in user mode by the application.</td>
</tr>
<tr>
<td>CpuInstance</td>
<td>String</td>
<td>Processor number. For a single processor system, this number should be &quot;0&quot;. For the average of all processors, use an empty string (&quot;&quot;&quot;).</td>
</tr>
</tbody>
</table>

Returns a double representing the percentage of time that the specified processor(s) is busy. A return value of -1 indicates an error condition.

**CPU.QueueLengthValue**

**Syntax**

```plaintext
CPU.QueueLengthValue
```

This function has no parameters. It returns the length of the processor queue in number of threads, as a double representing the number of
process threads in the processor queue. A return value of -1 indicates an error condition.

**Syntax of the Callback functions**

Refer to Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript” for more details.

**MSActions**

Allows a Knowledge Script to report events and initiate actions.

**Syntax**

MSActions severity, shortmsg, akpid, objlist, detailmsg [,detailmsg2, ..... ,detailmsg6] [, value]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>Long</td>
<td>Severity of the event.</td>
</tr>
<tr>
<td>shortmsg</td>
<td>String</td>
<td>Event message displayed in the List pane.</td>
</tr>
<tr>
<td>akpid</td>
<td>String</td>
<td>Action name or identifier for the action to be taken.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Objects that report the event (their icons will be set to blinking in the Operator Console’s TreeView pane).</td>
</tr>
<tr>
<td>detailmsg</td>
<td>String</td>
<td>Detail message from the AppManager agent(s) displayed in the event’s Properties dialog. At least one detailmsg or an empty string is required. The maximum size of the string is 32K. To pass additional information beyond the 32K, you can specify up to 6 message strings, each with a maximum size of 32K, to define the entire detail message for an event. For example, if the message you want to return is 64K, the message would be stored in two strings: MSActions Severity, “High”, AKPID, “”, detailmsg, detailmsg2</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Optional. The current value to raise an event.</td>
</tr>
</tbody>
</table>
**IterationCount**

Returns a Long representing the current iteration count.

**Syntax**

`IterationCount`

This function has no parameters.

**DataHeader**

Sends the data header for logging and graphing data streams (short form).

**Syntax**

`DataHeader legend, graph_id, stream_id [, objlist]`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>legend</td>
<td>String</td>
<td>Graphing legend displayed in the List and Graph panes. For example, the legend for one data stream created by <code>nt_CpuResource</code> is User CPU. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>graph_id</td>
<td>Long</td>
<td>Graph ID. This parameter is not currently used. It is always set to the value 0.</td>
</tr>
<tr>
<td>stream_id</td>
<td>Long or String</td>
<td>Data stream identifier. This identifier should be unique for each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Optional. Matching object where the data is collected.</td>
</tr>
</tbody>
</table>
DataLog

Sends data points back for logging and graphing. This call is always used in conjunction with a DataHeader call.

**Syntax**

\[\text{DataLog } \text{stream\_id}, \text{data}, \text{datapointmsg}\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream_id</td>
<td>Long or String</td>
<td>Data stream identifier. This identifier should be the same identifier used in the associated dataheader call for each data stream.</td>
</tr>
<tr>
<td>data</td>
<td>Double</td>
<td>Data point value.</td>
</tr>
<tr>
<td>datapointmsg</td>
<td>String</td>
<td>Detail message displayed in the Graph Data Detail dialog. The maximum size for this string is 32K.</td>
</tr>
</tbody>
</table>

MCGetMOID

Retrieves version information for the managed object installed on the computer where the Knowledge Script is running. This is used to ensure that a particular version of a Knowledge Script calls a suitable version of a managed object.

**Syntax**

\[\text{MCGetMOID } (\text{programid}, \text{version})\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>programid</td>
<td>String</td>
<td>Managed object program identifier. For example, NetIQ_Agent.NT.</td>
</tr>
<tr>
<td>version</td>
<td>String</td>
<td>Knowledge Script version (for example, App_Man_ID or KS_Ver_ID parameter).</td>
</tr>
</tbody>
</table>

Returns a String representing the managed object version.
The program logic

The main work in this script is done by the CpuCheck() subroutine. This subroutine is called from Main() with an argument that depends on the value of DO_OVERALL and the number of CPUs. If there is only one CPU, or if DO_OVERALL = "n", CpuCheck is called for each CPU individually (argument = name of CPU). If, on the other hand, there are multiple CPUs and DO_OVERALL = "y", CpuCheck is called for all CPUs aggregated together (argument = "").

CpuCheck() checks the current values for CPU total processor time, CPU user time, and CPU queue length against the user-defined thresholds. If the thresholds are exceeded, CpuCheck() generates events (and may initiate actions if the user defined any).

CpuCheck() requires sCPUName as an input parameter. sCPUName is a string that is defined in Main(). Its value is either the name of a CPU or an empty string, the latter signifying that CpuCheck() should check only the sum of all CPUs.

Sub Main()

IterationCount() returns the number of times that the Knowledge Script job has run. If this is the first time that the script has been run, IterationCount() will return 1. In that case, the body of the

If IterationCount() = 1 Then block will be executed to:

- Obtain the ID of the COM Object that contains the managed objects that will be used in the Knowledge Script (CPU.UtilValue and CPU.QueueLengthValue).
- Create the NT.CPU object so that these two methods can be called.

If IterationCount() = 1 Then
    ' Retrieve the prog id of the NetIQ NT MO COM object
    sProgID = MCGetMOID("NetiQAgent.NT", AppManID)
    Set NT = CreateObject(sProgID)
    Set CPU = NT.CPU
End If
The Callback function `MCGetMOID()` constructs the COM object ID from the string “NetIQAgent.NT” and the parameter `AppManID`.

`AppManID` is the AppManager build number that is appropriate for this Knowledge Script. It is defined in the non-code XML section of the Knowledge Script, and appears in the header section of the final, generated script that the AppManager agent runs:

```vbnet
'### Begin KSID Section
Const AppManID = "4.0.15.1"
Const KSVerID = "1.0"
'### End KSID Section.
```

**Note** The COM ID will be of the form `NetIQAgent.NT.n`, where `n` is an integer. In general, `n` will not be equal to the actual value of `AppManID`. For example, in this script `AppManID`=4.0.15.1, while the ID of the COM object for AppManager 5 is `NetIQAgent.NT.4`. The Callback function `MCGetMOID()` translates the `AppManID` into the proper value for the COM ID.

The object type `NT_CPUNumber` contains a comma-delimited string listing all the individual CPUs (the names of all the CPU objects on which the script was dropped—determined by the Operator Console when the script is dropped). The Summit BasicScript function `ItemCount()` returns the number of items in the list, given that the comma (”,") is the delimiter, and assigns the value to `iNumberCPU`.

```vbnet
iNumberCPU = ItemCount(NT_CPUNumber, ",")
```

If there is only one CPU or if `DO_OVERALL` is set to “n”, the `For I = 1 To iNumberCPU` loop is executed. This loop uses the Summit BasicScript function `Item$()` to step through the individual CPUs, calling the `CpuCheck()` subroutine on each one in turn:

```vbnet
If iNumberCPU = 1 Or DO_OVERALL = "n" Then
' Check each individual CPU in the object list
For I = 1 To iNumberCPU
    sCPUName = Item$(NT_CPUNumber, I, ",")
    CpuCheck sCPUName
Next I
```
If there is more than one CPU and do_overall is set to “y”, CpuCheck() is called with an empty string to check only the sum of all CPUs:
Else
   ' Just check the overall CPU usage
   CpuCheck ""
End If

Sub CpuCheck()

The parameter passed to this subroutine is either the name of the individual CPU or “”. When an empty string is passed in, CpuCheck checks only the sum of all CPUs.

To fire an event, both the CPU total usage and the CPU queue length thresholds set by the user must be exceeded. An exception to this is that the CPU queue length is ignored if the user has set its threshold to -1.

The program will exit the CpuCheck() subroutine if any calls to the managed object methods CPU.UtilValue or CPU.QueueLengthValue fail (return -1).

The subroutine declares variables for the four quantities that will be checked (actually, only three will be checked—dPrivilegeTime will be calculated from dTotalTime and dUserTime):
   Dim dUserTime#, dPrivilegeTime#, dTotalTime#, dQueueLen#

Then, the subroutine creates:
• sObjectList, a string that is used to tell the Operator Console which object icon to blink when an event is raised.
• lstStreamID, an ID for tagging any data streams that are created. lstStreamID=0 for the sum of all CPUs or lstStreamID=n for individual CPU n.
• SCPUMsg for identifying the CPU# (or “OVERALL CPU”) when returning messages.

   If (scpuname = "") Then
' Set the machine object as the resource. This will
' cause the machine object to blink
' if there is an event.
sObjectList = "NT_CPUFolder = " + NT_CPUFolder
lStreamID = 0
sCPUMsg = "Overall CPU"
Else
' Set the individual cpu name as the resource.
' This will cause the individual cpu object
' to blink if there is an event for each individual cpu.
sObjectList = "NT_CPUNumber = " + sCPUName
lStreamID = Val(sCPUName)
sCPUMsg = "CPU# " + sCPUName
End If

Next, CpuCheck() calls the NT.CPU.UtilValue() managed object to get the “total CPU time” for the processor or processors identified by sCPUName:

dTotalTime = CPU.UtilValue("PROCESSOR", sCPUName)

Then, CpuCheck() calls the NT.CPU.UtilValue() managed object to get the “user CPU time” for the processor or processors identified by sCPUName:

dUserTime = CPU.UtilValue("USER", sCPUName)

If NT.CPU.UtilValue() fails, it returns -1. Here, if either call to NT.CPU.UtilValue() fails, the Callback function MSActions returns an event with an error message and blinks the correct object icon. Then the subroutine exits:

If dTotalTime = -1 Or dUserTime = -1 Then
' A return value of -1 indicates a failure to
' retrieve the value of the counter
MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
sObjectList,"Processor or user counter not found _
Proc: " & Cstr(dTotalTime) & ", User: " & _
Cstr(dUserTime) & ")"
Exit Sub
End If
If both NT.CPU_UtilValue() calls succeed, dPrivilegeTime is calculated from dTotalTime and dUserTime:

If dTotalTime > dUserTime Then
    dPrivilegeTime = dTotalTime - dUserTime
Else
    dPrivilegeTime = 0
End If

Next, the Callback function IterationCount() returns the number of times the Knowledge Script job has been run, including the current job. If this is the first time, and if the script is to collect data (DO_DATA = "y"), then a heading for the data to be collected is created with the Callback function DataHeader:

If IterationCount() = 1 Then
    If DO_DATA = "y" Then
        DataHeader "PROCESSOR Utilization - " & sCPUMsg 
        & UNITPERCENT, 0, lStreamID
    End if
End If

Next, the If DO_EVENT = "y" And dTotalTime > TH_UTIL Then block is executed only if both these conditions are true:

● the CPU threshold is exceeded (dTotalTime > TH_UTIL), and
● events are to be sent (DO_EVENT = "y").

If DO_EVENT = "y" And dTotalTime > TH_UTIL Then
    dQueueLen = CPU.QueueLengthValue
    If dQueueLen = -1 Then
        MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
        sObjectList, "The queue length counter could not be found"
        Exit Sub
    End If

' if TH_QLEN = -1 ignore query length value and raise event
' else if query length value exceeds threshold value then raise event
If TH_QLEN = -1 Then
    sDetailMsg = sCPUMsg + " utilization% is " & _
    Format$(dTotalTime, "0.00") & _
"; >TH = " & Cstr(TH_UTIL)
    MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
    sObjectList, sDetailMsg
Elseif dQueueLen > TH_QLEN Then
    sDetailMsg = sCPUMsg + " utilization% is " & _
    Format$(dTotalTime, "0.00") & "; >TH = " & _
    Cstr(TH_UTIL) & " AND" & chr$(10) & "CPU queue _
    length is " & Cstr(dQueueLen) & "; >TH = " & _
    Cstr(TH_QLEN)
    MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
    sObjectList, sDetailMsg
End If
End If

In the block of code above, the managed object
NT.CPU.QueueLengthValue retrieves the CPU queue length and
assigns it to dQueueLength, provided that the user is interested in the
queue length (TH_QLEN <> -1):

dQueueLen = CPU.QueueLengthValue

If QueueLengthValue fails (returns -1) the Callback function
MSActions returns an event with an error message and blinks the
correct object icon. Then the program exits the CpuCheck() subroutine:

If dQueueLen = -1 Then
    MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
    sObjectList, "The queue length counter could _
    not be found"
    Exit Sub
End If

If the user is uninterested in dQueueLength (TH_QLEN = -1), then the
Callback function MSActions is used to raise an event with a message
that the CPU threshold has been exceeded:
If TH_QLEN = -1 Then
    sDetailMsg = sCPUMsg + " utilization% is " & _
    Format$(dTotalTime, "0.00") & _
    "; >TH = " & Cstr(TH_UTIL)
    MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
    sObjectList, sDetailMsg
If the user has asked to include the queue length threshold \((TH_{QLEN} <> -1)\) and the queue length threshold has been exceeded \((dQueueLength > TH_{QLEN})\), then the Callback function MSActions is used to raise an event with a message that the CPU and queue length thresholds have been exceeded:

```pascal
Elseif dQueueLen > TH_QLEN Then
    sDetailMsg = sCPUMsg + " utilization% is " & _
        Format$(dTotalTime, "0.00") & "; >TH = " & _
        Cstr(TH_UTIL) & " AND" & chr$(10) & "CPU queue _
        length is " & Cstr(dQueueLen) & "; >TH = " & _
        Cstr(TH_QLEN)
    MSActions Severity, sCPUMsg & " Overloaded", AKPID, _
    sObjectList, sDetailMsg
End If
```

Finally, if the user has asked to collect data, then the Callback function DataLog is used to return the current values of CPU usage and queue length. This data is returned whether or not the thresholds have been exceeded:

```pascal
If DO_DATA = "y" Then
    sDetailMsg = sCPUMsg + " utilization% is: " & chr$(10) & _
        "Privileged " & Format$(dPrivilegeTime, "0.00") & _
        chr$(10) & "User " & Format$(dUserTime, "0.00") & _
        chr$(10) & "Total " & Format$(dTotalTime, "0.00")
    DataLog lStreamID, dTotalTime, sDetailMsg
End If
```
The modified script, NT_CpuLoadedEx.qml

Now that you know how Samples_CpuLoaded.qml works, you can modify it to obtain more information. The expanded Knowledge Script is called Samples_CpuLoadedEx.qml.

In Samples_CpuLoadedEx.qml, the managed object method CPU.TopUsageValue() will be used to return information about the five processes that use the most CPU resources. The syntax of this method is:

```
CPU.TopUsageValue HowMany, AgtMsg, Flags
```

This function reports the total CPU consumption of all processes and, optionally, details about the processes consuming the most CPU. You can also use this function to check whether a particular application process is running or consuming an unexpected amount of CPU time (run-away process).

The function returns a double representing the overall CPU percentage used by all processes. A return value of -1 indicates an error condition.

The text string, AgtMsg, returns a list of process names and overall utilization numbers, sorted by the utilization percentage.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>HowMany</td>
<td>Long</td>
<td>Number of top CPU-consuming processes to include in the detail message (AgtMsg). If 0 is specified, all processes are returned. The default used in most Knowledge Scripts is 5, for example, to return details about the top 5 processes consuming the most CPU. Note that the return value for this function is the total CPU consumption (summation of all processes' CPU) irrespective of this setting.</td>
</tr>
<tr>
<td>Flags</td>
<td>Long</td>
<td>Set to 1 to return the detail message, AgtMsg, with details about top processes. If set to 0, no AgtMsg is returned.</td>
</tr>
</tbody>
</table>
Listing of Samples_CpuLoadedEx.qml

Samples_CpuLoadedEx.qml is exactly the same as Samples_CpuLoaded.qml, except that it calls CPU.TopUsageValue() to obtain the top five processes.

The new code is shown in bold and larger font:

Dim NT As Object
Dim CPU As Object
Const UNITPERCENT = "%"

' This sub routine checks for the processor time, user time, and queue length to see if they exceed the given thresholds for a given cpu, or the overall cpu
Sub CpuCheck(sCPUName As String)
Dim dUserTime#, dPrivilegeTime#, dTotalTime#, dQueueLen#
Dim sDetailMsg$, sObjectList$, sCPUMsg$
Dim lStreamID As Long

Dim sProcessNames As String
Dim lRetCode As Long

If (sCPUName = "") Then
    ' Set the machine object as the resource. This will cause the machine object to blink if there is an event.
    sObjectList = "NT_CPUFolder = " + NT_CPUFolder
    lStreamID = 0
    sCPUMsg = "Overall CPU"
Else
    ' Set the individual cpu name as the resource. This will cause the individual cpu object to blink if there is an event for each individual cpu
    sObjectList = "NT_CPUNumber = " + sCPUName
    lStreamID = Val(sCPUName)
    sCPUMsg = "CPU# " + sCPUName
End If

dTotalTime = CPU.UtilValue("PROCESSOR", sCPUName)
dUserTime = CPU.UtilValue("USER", sCPUName)
If dTotalTime = -1 Or dUserTime = -1 Then
    ' A return value of -1 indicates a failure to retrieve the value of the counter
MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
  sObjectList,"Processor or user counter not found _
  (Proc: " & Cstr(dTotalTime) & ", User: " & _
  Cstr(dUserTime) & ")"
  Exit Sub
End If

If dTotalTime > dUserTime Then
  dPrivilegeTime = dTotalTime - dUserTime
Else
  dPrivilegeTime = 0
End If

If IterationCount() = 1 Then
  If DO_DATA = "y" Then
    DataHeader "PROCESSOR Utilization - " & SCPUMsg _
    & UNITPERCENT, 0, lStreamID
  End if
End If

If DO_EVENT = "y" And dTotalTime > TH_UTIL Then
  dQueueLen = CPU.QueueLengthValue
  If dQueueLen = -1 Then
    MSActions PRM_KSERR, "Counter not found", "AKP_NULL", _
    sObjectList, "The queue length counter could _
    not be found"
    Exit Sub
  End If

  ' if TH_QLEN = -1 ignore query length value and raise
  ' event
  ' else if query length value exceeds threshold value then
  ' raise event
  If TH_QLEN = -1 Then

  lRetCode = CPU.TopUsageValue(5, sProcessNames, 1)
  sDetailMsg = SCPUMsg + " utilization% is " & _
  Format$(dTotalTime, "0.00") & _
  "; >TH = " & Cstr(TH_UTIL) _

  & chr$(10) & chr$(10) & "top 5 processes _
are: ” & sProcessNAMES

MSActions Severity, sCPUMsg & ” Overloaded”, AKPID, _
 sObjectList, sDetailMsg
 Elseif dQueueLen > TH_QLEN Then

 !RetCode = CPU.TopUsageValue(5, sProcessNames, 1)
 sDetailMsg = sCPUMsg + ” utilization% is ” & _
 Format$(dTotalTime, "0.00") & ”; >TH = ” & _
 Cstr(TH_UTIL) & ” AND” & chr$(10) & ”CPU queue length is ” & Cstr(dQueueLen) & ”; >TH = ” & _
 Cstr(TH_QLEN) _

 & chr$(10) & chr$(10) & ”top 5 processes _
 are: ” & sProcessNAMES

MSActions Severity, sCPUMsg & ” Overloaded”, AKPID, _
 sObjectList, sDetailMsg
 End If

End If

If DO_DATA = ”y” Then
 sDetailMsg = sCPUMsg + ” utilization% is: ” & chr$(10)
 & _
 ”Privileged ” & Format$(dPrivilegeTime, ”0.00”) & _
 chr$(10) & ”User ” & Format$(dUserTime, ”0.00”) & _
 chr$(10) & ”Total ” & Format$(dTotalTime, ”0.00”)
 DataLog lStreamID, dTotalTime, sDetailMsg
 End If
End Sub

Sub Main()
 Dim sCPUName$, sProgID$
 Dim iNumberOfCPU As Integer

 If IterationCount() = 1 Then
 ‘ Retrieve the prog id of the NetIQ NT MO COM object
 sProgID = MCGetMOID (”NetIQAgent.NT”, AppManID)
 Set NT = CreateObject (sProgID)
Set CPU = NT.CPU
End If

iNumberCPU = ItemCount(NT_CPUNumber, ",")
If iNumberCPU = 1 Or DO_OVERALL = "n" Then
  ' Check each individual CPU in the object list
  For I = 1 To iNumberCPU
    sCPUName = Item$(NT_CPUNumber, I, ",")
    CpuCheck sCPUName
  Next I
Else
  ' Just check the overall CPU usage
  CpuCheck ""
End If
End Sub
Chapter 6

Modifying a monitoring script written in Perl

This chapter dissects the code in a sample Knowledge Script called `Samples_HTTPHealth.qml`. This script sends an HTTP command to each URL in a user-specified list and reports when the Web server does not respond.

In the final section of this chapter, `Samples_HTTPHealth.qml` is modified to become `Samples_HTTPHealthEx.qml`. In this modified script, the user can elect to be informed if the Web server is unable to supply a particular HTML page of the user's choice.

You should open each sample Knowledge Script in your Developer’s Console where you can look at it in the various views and open its Script Properties dialog box.

You will also benefit from running them in the AppManager Operator Console and experimenting with various Properties choices.

The following topics are covered in this chapter:

- Listing of the `Samples_HTTPHealth.qml` script
- Preliminary discussion
- Syntax of the Callback functions
- The program logic
- The modified script, `Samples_HTTPHealthEx.qml`

Listing of the `Samples_HTTPHealth.qml` script

Here is a listing of the code section of the script. The Script Parameters, included by AppManager as variables, are not shown.
# begin main script
use strict;
use NetIQ::Nqext;
use IO::Socket;
our $resmsg;
our @address_array;
our $address;
my $connection;
our $datavalue;
our $line;
our $idx;
format_list($AddressList);

$resmsg = "UNIX_MachineFolder = $UNIX_MachineFolder";

if ($AddressList eq ''){
    NetIQ::Nqext::CreateEvent($Severity, "The supplied address list is empty", "AKP_NULL", $resmsg, 0, "Enter a list of addresses separated by a comma. E.g. www.netiq.com,www.microsoft.com", "", 0, 0);
}

$idx = 0;
@address_array = split (',',$AddressList);
foreach $address (@address_array){
    $datavalue = 100;
    $idx++;
    # Create a socket connection to the specified address
    $connection = IO::Socket::INET->new (Proto => "tcp", PeerAddr => $address, PeerPort => "http(80)");
    unless ($connection){
        NetIQ::Nqext::CreateEvent($Severity, "Failed to connect to HTTP server $address", $Akpid, $resmsg, 0, "Failed to connect to HTTP server $address", "", 0, 0);
    }
    if ($Do_data eq "y"){
        NetIQ::Nqext::CreateData($idx . "$address", "HTTP health for $address", "$resmsg", 0, ",", 0, 0);
    }
    next;
}
# Send a head command to the specified address to see
# if it is a valid web server
$connection->autoflush (1);
print $connection "HEAD /index.html HTTP/1.0\n\n";

$line = <$connection>;
unless ($line){
    if ($Do_event eq "y"){
        NetIQ::Nqext::CreateEvent($Severity, "Failed to
 connect to HTTP server
 $address", $Akpid,
 $resmsg, 0, "Failed to
 connect to HTTP server
 $address", "", 0, 0);
    }
    $datavalue = 0;
}
if ($Do_data eq "y"){
    NetIQ::Nqext::CreateData($idx . "$address", "HTTP
 health for $address", ","
 $resmsg, $datavalue, ",", 0);
}

close ($connection);

### End main script

# get rid of extraneous commas, extra white spaces, etc.
sub format_list {
    my ($input) = @_;
    $input =~ s/\s+,/,/g;
    $input =~ s/,\s+/,/g;
    $input =~ s/\s+//g;
    $input =~ s/\s+$//g;
    $input =~ s/,+/,/g;
    $input =~ s/^,//g;
    $input =~ s/,$//g;
    chomp($input);
    $_[0] = $input;
}

Preliminary discussion

Recall from Chapter 2 the steps that the script undergoes when it is run:
1 A user chooses a script and drags it to the target object.

2 The **Properties** dialog box opens.

3 The user sets Script Parameters, the schedule, actions, etc.—or accepts the defaults—and closes the dialog box.

4 The Operator Console creates a job (an instance of the script along with the user configured Script Parameters, schedule, actions, etc.) in the AppManager repository.

5 The AppManager management server retrieves the job, the schedule, any action scripts, and so forth from the AppManager repository and forwards it all to the AppManager agent which will run the job. The final script has all Script Parameters and object types defined as variables with assigned values.

### User-set Script Parameters

There are four Script Parameters that the user can alter when launching this script. These Script Parameter will become variables (with values assigned) in the running script. The code must provide alternatives that depend on the values the user chose for these Script Parameters. The Script Parameters are:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Do_data</td>
<td>Collect data? (y/n)</td>
<td>If = “y”, data will be collected.</td>
</tr>
<tr>
<td>$Do_event</td>
<td>Event? (y/n)</td>
<td>If = “y”, an event will be fired when threshold is exceeded.</td>
</tr>
<tr>
<td>$AddressList</td>
<td>Web server address list (separated by commas and no spaces)</td>
<td>Comma-delimited list of connections (URLs) to test.</td>
</tr>
</tbody>
</table>
Object types

The object type for this script is:
<Type name="UNIX_MachineFolder"></Type>

When the script is dragged onto the target object the Operator Console will assign the appropriate value:

- **UNIX_MachineFolder** will be assigned the name of the target computer.

Actions

$Akpid determines what action scripts, if any, are run. If there are to be action scripts, they will be run when an event is raised—$Akpid is a parameter of the Callback function `CreateEvent()`. If no events are raised, no action scripts will be run.

**Note** Raising events is the mechanism used to launch action scripts. Other than calling an event with $Akpid as a (required) parameter, you do not write code to run action scripts.

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Severity</td>
<td>Event severity level</td>
<td>Severity level of event fired (if DO_EVENT=&quot;y&quot;).</td>
</tr>
<tr>
<td>$Akpid</td>
<td>The &quot;description&quot; of this variable is &quot;action taken,&quot; but the user does not see it. It is hidden in the Operator Console.</td>
<td>Action script or scripts to run. If none, the Operator Console program will set it to the default (AKP_NULL). <strong>NOTE</strong> The user does not see this Script Parameter in the Operator Console Properties dialog box, although you defined it in your Script Properties dialog box in the Developer's Console. If the user adds actions, the value of $Akpid will be altered by the Operator Console program.</td>
</tr>
</tbody>
</table>
The default for $Akpid in this script is "AKP_NULL" (no action), which is the default for $Akpid in most scripts. If the user adds actions with the Properties dialog box when setting up the job, the value of $Akpid will be changed to "1,2,3,4,...,n" when the user adds n actions (n >= 1).

Functions called in the code

The code calls three types of functions:

- NetIQ Callback functions, by which the script requests information or action from the AppManager agent running the job. See Chapter 12, “AppManager Callbacks for Perl.”

Here are the functions called in the code, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetIQ::NQEXT::CreateEvent</td>
<td>Callback function that raises an event.</td>
</tr>
<tr>
<td>split</td>
<td>Built-in Perl function that splits a delimited string into a list (array) of strings.</td>
</tr>
<tr>
<td>IO::Socket::INET-&gt;new</td>
<td>Instantiates the class IO::Socket::INET to open a socket connection.</td>
</tr>
<tr>
<td>NetIQ::NQEXT::CreateData</td>
<td>Callback function that sends data points back for logging and graphing.</td>
</tr>
<tr>
<td>autoflush</td>
<td>A method of the IO::Socket::INET class that deletes cached data in a socket connection.</td>
</tr>
<tr>
<td>print</td>
<td>Built-in Perl function that (in this case) prints to the socket connection.</td>
</tr>
<tr>
<td>close</td>
<td>A method of the IO::Socket::INET class that closes the socket connection.</td>
</tr>
<tr>
<td>chomp</td>
<td>Built-in Perl function that removes a newline character from the end of a string.</td>
</tr>
</tbody>
</table>
**Syntax of the Callback functions**

Refer to Chapter 12, “AppManager Callbacks for Perl” for more details.

**CreateData**

Sends data points for dynamic data streams. This function allows you to collect data for data streams that may be instantiated at each iteration.

**Syntax**

```
NetIQ::Nqext::CreateData (streamId, legend, dynaleg, objlist, val, agentmsg, msgtype [,schema] [,loglimit] [,lowWM] [,hiWM] [,deletefile])
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>Long, String</td>
<td>Data stream ID</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>Data stream legend. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>dynaleg</td>
<td>String</td>
<td>Dynamic legend, contains the dynamic information that can be used for reporting.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Corresponding object name where the data is collected.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>Current data point value.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Contains either a plain text or a message file name.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Related to agentmsg: 0 for plain text, 1 for message file.</td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>XML schema for dynamic table creation in RDB. Default is an empty string.</td>
</tr>
<tr>
<td>loglimit</td>
<td>Long</td>
<td>Datalog limit in # of days. Default is -1.</td>
</tr>
<tr>
<td>lowWM</td>
<td>Double</td>
<td>Low watermark. Default is -1.0.</td>
</tr>
<tr>
<td>hiWM</td>
<td>Double</td>
<td>High watermark. Default is -1.0.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Used only when msgtype=1. Default is TRUE.</td>
</tr>
</tbody>
</table>

CreateData returns nothing.
CreateEvent

Used by a Knowledge Script to send an event to the AppManager agent. The AppManager agent will apply additional rule processing and will determine whether to send a new event or a duplicated (collapsed) event to the AppManager management server.

Syntax

NetIQ::Nqext::CreateEvent(sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [,deletefile])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>Event severity</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>Event message</td>
</tr>
<tr>
<td>akp</td>
<td>String</td>
<td>Action name</td>
</tr>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value (to raise the event)</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either a plain text detail message or a message file name</td>
</tr>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Event source</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Event ID</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Agent message type: 0 for plain text, 1 to refer to a file</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Only used when msgtype=1. Default is TRUE</td>
</tr>
</tbody>
</table>

CreateEvent returns nothing.

The program logic

Recall that the running script will include the user-defined Script Parameters. For example, if the user accepts the defaults, the following will be pre-pended to the script’s code (with the UNIX machine name filled in by AppManager):

#### Begin KSID Section
our $AppManID = "4.1u.6.0.1";
our $KSVerID = "1.0";

#### End KSID Section

#### Begin Type Section
our $UNIX_MachineFolder = "";
#### End Type Section

#### Begin KPP Section
our $Do_event="y";
our $Do_data="n";
our $AddressList="www.netiq.com";
our $Severity=8;
our $Akpid="AKP_NULL";
#### End KPP Section

**The main script**

The lines
```
use NetIQ::Nqext;
use IO::Socket;
```
include the libraries for the AppManager Callback functions and the socke
Functions that we need.

Then, after declaring variables, the code begins with
```
format_list($AddressList);
```
This is a call to the function format_list, which will be discussed after the main part of the script. This function strips all white space and extraneous commas from $AddressList, which is the list of URLs entered by the user.

Next, the $resmsg variable is assigned the “object type” string. This string is a CreateEvent parameter that identifies the source of the event and tells AppManager which icon in the TreeView pane should blink when an event has occurred.

```
$resmsg = "UNIX_MachineFolder = $UNIX_MachineFolder";
```

If the string variable that lists the URLs to test is empty, meaning that the user did not enter a list as they should have, an event is raised that reports this:
if ($AddressList eq ''){
    NetIQ::Nqext::CreateEvent($Severity, "The supplied address list is empty", "AKP_NULL", $resmsg,
    0, "Enter a list of addresses separated by a comma. E.g. www.netiq.com,www.microsoft.com",
    ",", 0, 0);
}

In the block of code immediately above, note two things:

- The action variable is "AKP_NULL", so that no action scripts will be run at this time (this event is created because of an error condition, not because the user-defined threshold has been exceeded).
- Even if the URL list is empty, script execution continues.

The URL list entered by the user, $AddressList, is a comma-delimited string of URLs. The next statement converts this string to an array of URLs:

@address_array = split (',',$AddressList);

The entire remainder of the main script is a foreach loop that steps through the array of URLs, one URL at a time. If the list is empty, the loop will not execute. At the beginning of each pass through the loop, $data_value is set to 100 which represents a “healthy” URL. If the URL is later found to not be healthy, $data_value will be reset to 0. These values only have meaning if data is to be collected ($Do_data = "y").

foreach $address (@address_array){
    $data_value = 100;
    $idx++;

Now it is time to open a socket connection to $address. This is done by instantiating a new IO::Socket::INET connection with $address as a parameter in the constructor.

# Create a socket connection to the specified address
$connection = IO::Socket::INET->new (Proto => "tcp",
    PeerAddr => $address,
    PeerPort => "http(80)");
If creation of the socket fails, the constructor will return \texttt{undef}. We test for this. If \texttt{undef} is returned, then:

- An event is raised signaling failure.
- A datapoint with a value of 0 is sent to the data stream for that URL, but only if \texttt{$\$\text{Do\_data eq "y"}}. The data stream ID is the URL preceded by its place in the list—for example, if the fourth URL in the list is \texttt{www.netiq.com}, its stream ID will be:

\texttt{$\text{idx} \cdot "$\text{address}" = 4www.netiq.com$}.
- Execution returns to the beginning of the \texttt{foreach} loop.

```perl
unless ($connection){
    # Send a head command to the specified address to see
    # if it is a valid web server
    $connection->autoflush (1);
    print $connection "HEAD /index.html HTTP/1.0\n\n";
    # If there is no response, it means that the Web server is absent or is not able to answer. In such a case, attempting to read the first line of the reply (i.e., <$connection>) will return undef. We assign the first line
```
of the answer to $line and test it. Unless it is not undef (that is, unless it has contents), we raise an event reporting failure and also set $datavalue to 0.

**Note** It isn’t important that the Web server can serve the index.html page. If the Web server can respond, but cannot serve index.html, it will return an error message, not undef. This means that the Web server is “healthy,” which is what we are looking for.

```
$line = <$connection>; 
unless ($line){
  if ($Do_event eq "y"){
    NetIQ::Nqext::CreateEvent($Severity, "Failed to connect to HTTP server $address", $Akpid,
    $resmsg, 0, "Failed to connect to HTTP server $address", ",", 0, 0);
  }
  $datavalue = 0;
}
```

At this point in the script, the value of $datavalue is 100 for a successful connection to the Web server, or 0 for a failed connection. We send back a datapoint to the $idx . "$address" data stream, provided that $Do_data eq "y". Then, the socket connection is closed and the main script is finished.

```
if ($Do_data eq "y"){
  NetIQ::Nqext::CreateData($idx . "$address", "HTTP health for $address", ",", $resmsg, $datavalue, ",", 0);
}
```

```
close ($connection);
```
Web server can return the index.html page, only that it responds.

The format_list subroutine

This subroutine, which is called on the user-input list of URLs at the beginning of the main script, uses the regular expression operator to make sure that the string listing the URLs is properly formatted: a list of URLs, separated by commas, with no white space and no empty elements (an empty element is two successive commas with nothing between them).

```perl
# get rid of extraneous commas, extra white spaces, etc.
sub format_list {
    my ($input) = @_;
    $input =~ s/\s+,/,/g;
    $input =~ s/,\s+/,/g;
    $input =~ s/\s+//g;
    $input =~ s/,+/,/g;
    $input =~ s/^,//g;
    $input =~ s/\s+$/; chomp($input);
    $_[0] = $input;
}
```

Here is what each line does:

<table>
<thead>
<tr>
<th>Line</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$input =~ s/\s+,/,/g;</td>
<td>Substitutes a comma for one or more white spaces followed by a comma.</td>
</tr>
<tr>
<td>$input =~ s/,\s+/,/g;</td>
<td>Substitutes a comma for a comma followed by one or more white spaces.</td>
</tr>
<tr>
<td>$input =~ s/\s+//g;</td>
<td>Substitutes nothing for any white space at the beginning of the string.</td>
</tr>
<tr>
<td>$input =~ s/\s+$/;</td>
<td>Substitutes nothing for any white space at the end of the string.</td>
</tr>
<tr>
<td>$input =~ s/,+/,/g;</td>
<td>Substitutes one comma for two or more consecutive commas.</td>
</tr>
<tr>
<td>$input =~ s/^,//g;</td>
<td>Substitutes nothing for a comma at the beginning of the string.</td>
</tr>
</tbody>
</table>
Developing Custom Knowledge Scripts

The modified script, Samples_HTTPHealthEx.qml

The code in Samples_HTTPHealth.qml checks to verify that the Web server at each URL in the user-supplied string $AddressList is responding. The Web server does not need to return a particular page to be considered healthy. The HEAD command is used to ask for the index.html page, but the script does not raise an event if the Web server reports failure to serve that page. An event is raised only if the Web server fails to respond at all.

In Samples_HTTPHealth.qml, the user can specify the name of a page in the Script Parameter $Html_page and the script will report a “health problem” if the Web server cannot return that page (only if another new user-defined Script Parameter, $Do_OkEvent, is set to "y"). In this case, it is not sufficient that the Web server simply responds—it must respond that it can serve the desired page.

The HEAD command in Samples_HTTPHealth.qml

    print $connection "HEAD /index.html HTTP/1.0\n\n";

is changed to

    print $connection "HEAD /$Html_page HTTP/1.0\n\n";

in the Samples_HTTPHealthEx.qml script.

In Samples_HTTPHealthEx.qml, whenever $Do_OkEvent is set to "y", the HEAD command must report success. If the HEAD command succeeds, the first line returned will be “HTTP/1.1 200 OK” (by comparison, a typical failure would return something like “HTTP/1.1 500 Server Error”). We assign the first line of the returned message to $line and then test it. The condition in the statement

    if ($line !~ /HTTP/1.1 200/)

<table>
<thead>
<tr>
<th>Line</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$input =~ s/,$//g;</td>
<td>Substitutes nothing for a comma at the end of the string.</td>
</tr>
<tr>
<td>chomp($input);</td>
<td>Deletes a newline character at the end of the string.</td>
</tr>
</tbody>
</table>

Substitutes nothing for a comma at the end of the string.

chomp($input); Deletes a newline character at the end of the string.
will be true if the string "HTTP/1.1 200" is not found in the Web server response to the **HEAD** command.

**Altered code**

The portion of `Samples_HTTPHealth.qml` that is altered to produce `Samples_HTTPHealthEx.qml` is shown below. The new code is shown in a larger font and in bold.

```perl
# Send a head command to the specified address to see # if it is a valid web server
$connection->autoflush (1);
print $connection "HEAD /$Html_page HTTP/
1.0\n\n";
while (<$connection>){
    # Need to remove the ^M character # because it doesn't display well in # the operator console.
    s/\cM//;
    $line .= $_;
}
$line = <$connection>;
unless ($line){
    if ($Do_event eq "y"){
        NetIQ::Nqext::CreateEvent($Severity, "Failed to connect to HTTP server $address", $Akpid, $resmsg, 0, "Failed to connect to HTTP server $address", ",", 0, 0);
    }
    $datavalue = 0;
}
else {
    if ($line !~ /HTTP/1\.1 200/) {
        if ($Do_OKEvent eq "y") {
            NetIQ::Nqext::CreateEvent($Severity, "Bad page, $Html_page, for $address", $Akpid, $resmsg, 0, "This job successfully connected to HTTP server $address", ",", 0, 0);
        }
    }
}
```

Chapter 6 • Modifying a monitoring script written in Perl  131
Developing Custom Knowledge Scripts

ADDRESS, however, the requested page, HTML_PAGE, returned: 
\n$line", ","0, 0);   

}   
}   
}   

There is one new feature in the Samples_HTTPHealth.qml code above that needs explanation. In the altered script, there is a new user-defined Script Parameter called $Do_OkEvent. If this Script Parameter is set to “y”, then the script will raise an event when the HEAD command reports failure to serve HTML_PAGE. Further, this event will supply the error message returned by the HEAD command as a message string. The error message returned by the HEAD command will contain ^M (that is, Ctrl + M) characters. These characters will display poorly in the AppManager Console, so the script removes them with the code:

while (<$connection>>){
    # Need to remove the ^M character
    # because it doesn't display well in
    # the operator console.
    s/$\cM//;
    $line .= $;  
}

This loop steps through each line in the message returned by the HEAD command and substitutes nothing for \cM characters. Note that the $line string is empty when the loop begins.
Chapter 7

Modifying an action script written in VBScript

In AppManager, “performing an action” means running an action Knowledge Script as a result of an event being raised in some other type of script.

This chapter describes an action script, Action_WriteToFile, that does what its name implies—it writes a message to a file. This Knowledge Script, written in VBScript, is similar to the Summit BasicScript action script, Action_WriteMsgToFile.

Action_WriteMsgToFile and Action_WriteToFile will write either of two messages to a file: a default message or a custom message. In the last part of this chapter, Action_WriteToFile will be modified so that the script can also write both of the custom and default messages. The modified script is called Action_WriteToFileEx.

This chapter covers the following topics:

- Setting up to perform actions
- Invoking actions
- Events without actions
- Ending actions
- XML messages
- Listing of the Action_WriteToFile.qml script
- User-set Script Parameters
- Parameters supplied by AppManager
- Functions called in the code
- Syntax of the Callback functions
Developing Custom Knowledge Scripts

- The program logic
- The modified script, Action_writeToFileEx.qml

Setting up to perform actions

Actions can be defined for “normal” (monitoring and report), discovery, and install scripts. It is not possible to define further actions for action scripts.

Actions for a Knowledge Script can be defined either:

- by the script developer, using the Script Properties dialog box in the Developer’s Console, or
- by users of the AppManager Operator Console, using the Properties dialog box that opens when a script is dragged to a target object in the TreeView pane.

Script developers

When developing a monitoring, reporting, or discovery Knowledge Script, you should use the Parameters tab of the Script Properties dialog box in the Developer’s Console to define a Script Parameter called AKPID. You should also give this Script Parameter the default value “AKP_NULL”. You are not forced to do this, but trouble can arise if you do not.

You, the script developer, can also define actions for your script using the Script Properties dialog box. This is hardly ever done by script developers, as it is difficult to predict what type of action a user will want performed. You should define actions rarely, if ever.

Note If you do, in fact, define actions yourself, you might think that setting a default value of “AKP_NULL” for AKPID is unnecessary. However, a user can undo your choices of actions when setting up a Knowledge Script job, so that a default value will be required in any case.
AppManager Operator Console users

When an AppManager Operator Console user drags a script to a target object in the TreeView pane, the Properties dialog box opens. For every type of Knowledge Script except action scripts, the dialog box will have an Actions tab. In this tab, users can add as many actions as they desire. In the rare event that the script writer associated actions with this script, the user can delete them.

Caution You must choose Action for the Knowledge Script type in the Header tab of the Script Properties dialog box of the Developer's Console. If you fail to make this choice for an action script, it will not be available as a new action to an Operator Console user in the Action tab of the Knowledge Script Properties dialog box:
Invoking actions

It is the responsibility of non-action scripts to invoke actions. Action scripts are executed only when events are raised. More specifically, when:

- actions have been associated with a monitoring, discovery, install, or reporting Knowledge Script job,
- an event is raised by one of those scripts, and
- the event Callback’s action parameter is set to AKPID.

When you are developing a script, you can choose to raise an event that does not call any action scripts that may be chosen by a user. In the Callback function that raises the event (CreateEvent in VBScript, MSActions in Summit BasicScript), you set the action parameter to “AKP_NULL” rather than AKPID.

Thus, for any given event, you can choose to have all action scripts executed or none. If you set the action parameter of CreateEvent or MSActions to AKPID, all actions chosen by a user will be executed. If the parameter is set to “AKP_NULL” no action script will be executed.

Note There is no mechanism for you to associate several different actions with a script and choose which one should be executed when a particular event is raised.

Events without actions

In general, you want to generate events without invoking actions when your script detects an error condition that you feel the user should be aware of. For example, if the user enters an invalid script parameter, the script should raise an event, but not invoke an action.

Monitoring scripts should invoke actions only if the conditions or thresholds that the user wants to monitor have been met or exceeded.
Ending actions

It is the responsibility of the action script itself to signal the end of an action.

Toward the end of your action script, your code should signal the completion of the action script by raising an event with the action parameter set to “AKP_COMPLETE.” For example, in the Action_WriteToFile script, the final statement in the code is:

```
NQEXT.CreateEvent 2, "", "AKP_COMPLETE", "", 0, "", "", 0, 0
```

An event that sets the AKPID parameter to "AKP_COMPLETE" will cause the Message in the Action tab of the Event Properties dialog box to read:

- “Action Complete” if the event message (second parameter in the event parameter list) is an empty string, as it is in the example immediately above, or
- the event message, if it is not an empty string.

If you do not raise an event with the action parameter set to “AKP_COMPLETE”, the Message in the Action tab of the Event Properties dialog box will continue to read “<Location> Action in Progress,” even though the action has, in fact, completed.

**Note** Any event raised with an action parameter other than “AKP_COMPLETE” will create a new event.

XML messages

Beginning with AppManager 5.0, you can write custom event messages for your monitoring scripts in XML format. AppManager will parse these XML messages to create formatted tables in the Message pane of the Event Properties dialog box. Here is an example from an event raised by the webservices_LinkSummary Knowledge Script.
By comparison, the screen below shows an event message that is not in XML format.
The importance of XML messages in this chapter is this: You must take the XML format possibility into account in your action scripts. The event message will be passed to the action script—since this message may be in either plain text or in XML format, the action script will need to take this into consideration. The Action_WriteToFile and Action_WriteToFileEx examples in this chapter show how to do this.
Listing of the Action_WriteToFile.qml script

Here is a listing of the code section of Action_WriteToFile.qml. The Script Parameters, included by AppManager as variables, are not shown.

Const MIN_MC_VERSION = "4.5"
Dim strAgtVersion ' The NetIQmc agent version

' Function converts the detail message from XML into normal text if needed
Function PreProcessForXML (strXMLMsg)
    Dim strProcessedMsg
    Dim lngRetCode

    NQEXT.GetVersion "netiqmc.exe", strAgtVersion
    ' Conversion of XML text to normal text is only supported
    ' in AppManager agent version 5.0 and higher
    If (strAgtVersion >= MIN_MC_VERSION) Then
        lngRetCode = NQEXT.EventXMLToPlainText(strXMLMsg,
            strProcessedMsg)
    End If

    Select Case lngRetCode
        Case 0
            PreProcessForXML = strProcessedMsg
        Case -1 ' Malformed XML Doc
            NQEXT.CreateEvent 2, "EventXMLToPlainText Failed.", "AKP_COMPLETE", "The XML is a malformed XML document", 0, "", "", 0, 0
            PreProcessForXML = strXMLMsg
        Case -2 ' Not event XML Doc
            PreProcessForXML = strXMLMsg
        Case -3 ' Miscellaneous
            NQEXT.CreateEvent 2, "EventXMLToPlainText Failed.", "AKP_COMPLETE", "XML Translation failed with unknown reason", 0, "", "", 0, 0
            PreProcessForXML = strXMLMsg
        Case Else
            PreProcessForXML = strXMLMsg
    End Select
End Function

Else
PreProcessForXML = strXMLMsg
End If
End Function

Sub Main
Dim objFso, objFile
Dim strMessage
Dim lngIoMode

Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8

' Check to see if we would like to append to the file _
' or overwrite it
If Append = "y" Then
    lngIoMode = ForAppending
Else
    lngIoMode = ForWriting
End If

If Filename = "" Then
    NQEXT.CreateEvent 2, "No file name was specified to _
    write to.", "AKP_COMPLETE", ",", 0, "", ",", 0, 0
    Exit Sub
End If

On Error Resume Next
Set objFso = CreateObject("Scripting.FileSystemObject")
If Err.Number <> 0 Then
    NQEXT.CreateEvent 2, "Failed to create file system _
    object: " & Err.Description, "AKP_COMPLETE", _
    ",", 0,"", ",", 0, 0
    Exit Sub
End If

' Open the text file or create it if necessary
Set objFile = objFso.OpenTextFile(Filename, _
    lngIoMode, True)
If Err.Number <> 0 Then
    NQEXT.CreateEvent 2, "Failed to create file: " _
    & Filename & " Error: " & Err.Description, _
    "AKP_COMPLETE", ",", 0, ",", ",", 0, 0
    Exit Sub
End If
On Error Goto 0
If Message = "" Then
    ' No message was supplied so use the default message
    objFile.WriteLine("JobID = " + JobID)
    objFile.WriteLine("KSName = " + KPName)
    objFile.WriteLine("Object Name = " + ObjList + ")")
    objFile.WriteLine("EventMsg = " + EventMsg)
    objFile.WriteLine("LongMsg = " + PreProcessForXML (AgentMsg))
Else
    ' Use the messaged that was supplied
    objFile.WriteLine(Message)
End If
objFile.Close
NQEXT.CreateEvent 2, "", "AKP_COMPLETE", 
    "", 0, "", "", 0, 0
End Sub

User-set Script Parameters

Users can set action script properties when the “calling script” is
dragged and dropped. As an example, assume you drag the
Knowledge Script NT_CPULoaded (the “calling script”) to a target CPU
in the AppManager Operator Console TreeView pane. In the
Properties dialog box, you select the Actions tab and add the
Action_WriteToFile script as your action for NT_CPULoaded.
After you have chosen Action_WriteToFile as your action, click the Properties button. This opens the Properties dialog box for Action_WriteToFile:
Here you must enter values for the Script Parameters required by Action_writeToFile. The Script Parameters are:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename</td>
<td>Name of the file to write to.</td>
<td>The name of the file to write to. If this Script Parameter is not given a value, the action will abort.</td>
</tr>
</tbody>
</table>
Parameters supplied by AppManager

For action scripts, unlike other types of scripts, AppManager adds a number of variables (constants in Summit BasicScript action scripts) to the beginning of the script when it is run. The variables have to do with the event that caused the action script to be launched.

You can use these AppManager-added variables in your script, but you cannot see them in any of the Developer’s Console views. You simply have to know that they are there and what they are:

<table>
<thead>
<tr>
<th>AppManager-added variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobID</td>
<td>The JobID of the “calling script” (the script that raised the error that caused the action script to execute).</td>
</tr>
<tr>
<td>Severity</td>
<td>The severity of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>MachineName</td>
<td>The name of the machine running the calling script whose event caused execution of the action script.</td>
</tr>
<tr>
<td>KPName</td>
<td>The Knowledge Script name of the “calling script” (the script that raised the error that caused the action script to execute).</td>
</tr>
</tbody>
</table>
Functions called in the code

The code calls two types of functions:

- Callback functions, by which the script requests information or action from the AppManager agent running the job. See Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript.”


Here are the functions and objects called in the script, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQEXT.GetVersion</td>
<td>Callback function that retrieves the version number of the AppManager agent or component where the action is running.</td>
</tr>
<tr>
<td>NQEXT.EventXMLToPlainText</td>
<td>Callback function that converts XML event messages to plain text.</td>
</tr>
<tr>
<td>NQEXT.CreateEvent</td>
<td>Callback function that raises an event.</td>
</tr>
</tbody>
</table>

Note: These variables are added when the action script is run on either the management server or the managed client.
### Syntax of the Callback functions

Refer to Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript” for more details.

#### GetVersion

Obtains the latest version string for the specified file name.

**Syntax**

```vbnet
GetVersion file, verstr
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>String</td>
<td>Name of NetIQ file, or any other file.</td>
</tr>
<tr>
<td>verstr</td>
<td>String</td>
<td>The returned version string (passed by reference).</td>
</tr>
</tbody>
</table>

GetVersion returns nothing.
EventXMLToPlainText

Converts event messages in XML format to plain text. AppManager 5.0 only.

Syntax

EventXMLToPlainText XMLMsg, ProcessedMsg

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLMsg</td>
<td>String</td>
<td>Message in XML format.</td>
</tr>
<tr>
<td>ProcessedMsg</td>
<td>String</td>
<td>The message after translation to plain text. (passed by reference).</td>
</tr>
</tbody>
</table>

EventXMLToPlainText returns 0 for success, -1 for malformed XML, -2 if the message is not XML, or -3 if translation failed for some other reason. Any other value represents failure for an unknown reason.

CreateEvent

Used by a Knowledge Script to send an event to the AppManager agent. The AppManager agent will apply additional rule processing and will determine whether to send a new event or a duplicated (collapsed) event to the AppManager management server.

Syntax

CreateEvent sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [,deletefile]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>The event severity. A value from 1 to 40.</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>The message to be displayed under the Message column in the Events tab.</td>
</tr>
</tbody>
</table>
CreateEvent returns nothing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>akp</td>
<td>String</td>
<td>Name of the action script to launch as a response to this event. You would normally create an AKPID parameter as part of your script. When the job is dropped and you select an action, the UI will fill in the AKPID variable with the action name. You will just need to pass in the AKPID variable to the script.</td>
</tr>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised. This value will determine which object in the TreeView pane to blink. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;UNIX_DiskObject = /mnt/cdrom&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. UNIX_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value to raise the event. This parameter is currently not used. Set to 0.0.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the detail message or a file name that contains the detail message. The detailed message is displayed in the Message tab of the Event Property dialog box. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Not used. Should always be empty.</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Not used. Should always be 0.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg parameter is a file name or the detailed message itself. If it is a file name, then the contents of the file are read and passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype != 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
The program logic

The function `PreProcessForXML` is at the beginning of the file. This is used to convert any custom messages in XML to plain text. This function is discussed after `Sub Main` is analyzed.

Sub Main

Recall that there are three user-definable Script Parameters used as constants in the script: `Filename`, `Message`, and `Append`.

`Sub Main` begins with this code:

```vbs
Dim objFso, objFile
Dim strMessage
Dim lngIoMode

Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8

' Check to see if we would like to append to the file _
' or overwrite it
If Append = "y" Then
    lngIoMode = ForAppending
Else
    lngIoMode = ForWriting
End If
```

The declared variables are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objFso</td>
<td>The VBScript file system object.</td>
</tr>
<tr>
<td>objFile</td>
<td>The VBScript text file object.</td>
</tr>
<tr>
<td>lngIoMode</td>
<td>Variable to hold the file I/O constant that represents the user’s choice of appending to the file or overwriting it.</td>
</tr>
</tbody>
</table>

The constants `ForWriting` and `ForAppending` will be used as input parameters to the VBScript `FileSystemObject` method.
OpenTextFile, to determine whether to overwrite the contents of the file or append to its existing contents.

**Note** The constant ForReading is included for completeness, but is not used.

Then, with the overwrite/append constants defined, the script chooses the appropriate constant and assigns it to lngIoMode:

- If the user kept the default for Append ("y"), lngIoMode is set to ForAppending.
- If the user changed the value of Append to "n", lngIoMode is set to ForWriting.

The next section of code checks to see if Filename is defined (the user should have given this Script Parameter a value).

```vbscript
If Filename = "" Then
    NQEXT.CreateEvent 2, "No file name was specified to write to.", "AKP_COMPLETE", "", 0, "", "", 0, 0
    Exit Sub
End If
```

If Filename is not defined, the Callback function CreateEvent is called to create an event that:

- Transmits the message "No file name was specified ....."
- Sets the action variable to "AKP_COMPLETE," indicating that the action script is ending.

Then Sub Main, and the script, exits.

The next section of code is going to call CreateObject to create a VBScript object and then call one of the object's methods. Either of these calls could result in an error. A VBScript run-time error will be reported to the operating system and will end execution, unless the script handles the errors. For this reason, the code is written to handle the errors.

```vbscript
On Error Resume Next
Set objFso = CreateObject("Scripting.FileSystemObject")
```

Chapter 7 • Modifying an action script written in VBScript 151
If Err.Number <> 0 Then
The last parameter of the `OpenTextFile` method call, when `True`, tells the
method to create a new file if it does not already exist.

Once again, if the `OpenTextFile` fails (`Err.Number` is non-zero), an event is raised with a failure message and the action variable set to
"AKP_COMPLETE." Then `Sub Main` is exited.

The last statement, `On Error Goto 0`, disables error handling for the
subsequent code.

At this point a file is opened for overwriting or appending, and all that
remains to be done is to write the desired message to the file. Two text
file object (`objFile`) methods will be called in the remaining code,
`objFile.WriteLine` and `objFile.Close`. Neither of these two
methods are likely to throw errors, so the script does not bother with
further error handling.

```vbs
If Message = "" Then
  ' No message was supplied so use the default message
  objFile.WriteLine("JobID = " + JobID)
  objFile.WriteLine("KSName = " + KPName)
  objFile.WriteLine("Object Name = " + ObjList + ">")
  objFile.WriteLine("EventMsg = " + EventMsg)
  objFile.WriteLine("LongMsg = " + 
                  PreProcessForXML (AgentMsg))
Else
  ' Use the message that was supplied
  objFile.WriteLine(Message)
End If
```

**Note** This code uses the AppManager-added variables discussed in
“Parameters supplied by AppManager” on page 145.

There are two possible messages that can get written to the file:

1. The contents of user-defined Script Parameter `Message`.
2. In the case that `Message` is empty, a “default message” that was
   created by the action script writer (in the script).
**WriteLine** writes a string plus a newline character, so the default message is written out line-by-line:

<table>
<thead>
<tr>
<th>Message Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;JobID = &quot; + JobID</td>
<td>The JobID of the &quot;calling script&quot; (the script that raised the error that caused the action script to execute). This is provided by the AppManager infrastructure.</td>
</tr>
<tr>
<td>&quot;KSName = &quot; + KPName</td>
<td>The Knowledge Script name of the &quot;calling script&quot; (the script that raised the error that caused the action script to execute). This is provided by the AppManager infrastructure.</td>
</tr>
<tr>
<td>&quot;ObjectName = &lt;&quot; + ObjList + &quot;&gt;&quot;</td>
<td>The obj parameter of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>&quot;EventMsg = &quot; + EventMsg</td>
<td>The evtmsg parameter (event message) of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>&quot;LongMsg = &quot; + PreProcessForXML(-AgentMsg)</td>
<td>The agentmsg parameter (optional long message) of the calling script event that caused execution of the action script. <strong>NOTE:</strong> This message, defined by the author of the calling script, may have been written in XML format. Therefore, the <strong>PreProcessForXML</strong> function is used to convert it to plain text, if necessary.</td>
</tr>
</tbody>
</table>

Finally, the text file object is closed, a "success" event is raised, and the Main subroutine exits.

This **CreateEvent** call differs from all the previous calls in this script. The previous calls were all in response to an error or failure and their **evtmsg** parameter (the second parameter) contained an error message. In this case, since the action completed successfully, you should pass in an empty string, ",", instead of an error message. As a result, the user will see "Action Complete" for the action status in the Event Properties window.

```
objFile.Close
NQEXT.CreateEvent 2, ",", "AKP_COMPLETE",_ 
",", 0, ",", ",", 0, 0
End Sub
```

**Note** Do not confuse "Action Complete" with "AKP_COMPLETE".
“AKP_COMPLETE” signals to AppManager that your script has completed. “Action Complete” is written to the Event Properties dialog box only when the script has completed successfully.

**Function PreProcessForXML**

Beginning with AppManager 5.0, Knowledge Script developers can create monitoring script event messages in XML. AppManager will parse these XML messages to create formatted tables in the Message pane of the Event Properties dialog box.

We do not want messages that we write to a text file to contain XML tags, so we want some way of stripping these tags before writing the message to the file.

The Callback function `NQEXT.EventXMLToPlainText` converts a NetIQ XML-formatted message to plain text. The AppManager agent on the computer running the script containing this function must be version 5.0 or later.
The optional long message for any particular monitoring script may or may not be written in XML. Since we have no way of knowing which it is, we must call `NQEXT.EventXMLToPlainText` for every such message. The `Action_WriteMessageToFile` script includes a function that checks the AppManager agent version number, calls the Callback function `NQEXT.EventXMLToPlainText`, and handles any errors returned by the Callback.

`NQEXT.EventXMLToPlainText(strXMLMsg)` takes one parameter, the message text to convert. The function begins by checking that the AppManager version number is greater than or equal to 4.5 (this is the version number for AppManager 5.0). If the version number is less than 4.5, the function simply returns the name of the input text, without attempting conversion, and then exits.

```vbs
Const MIN_MC_VERSION = "4.5"
Dim strAgtVersion ' The NetIQmc agent version

' Function converts the detail message from XML into normal text if needed
Function PreProcessForXML (strXMLMsg)
    Dim strProcessedMsg
    Dim lngRetCode
    NQEXT.GetVersion "netiqmc.exe", strAgtVersion
    ' Conversion of XML text to normal text is only supported in AppManager agent version 5.0 and higher
    If (strAgtVersion >= MIN_MC_VERSION) Then
        If the version number is sufficient, `NQEXT.EventXMLToPlainText` is called to convert the input message, `strXMLMsg`, to plain text output, `strProcessedMsg`.

        Since `NQEXT.EventXMLToPlainText` can return several different values, a `Select` block is used to handle the alternatives.

        `lngRetCode = NQEXT.EventXMLToPlainText(strXMLMsg, _ strProcessedMsg)`

        Select Case lngRetCode
            Case 0
                PreProcessForXML = strProcessedMsg
            
```
Case -1 'Malformed XML Doc
    NQEXT.CreateEvent 2, "EventXMLToPlainText Failed.", "AKP_COMPLETE", "The XML is a malformed XML document", 0, "", "", 0, 0
    PreProcessForXML = strXMLMsg

Case -2 'Not event XML Doc
    PreProcessForXML = strXMLMsg

Case -3 ' Miscellaneous
    NQEXT.CreateEvent 2, "EventXMLToPlainText Failed.", "AKP_COMPLETE", "XML Translation failed with unknown reason", 0, "", "", 0, 0
    PreProcessForXML = strXMLMsg

Case Else
    PreProcessForXML = strXMLMsg
End Select

<table>
<thead>
<tr>
<th>Return value</th>
<th>Meaning</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The input message was in the proper XML format and was successfully converted.</td>
<td>The PreProcessForXML function returns the converted text, strProcessedMsg.</td>
</tr>
<tr>
<td>-1</td>
<td>The input message is in XML format, but it is not well-formed XML. Conversion failed.</td>
<td>The PreProcessForXML function returns the input text, strXMLMsg.</td>
</tr>
<tr>
<td>-2</td>
<td>The input message is not in XML format. Conversion failed.</td>
<td>The PreProcessForXML function returns the input text, strXMLMsg.</td>
</tr>
<tr>
<td>-3</td>
<td>The input message is in XML format, but something unknown caused conversion to fail.</td>
<td>The PreProcessForXML function returns the input file, strXMLMsg.</td>
</tr>
<tr>
<td>Any other integer</td>
<td>Conversion failed for some other reason.</td>
<td>The PreProcessForXML function returns the input file, strXMLMsg.</td>
</tr>
</tbody>
</table>

In two cases, -1 and -3, where the input file is XML but could not be converted, CreateEvent is used to raise an event and return an error
message. In the other cases, where the message is not XML, no event is raised.

**Note** In the event that the optional long message is indeed in XML, but the conversion by `NQEXT.EventXMLToPlainText` fails, the Sub Main line of code

```vbnet
objFile.WriteLine("LongMsg = " + _
    PreProcessForXML (AgentMsg))
```

will print the XML message to the text file.

### The modified script, Action_writeToFileEx.qml

In the `Action_WriteToFile` script, the message written to file is either a “default” (defined by the script writer, in the script) or a user-supplied message. This is the code to make this choice:

```vbnet
If Message = "" Then
    ' No message was supplied so use the default message
    objFile.WriteLine("JobID = " + JobID)
    objFile.WriteLine("KSName = " + KPName)
    objFile.WriteLine("Object Name = <" + ObjList + ">")
    objFile.WriteLine("EventMsg = " + EventMsg)
    objFile.WriteLine("LongMsg = " + _
        PreProcessForXML (AgentMsg))
Else
    ' Use the message that was supplied
    objFile.WriteLine(Message)
End If
```

In many cases, a user may want to write both of these messages. The `Action_WriteToFileEx` script is a modification of the `Action_WriteToFile` script that does this. A new user-definable Script Parameter has been added, `PrependMsg`. If the user sets this Script Parameter to “y” (default = “n”), that means that the user's message should be written first, followed by the default message. Then, the code immediately above is altered (new code shown in larger font and bold) like this:

```vbnet
If Message = "" Then
    ' No message was supplied so use the default message
    objFile.WriteLine("JobID = " + JobID)
    objFile.WriteLine("KSName = " + KPName)
    objFile.WriteLine("Object Name = <" + ObjList + ">")
    objFile.WriteLine("EventMsg = " + EventMsg)
    objFile.WriteLine("LongMsg = " + _
        PreProcessForXML (AgentMsg))
Else
    ' Use the message that was supplied
    objFile.WriteLine(Message)
End If
```
If Message = "" Or PrependMsg = "y" Then
  ' No message was supplied so use the default message
  objFile.WriteLine(Message)
  objFile.WriteLine("JobID = " + JobID)
  objFile.WriteLine("KSName = " + KPName)
  objFile.WriteLine("Object Name = <" + ObjList + ">")
  objFile.WriteLine("EventMsg = " + EventMsg)
  objFile.WriteLine("LongMsg = " + _
                  PreProcessForXML (AgentMsg))
Else
  ' Use the message that was supplied
  objFile.WriteLine(Message)
End If

With this simple modification, the first part of the If block will write both messages if PrependMsg = "y," irrespective of the value of Message. If PrependMsg = "y" and Message is empty, an empty line will be written before the default message.

The Else block, where only Message is written, will be reached only if Message has a value and PrependMsg = "n."

Note This script will behave exactly like Action_WriteToFile if PrependMsg = "n," except for an extra blank line before the default message.
Chapter 8

Modifying an action script written in Summit BasicScript

This chapter presumes that you have read all of the introductory material in the previous chapter. If you have not already done so, please read Chapter 7 through the end of the section titled “XML Messages.”

This chapter describes an action script, Action_Messenger, that sends a message using the Windows Message Service. This Knowledge Script, written in Summit BasicScript, is very similar to the AppManager action script of the same name.

Action_Messenger will send either of two messages: a default message or a custom message. In the last part of this chapter, the script will be modified so that the script can also send both the custom and the default messages. The modified script is called Action_MessengerEx.

This chapter covers the following topics:

- Listing of the Action_Messenger.qml script
- User-set Script Parameters
- Parameters supplied by AppManager
- Functions called in the code
- Syntax of the Callback functions
- The program logic
- The modified script, Action_MessengerEx.qml
Listing of the Action_Messenger.qml script

Here is a listing of the code section of Action_Messenger.qml. The Script Parameters, included by AppManager as constants, are not shown.

Const QUO = chr$(34) ' a double quote
Const NL = chr$(10) ' newline
Const MAX_RETRY = 5 ' maximum number of times to retry
' sending the message

Declare Function NetMessageBufferSend Lib "netapi32.dll" _
(ByVal pszServer As String, ByVal pszRecipient As String, _
ByVal pszSender As String, ByVal pbBuffer As String, _
ByVal cbBuffer As Long) As Long

Const V3GSP1 = "3.0.370.0"
Const MIN_MC_VERSION = "4.5"
Dim sAgtVersion$ ' The NetIQmc agent version

Function PreProcessForXML (sXMLMsg As String) As String
 Dim sProcessedMsg As String
 Dim lRetCode As Long

If (sAgtVersion >= MIN_MC_VERSION) Then
 lRetCode = EventXMLToPlainText(sXMLMsg, sProcessedMsg)
 Select Case lRetCode
 Case 0
  PreProcessForXML = sProcessedMsg
 Case -1 'Malformed XML Doc
  MSActions 2, "EventXMLToPlainText Failed.", _
  "AKP_COMPLETE", ",", "The XML is a _
  malformed XML document"
  PreProcessForXML = sXMLMsg
 Case -2 'Not event XML Doc
  PreProcessForXML = sXMLMsg
 Case -3 ' Miscellaneous
  MSActions 2, "EventXMLToPlainText Failed.", _
  "AKP_COMPLETE", ",", "XML Translation _
  failed with unknown reason"
PreProcessForXML = sXMLMsg

Case Else
    PreProcessForXML = sXMLMsg
End Select
Else
    PreProcessForXML = sXMLMsg
End If
End Function

Sub Main()
Const vbUnicode = 64
Dim lResult As Long
Dim lMsgLen As Long
Dim lRetry As Long
Dim sTargetName As String
Dim sMessage As String
Dim sHostname As String
Dim sErrMsg As String
Dim bError As Boolean

'Get MC version
sAgtVersion = ""
MCVersion "netiqmc.exe", sAgtVersion

sErrMsg = "Obj/Err:"
bError = False

If Message = ""
' No message was supplied so use the default message
sMessage = "JobID = " + JobID + NL + "" + KPName + NL + "" + MachineName + NL + "" + ObjList + "" + EventMsg + NL + LongMsg = _ + PreProcessForXML (AgentMsg) + NL
Else
' Use the messaged that was supplied
sMessage = Message
End If

For I = 1 To ItemCount(Recipient, ",")
    sTargetName = Item$(Recipient, I, ",")
lMsgLen = Len(sMessage)
lRetry = 0

'Truncate message if too long
If lMsgLen > 1024 Then
    lMsgLen = 1024
    sMessage = Mid(sMessage, 1, lMsgLen) & "...
End If

sHostname = GetMachName
resend:
If sAgtVersion < V3GSP1 Then
    lResult = NetMessageBufferSend ( _
        StrConv("",vbUnicode), _
        StrConv(sTargetName,vbUnicode), _
        StrConv(sHostname,vbUnicode),_
        StrConv(sMessage,vbUnicode), _
        Len(StrConv(sMessage,vbunicode)))
Else
    lResult = MCNetMessageBufferSend ("", _
        sTargetName, _
        sHostname,_
        sMessage)
End If

If lResult <> 0 Then
    lRetry = lRetry + 1
    If (lRetry < MAX_RETRY) Then
        MCSleep 100
        GoTo resend
    End If
    If (bError = True) Then
        sErrMsg = sErrMsg & ","
    Else
        bError = True
    End If
    sErrMsg = sErrMsg & sTargetName & "/" & CStr(lResult)
End If
Next I

If bError=True Then
    MSActions 2, sErrMsg, "AKP_COMPLETE", "", ""
    MSActions 2, "Action_Messenger failed", "AKP_NULL", _
        ",", sErrMsg
Else
    MSActions 2, "", "AKP_COMPLETE", "", ""
End If
User-set Script Parameters

Users can set action script properties when the “calling script” is dragged and dropped. As an example, assume you drag the Knowledge Script `NT_CPULoaded` (the “calling script”) to a target CPU in the AppManager Operator Console `TreeView` pane. In the Properties dialog box, you select the Actions tab and add the `Action_Messenger` script as your action for `NT_CPULoaded`.

```
Exit Sub
End Sub
```
After you have chosen Action_Messenger as your action, you click the Properties button. This opens the Properties dialog box for Action_Messenger:

Here you must enter values for the Script Parameters used in the Action_Messenger code. The Script Parameters are:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipient</td>
<td>List of computers to receive message</td>
<td>A comma-delimited list of the computers that should receive the Windows Message Service message (required).</td>
</tr>
<tr>
<td>Message</td>
<td>Custom Message</td>
<td>A custom message (optional).</td>
</tr>
</tbody>
</table>
Parameters supplied by AppManager

For action scripts, unlike other types of scripts, AppManager adds a number of constants (variables in VBScript action scripts) to the beginning of the script when it is run. The variables have to do with the event that caused the action script to be launched.

You can use these AppManager-added variables in your script, but you cannot see them in any of the Developer’s Console views. You simply have to know that they are there and what they are:

<table>
<thead>
<tr>
<th>AppManager-added constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobID</td>
<td>The JobID of the “calling script” (the script that raised the error that caused the action script to execute).</td>
</tr>
<tr>
<td>Severity</td>
<td>The severity of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>MachineName</td>
<td>The name of the machine running the calling script whose event caused execution of the action script.</td>
</tr>
<tr>
<td>KPName</td>
<td>The Knowledge Script name of the “calling script” (the script that raised the error that caused the action script to execute).</td>
</tr>
<tr>
<td>ObjList</td>
<td>The obj parameter of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>EventMsg</td>
<td>The evtmsg parameter (event message) of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>AgentMsg</td>
<td>The agentmsg parameter (optional long message) of the calling script event that caused execution of the action script.</td>
</tr>
</tbody>
</table>

**Note** These variables are added when the action script is run on either the management server or the managed client.
Functions called in the code

The code calls three types of functions:

- Callback functions, by which the script requests information or action from the AppManager agent running the job. See Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript.”
- Built-in functions of Summit BasicScript. See the BasicScript documentation in appmanager\documentation\development_tools\summit_basicscript\documentation.

Here are the functions called in the script, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventXMLToPlainText</td>
<td>Callback function that converts XML event messages to plain text.</td>
</tr>
<tr>
<td>MCVersion</td>
<td>Callback function that retrieves the version number of the AppManager agent or component where the action is running.</td>
</tr>
<tr>
<td>getItemCount</td>
<td>Summit BasicScript built-in function that returns the number of items in a delimited text string list.</td>
</tr>
<tr>
<td>Item$</td>
<td>Summit BasicScript built-in function that returns a discrete item in a delimited text string list.</td>
</tr>
<tr>
<td>Len</td>
<td>Summit BasicScript built-in function that returns the number of characters in a string.</td>
</tr>
<tr>
<td>Mid</td>
<td>Summit BasicScript built-in function that finds a sub-string within a string.</td>
</tr>
<tr>
<td>GetMachName</td>
<td>Callback function that returns the name of the computer running this script.</td>
</tr>
</tbody>
</table>
Syntax of the Callback functions

Refer to Chapter 11, “AppManager Callbacks for Summit BasicScript and VBScript” for more details.

EventXMLToPlainText

Converts event messages in XML format to plain text (AppManager 5.0 only).

Syntax

EventXMLToPlainText XMLMsg, ProcessedMsg

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLMsg</td>
<td>String</td>
<td>Message in XML format.</td>
</tr>
<tr>
<td>ProcessedMsg</td>
<td>String</td>
<td>The message after translation to plain text. (passed by reference).</td>
</tr>
</tbody>
</table>
EventXMLToPlainText returns 0 for success, -1 for malformed XML, -2 if the message is not XML, or -3 if translation failed for some other reason. Any other value represents failure for an unknown reason.

GetMachName

Returns the machine name (host name) of a managed computer as a string.

Syntax

GetMachName

Parameters

None.

MCNetMessageBufferSend

Sends a message using the Windows Messenger Service. Essentially the same as the Win32 API function NetMessageBufferSend.

Syntax

MCNetMessageBufferSend ("", TargetName, Hostname, Message)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServerName</td>
<td>String</td>
<td>Name of the remote server on which the function is to execute. If this parameter is empty, the local computer is used.</td>
</tr>
<tr>
<td>TargetName</td>
<td>String</td>
<td>Name of computer to which the message is sent.</td>
</tr>
<tr>
<td>Hostname</td>
<td>String</td>
<td>Name of computer from which the message is sent.</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>Message to be sent.</td>
</tr>
</tbody>
</table>

Note The Win32 API function NetMessageBufferSend has a fifth parameter, the length of Message in bytes.
**MCSleep**

Requests the AppManager agent to sleep for an interval during execution of the calling Knowledge Script.

**Syntax**

```
MCSleep intv
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intv</td>
<td>Long</td>
<td>Sleep interval in msec.</td>
</tr>
</tbody>
</table>

Returns 1 when sleep completes, -1 if sleep aborts.

**MCVersion**

Requests the AppManager agent to obtain the version string for the specified component file name.

**Syntax**

```
MCVersion component, verstr [,fullpath]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>String</td>
<td>Component file name.</td>
</tr>
<tr>
<td>verstr</td>
<td>String</td>
<td>The returned corresponding version string (passed by reference).</td>
</tr>
<tr>
<td>fullpath</td>
<td>Bool</td>
<td>If TRUE, component contains the full path to the filename; if FALSE, the component's location is relative to the AppManager\bin directory. By default, this value is FALSE.</td>
</tr>
</tbody>
</table>

**MCVersion** returns nothing.
MSActions

Allows a Knowledge Script to report events and initiate actions.

Syntax

MSActions severity, shortmsg, akpid, objlist, detailmsg
[, detailmsg2, ...., detailmsg6] [,value]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>Long</td>
<td>Severity of the event.</td>
</tr>
<tr>
<td>shortmsg</td>
<td>String</td>
<td>Event message displayed in the List pane.</td>
</tr>
<tr>
<td>akpid</td>
<td>String</td>
<td>Action name or identifier for the action to be taken.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Objects that report the event (their icons will be set to blinking in the Operator Console’s TreeView pane).</td>
</tr>
</tbody>
</table>
| detailmsg | String    | Detail message from the AppManager agent(s) displayed in the event’s Properties dialog. At least one detailmsg is required. The maximum size of the string is 32K. To pass additional information beyond the 32K, you can specify up to 6 message strings, each with a maximum size of 32K, to define the entire detail message for an event. For example, if the message you want to return is 64K, the message would be stored in two strings: MSActions Severity, ”High”, AKPID, ””, detailmsg, detailmsg2
| value     | Double    | Optional. The current value to raise an event.   |

MSAction returns nothing.
The program logic

The Win32 API function `NetMessageBufferSend` is declared at the beginning of the file. This must be done so that it can be called in the code that follows.

\[
\text{Declare Function NetMessageBufferSend Lib "netapi32.dll" } \\
\text{(ByVal pszServer As String, ByVal pszRecipient As String, } \\
\text{ByVal pszSender As String, ByVal pbBuffer As String, } \\
\text{ByVal cbBuffer As Long) As Long}
\]

The function `PreProcessForXML` is at the beginning of the file, right after the declaration of `NetMessageBufferSend`. `PreProcessForXML` is used to convert any custom messages in XML to plain text. This function will be discussed after `Sub Main` is analyzed.

Sub Main

Recall that there are two user-definable Script Parameters used as constants in the script: `Recipient` and `Message`.

- `Recipient` is a comma-delimited string that lists all the computers that should receive the Messenger Service message.
- `Message`, an optional Script Parameter, is a string—if the user does not enter a value, `Message` will be an empty string.

Global constants and variables are:

\[
\text{Const QUO = chr$(34) ' a double quote} \\
\text{Const NL = chr$(10) ' newline} \\
\text{Const MAX_RETRY = 5 ' maximum number of times to retry sending the message} \\
\text{Const V3GSP1 = "3.0.370.0"} \\
\text{Const MIN_MC_VERSION = "4.5"} \\
\text{Dim sAgtVersion$ ' The NetIQmc agent version}
\]

`Sub Main` begins with this code:

\[
\text{Const vbUnicode = 64} \\
\text{Dim lResult As Long} \\
\text{Dim lMsgLen As Long} \\
\text{Dim lRetry As Long}
\]
Dim sTargetName As String
Dim sMessage As String
Dim sHostname As String
Dim sErrMsg As String
Dim bError As Boolean

'Get MC version
sAgtVersion = ""
MCVersion "netiqmc.exe", sAgtVersion

The declared variables are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lResult</td>
<td>Variable to store the value returned by the API function NetMessageBufferSend (or the Callback function MCNetMessageBufferSend).</td>
</tr>
<tr>
<td>lMsgLen</td>
<td>The number of characters in sMessage.</td>
</tr>
<tr>
<td>lRetry</td>
<td>The number of attempts to send sMessage to a given computer.</td>
</tr>
<tr>
<td>sTargetName</td>
<td>Name of a computer to which sMessage should be sent.</td>
</tr>
<tr>
<td>sMessage</td>
<td>The message to be sent by the Windows Messenger Service.</td>
</tr>
<tr>
<td>sHostname</td>
<td>Name of the computer running the script (sending the message).</td>
</tr>
<tr>
<td>sErrMsg</td>
<td>Error message for event if message delivery fails for one or more computers.</td>
</tr>
<tr>
<td>bError</td>
<td>Will become True if message delivery fails for one or more computers.</td>
</tr>
</tbody>
</table>

The Callback function MCVersion is used to get the AppManager agent version number, which is returned as sAgtVersion (passed by reference).

The next section of code prepares for error reporting if the API function NetMessageBufferSend fails. Upon failure, bError will be reset to True and sErrMsg will become the basis for an error message string. This error handling is internal to the script—it has nothing to do with error handling or reporting by Summit BasicScript or the Win32 API.
sErrMsg = "Obj/Err: \\
bError = False

The code now forms the message (sMessage) to be sent to the target computers. The variable Message contains the message supplied by the user, or an empty string if the user chose not to provide a message.

If Message = "" Then
' No message was supplied so use the default message
sMessage = "JobID = " + JobID + NL +  
"KSName = " + KPName + NL +  
"MC MachineName = " + MachineName + NL +  
"Object Name = <" + ObjList + ">" + NL +  
"EventMsg = " + EventMsg + NL + "LongMsg =  
" + PreProcessForXML (AgentMsg) + NL
Else
' Use the messaged that was supplied
sMessage = Message
End If

There are two possible messages that can get sent by the messenger:

1. The contents of user-defined Script Parameter Message.
2. In the case that Message is empty, a “default message” that was created by the action script writer (in the script).

Because of the + NL inclusions in sMessage, the default message is written out line-by-line:

<table>
<thead>
<tr>
<th>Message Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;JobID = &quot; + JobID</td>
<td>The JobID of the “calling script” (the script that raised the error that caused the action script to execute). This is provided by the AppManager infrastructure.</td>
</tr>
<tr>
<td>&quot;KSName = &quot; + KPName</td>
<td>The Knowledge Script name of the “calling script” (the script that raised the error that caused the action script to execute). This is provided by the AppManager infrastructure.</td>
</tr>
<tr>
<td>&quot;MC MachineName = &quot; + MachineName</td>
<td>The name of the computer that is sending the message. This is provided by the AppManager infrastructure.</td>
</tr>
<tr>
<td>&quot;Object Name = &lt;&quot; + ObjList + &quot;&gt;&quot;</td>
<td>The obj parameter of the calling script event that caused execution of the action script.</td>
</tr>
</tbody>
</table>
Now that the message has been determined, the code steps through each computer in the list of recipients (recall that Recipient is a comma-delimited string that lists all the computers that should receive the Messenger Service message). Two built-in functions of Summit BasicScript are used for the loop:

- **ItemCount** returns the number of computers in the list (requires delimiter, in this case a comma, as one input parameter).
- **Item$** returns the name of individual computer number \( I \).

```plaintext
For I = 1 To ItemCount(Recipient, ",")
    sTargetName = Item$(Recipient, I, ",")
    lMsgLen = Len(sMessage)
    lRetry = 0
    'Truncate message if too long
    If lMsgLen > 1024 Then
        lMsgLen = 1024
        sMessage = Mid(sMessage, 1, lMsgLen) & "...
    End If
End For
```

The Windows Messenger service will only accept messages of 1024 characters or less. The Summit BasicScript function **Len** returns the length of sMessage and assigns it to lMsgLen. If lMsgLen is, in fact, greater than 1024, then sMessage will be truncated. The truncation is achieved by using Summit BasicScript function **Mid** to find the substring in sMessage that begins with the first character and is 1024 characters long.

Once sMessage is truncated, we have the exact message we want to send. We are in a loop where we have the name of one of the

---

**Table: Message Line Description**

<table>
<thead>
<tr>
<th>Message Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;EventMsg = &quot; + EventMsg</td>
<td>The evtmsg parameter (event message) of the calling script event that caused execution of the action script.</td>
</tr>
<tr>
<td>&quot;LongMsg=&quot; + PreProcessForXML(AgentMsg)</td>
<td>The agentmsg parameter (optional long message) of the calling script event that caused execution of the action script. <strong>NOTE:</strong> This message, defined by the author of the calling script, may have been written in XML format. Therefore, the <strong>PreProcessForXML</strong> function is used to convert it to plain text, if necessary.</td>
</tr>
</tbody>
</table>
computers on the list of recipients. The next step is to send the message. There is a Callback function (MCNetMessageBufferSend) that can be used to send the message, provided that the AppManager agent version number is later than V3GSP1 (= 3.0.370.0). If the agent is an earlier version, we use the Win32 API function NetMessageBufferSend, which requires that the input strings are in UNICODE. Both NetMessageBufferSend and MCNetMessageBufferSend require the name of the computer sending the message (sHostname).

We are prepared to re-send this message up to MAX_RETRY (= 5 in this code) times.

```basic
sHostname = GetMachName
resend:
  If sAgtVersion < V3GSP1 Then
    lResult = NetMessageBufferSend ( _
      StrConv(,,vbUnicode), _
      StrConv(sTargetName,vbUnicode), _
      StrConv(sHostname,vbUnicode), _
      StrConv(sMessage,vbUnicode), _
      Len(StrConv(sMessage,vbUnicode)))
  Else
    lResult = MCNetMessageBufferSend ("", _
      sTargetName, _
      sHostname, _
      sMessage)
  End If

Both MCNetMessageBufferSend and NetMessageBufferSend return 0 in lResult if they are successful. We test for success. If the function fails, we sleep for 100 milliseconds, go to the resend: label, and try again—up to MAX_RETRY attempts.

```
MAX_RETRY (= 5 in this script) attempts to send have failed for this computer.

If this is the first time we have reached this point, bError will be False and we change it to True (it will remain True for the rest of the script).

If this is not the first time we have reached this point, bError will be True and a comma will be added to sErrMsg.

Then, the name of the destination computer for which the message sending function failed is added to ErrMsg, along with the error number returned by the function.

The For I = 1 To ItemCount(Recipient, ",") loop continues until the list of destination computers is exhausted.

```vbnet
If (bError = True) Then
    sErrMsg = sErrMsg & ",
Else
    bError = True
End If
sErrMsg = sErrMsg & sTargetName & "/" & CStr(lResult)
End If
Next I
```

At this point in the script, if one or more message deliveries fails, bError will be True and sErrMsg will have a comma-separated list of all the target computers for which delivery failed.

If the message delivery has succeeded for all computers on the delivery list, then bError will be False.

When bError is True, two events are raised:

- An "AKP_COMPLETE" event, with the list of unsuccessful deliveries, is sent to complete the action.
- An “AKP_NULL” event is sent to the Operator Console to report a failure of Action_Messenger.

If bError is False, all deliveries have succeeded and we raise only one event. This event reports “Action Complete” as an Event Property because the second parameter of MSActions is an empty string.
If bError=True Then
  MSActions 2, sErrMsg, "AKP_COMPLETE", "", ""
  MSActions 2, "Action_Messenger failed", "AKP_NULL", _
          "", sErrMsg
Else
  MSActions 2, "", "AKP_COMPLETE", "", ""
End If
Exit Sub
End Sub

**Note** Do not confuse “Action Complete” with “AKP_COMPLETE.” The latter signals to AppManager that your script has completed. The former is written to the **Event Properties** dialog box when the script has completed *successfully.*
Function PreProcessForXML

Beginning with AppManager 5.0, Knowledge Script developers can create monitoring script event messages in XML. AppManager will parse these XML messages to create formatted tables in the Message pane of the Event Properties dialog box.

We do not want messages to contain XML tags, so we want some way of stripping these tags.

The Callback function EventXMLToPlainText converts a NetIQ XML-formatted message to plain text. The AppManager agent on the computer running the script containing this function must be version 5.0 or later.

The optional long message for any particular monitoring script may or may not be written in XML. Since we have no way of knowing which it is, we must call EventXMLToPlainText for every such message. The Action_Messenger script includes a function that checks the AppManager agent version number, calls the Callback function EventXMLToPlainText, and handles any errors returned by the Callback.

EventXMLToPlainText(strXMLMsg) takes one parameter, the name of the message text to convert. The function begins by checking that the AppManager version number is greater than or equal to 4.5 (this is the version number for AppManager 5.0). If the version number is less than 4.5, the function simply returns the name of the input text, without attempting conversion, and then exits.

Note The PreProcessForXML function does not obtain the agent version itself. Sub Main obtains a value for the global variable strAgtVersion before it calls PreProcessForXML.

Const MIN_MC_VERSION = "4.5"
Dim sAgtVersion$  ' The NetIQmc agent version

Function PreProcessForXML (sXMLMsg As String) As String

    Dim sProcessedMsg As String
    Dim lRetCode As Long


If (sAgtVersion >= MIN_MC_VERSION) Then

If the version number is sufficient, EventXMLToPlainText is called to convert the input message, strXMLMsg, to plain text output, strProcessedMsg.

Since EventXMLToPlainText can return several different values, a Select block is used to handle the alternatives.

lRetCode = EventXMLToPlainText(sXMLMsg, sProcessedMsg)
Select Case lRetCode
    Case 0
        PreProcessForXML = sProcessedMsg
    Case -1 'Malformed XML Doc
        MSActions 2, "EventXMLToPlainText Failed.", _
                "AKP_COMPLETE", ",", "The XML is a _
                malformed XML document"
        PreProcessForXML = sXMLMsg
    Case -2 'Not event XML Doc
        PreProcessForXML = sXMLMsg
    Case -3 ' Miscellaneous
        MSActions 2, "EventXMLToPlainText Failed.", _
                "AKP_COMPLETE", ",", "XML Translation _
                failed with unknown reason"
        PreProcessForXML = sXMLMsg
    Case Else
        PreProcessForXML = sXMLMsg
End Select

If the agent is an older one, the input string is returned without calling the EventXMLToPlainText function.

Else
    PreProcessForXML = sXMLMsg
End If
End Function
In two cases, -1 and -3, where the input file is XML but could not be converted, `CreateEvent` is used to raise an event and return an error message. In the other cases, where the message is not XML, no event is raised.

**Note** In the event that the optional long message is indeed in XML, but the conversion by `EventXMLToPlainText` fails, the default message will include the unconverted XML message.
The modified script, Action_MessengerEx.qml

In the Action_Messenger script, the message is either a “default” (defined by the script writer, in the script) or a user-supplied message. This is the code to make this choice:

If Message = "" Then
    ' No message was supplied so use the default message
    sMessage = "JobID = " + JobID + NL + _
    "KSName = " + KPName + NL + _
    "MC MachineName = " + MachineName + NL + _
    "Object Name = <" + ObjList + ">" + NL + _
    "EventMsg = " + EventMsg + NL + "LongMsg = _
    " + PreProcessForXML (AgentMsg) + NL
Else
    ' Use the messaged that was supplied
    sMessage = Message
End If

In many cases, a user may want to send both of these messages. The Action_MessengerEx script is a modification of the Action_Messenger script that does this. A new user-definable Script Parameter has been added, PrependMsg. If the user sets this Script Parameter to “y” (default = “n”), that means that the user’s message should be written first, followed by the default message. Then, the code immediately above is altered (new code shown in larger font and bold) like this:

If Message = "" Or PrependMsg = "y" Then
    ' Prepend the custom message to the default message
    sMessage = Message + NL + _
    "JobID = " + JobID + NL + _
    "KSName = " + KPName + NL + _
    "MC MachineName = " + MachineName + NL + _
    "Object Name = <" + ObjList + ">" + NL + _
    "EventMsg = " + EventMsg + NL + "LongMsg = _
    " + PreProcessForXML (AgentMsg) + NL
Else
    ' Use the messaged that was supplied
    sMessage = Message
End If
With this simple modification, the first part of the If block will send both messages if \texttt{PrependMsg} = “y,” irrespective of the value of \texttt{Message}. If \texttt{PrependMsg} = “y” and \texttt{Message} is empty, an empty line will be added before the default message.

The \texttt{Else} block, where only \texttt{Message} is written, will be reached only if \texttt{Message} has a value and \texttt{PrependMsg} = “n.”

\textbf{Note} This script will behave exactly like \texttt{Action_Messenger} if \texttt{PrependMsg} = “n,” except for an extra blank line before the default message.
Chapter 9

Modifying an action script written in Perl

In AppManager, “performing an action” means running an action Knowledge Script as a result of an event being raised in some other type of script.

This chapter describes an action script, Action_UXCommand.qml, that does what its name implies—it runs a non-interactive UNIX command (no user input is allowed) at the command line. This Knowledge Script, written in Perl for UNIX, is similar in function to Action_DOSCommand.qml for Windows.

Both Action_DOSCommand.qml and Action_UXCommand.qml are very powerful Knowledge Scripts, even though they are quite short. You can use them to run entire programs at the command line.

In the last part of this chapter, Action_UXCommand.qml will be extended so that the script can also write to a log file. The modified script is called Action_UXCommandEx.qml.

This chapter covers the following topics:

- Setting up to perform actions
- Invoking actions
- Events without actions
- Ending actions
- Listing of the Action_UXCommand.qml script
- User-set Script Parameters
- Parameters supplied by AppManager
- Functions called in the code
- Syntax of the Callback functions
Developing Custom Knowledge Scripts

The program logic
The modified script, Action_UXCommandEx.qml

Setting up to perform actions

Actions can be defined for “normal” (monitoring and report), discovery, and install scripts. It is not possible to define further actions for action scripts.

Actions for a Knowledge Script can be defined either:

- by the script developer, using the Script Properties dialog box in the Developer’s Console.
- by users of the AppManager Operator Console, using the Properties dialog box that opens when a script is dragged to a target object in the TreeView pane.

Script developers

When developing a monitoring, reporting, or discovery Knowledge Script, you should use the Parameters tab of the Script Properties dialog box in the Developer’s Console to define a Script Parameter with a variable name of $Akpid. You should also give this Script Parameter the default value “AKP_NULL”. You are not forced to do this, but trouble can arise if you do not.

You, the script developer, can also define actions for your script using the Script Properties dialog box. This is hardly ever done by script developers, as it is difficult to predict what type of action a user will want performed. You should define actions rarely, if ever.

Note If you do, in fact, define actions yourself, you might think that setting a default value of “AKP_NULL” for $Akpid is unnecessary. However, a user can undo your choices of actions when setting up a Knowledge Script job, so that a default value will be required in any case.
AppManager Operator Console users

When an AppManager Operator Console user drags a script to a target object in the TreeView pane, the Properties dialog box opens. For every type of Knowledge Script except action scripts, the dialog box will have an Actions tab. In this tab, users can add as many actions as they desire. In the rare event that the script writer associated actions with this script, the user can delete them.

Caution You must choose Action for the Knowledge Script type in the Header tab of the Script Properties dialog box of the Developer's Console. If you fail to make this choice for an action script, it will not be available as a new action to an Operator Console user in the Action tab of the Knowledge Script Properties dialog box:
Invoking actions

It is the responsibility of non-action scripts to invoke actions.

Action scripts are executed only when events are raised. More specifically, when:

- actions have been associated with a monitoring, discovery, install, or reporting Knowledge Script job,
- an event is raised by one of those scripts, and
- the event Callback’s action parameter is set to $Akpid.$

When you are developing a script, you can choose to raise an event that does not call any action scripts that may be chosen by a user. In the NetIQ::Nqext::CreateEvent Callback function that raises the event, you set the action parameter to “AKP_NULL” rather than $Akpid.$

Thus, for any given event, you can choose to have all action scripts executed or none, depending on the value you use for the action parameter. If you set this parameter of NetIQ::Nqext::CreateEvent to $Akpid,$ all actions chosen by a user will be executed. If the parameter is set to “AKP_NULL” no action script will be executed.

**Note** There is no mechanism for you to associate several different actions with a script and choose which one should be executed when a particular event is raised.

Events without actions

In general, you want to generate events without invoking actions when your script detects an error condition that you feel the user should be aware of. For example, if the user enters an invalid script parameter, the script should raise an event, but not invoke an action.

Monitoring scripts should invoke actions only if the conditions or thresholds that the user wants to monitor have been met or exceeded.
Ending actions

It is the responsibility of the action script itself to signal the end of an action.

Toward the end of your action script, your code should signal the completion of the action script by raising an event with the action parameter set to “AKP_COMPLETE.” For example:

NetIQ::Nqext::CreateEvent(25, "", AKP_COMPLETE", $resmsg, 0, "", "", 0, 0);

An event that sets the $Akpid parameter to “AKP_COMPLETE” will cause the Message in the Action tab of the Event Properties dialog box to read:

- “Action Complete” if the event message (second parameter in the event parameter list) is an empty string, as it is in the example immediately above, or
- the event message, if it is not an empty string.

If you do not raise an event with the action parameter set to “AKP_COMPLETE”, the Message in the Action tab of the Event Properties dialog box will continue to read “<Location> Action in Progress,” even though the action has, in fact, completed.

Note Any event raised with an action parameter other than “AKP_COMPLETE” will create a new event.

XML messages

Beginning with AppManager 5.0, you can write custom event messages for your monitoring scripts in XML format. AppManager will parse these XML messages to create formatted tables in the Message pane of the Event Properties dialog box. Here is an example from an event raised by the WebServices_LinkSummary Knowledge Script.
By comparison, the screen below shows an event message that is not in XML format.
The importance of XML messages in this chapter is this: You must take the XML format possibility into account in your action scripts. The event message will be passed to the action script—since this message may be in either plain text or in XML format, the action script will need to take this into consideration. It is not necessary to do this for the sample script in this chapter, but it often arises (see, for example, “Function PreProcessForXML” on page 155).
Listing of the Action_UXCommand.qml script

Here is a listing of the code section of Action_UXCommand.qml. The Script Parameters, included by AppManager as variables, are not shown.

```qml
#main
    use strict;
    use NetIQ::Nqext;
    our $msg;
    our $resmsg = "";

    NetIQ::Nqext::ExecCmd("$Cmd",2);
    $msg = "Action Completed";

    NetIQ::Nqext::CreateEvent(22, $msg,
        "AKP_COMPLETE",
        $resmsg, 0, "",
        "", 0, 0);

#end of main
```

User-set Script Parameters

Users can set action script properties when the “calling script” is dragged and dropped. As an example, assume you drag the UNIX_TopCpuProcs Knowledge Script (the “calling script”) to a target UNIX CPU in the AppManager Operator Console TreeView pane. In the Properties dialog box, you select the Actions tab and add the Action_UXCommand.qml script as your action for UNIX_TopCpuProcs.

After you have chosen Action_UXCommand.qml as your action, you click the Properties button. This opens the Properties dialog box for Action_UXCommand.qml:
Here you must enter a value for the one Script Parameter required by Action_UXCommand.qml:

<table>
<thead>
<tr>
<th>Variable name used in the code</th>
<th>Description the Operator Console user will see</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$cmd</td>
<td>Non-interactive UNIX command</td>
<td>Non-interactive UNIX command (no user input allowed).</td>
</tr>
</tbody>
</table>

### Parameters supplied by AppManager

For action scripts, unlike other types of scripts, AppManager adds a number of variables to the beginning of the script when it is run. The variables have to do with the event that caused the action script to be launched.

You can use these AppManager-added variables in your script, but you cannot see them in any of the Developer's Console views. You simply have to know that they are there and what they are:

<table>
<thead>
<tr>
<th>AppManager-added variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$JobID</td>
<td>The JobID of the &quot;calling script&quot; (the script that raised the error that caused the action script to execute).</td>
</tr>
<tr>
<td>$Severity</td>
<td>The severity of the calling script event that caused execution of the action script.</td>
</tr>
</tbody>
</table>
Functions called in the code

The code calls two AppManager Callback functions, by which the script requests information or action from the AppManager agent running the job.

Here are the Callback functions called in the script, in order of their appearance:

<table>
<thead>
<tr>
<th>Function or subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetIQ::NQEXT::ExecCmd</td>
<td>Callback function that runs a non-interactive command.</td>
</tr>
<tr>
<td>NetIQ::NQEXT::CreateEvent</td>
<td>Callback function that raises an event.</td>
</tr>
</tbody>
</table>

**Note** These variables are added when the action script is run on either the management server or the managed client.
Syntax of the Callback functions

Refer to Chapter 12, “AppManager Callbacks for Perl,” for more details.

**ExecCmd**

The Perl language allows invocation of external commands by using back quotes (``) to substitute the output of the enclosed command. The NetIQ UNIX agent does not support this. Instead, use the `NetIQ::Nqext::ExecCmd` to instruct the agent to execute an external command on behalf of the Knowledge Script.

**Syntax**

`NetIQ::Nqext::ExecCmd (cmd [, flag])`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>String</td>
<td>The non-interactive command.</td>
</tr>
<tr>
<td>flag</td>
<td>Long</td>
<td>Optional. 0: the Callback returns the stdout. 1: the Callback returns the temporary file name containing the stdout. 2: the Callback returns the stdout along with the stderr. 3: the Callback returns the temporary file name containing both the stdout and stderr. Default is 0. <strong>NOTE:</strong> If flag == 1 or 3, then the Knowledge Script must remove the temporary file after it is used.</td>
</tr>
</tbody>
</table>

**Return value**

String. Depending on the flag passed in, this Callback will either return the `stdout` and/or `stderr` results or a filename containing the `stdout`/`stderr` results from executing the external command.

**CreateEvent**

Used by a Knowledge Script to send an event to the AppManager agent. The agent will apply additional rule processing and will determine whether to send a new event or a duplicated (collapsed) event to the AppManager management server.
Syntax

NetIQ::Nqext::CreateEvent(sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [,deletefile])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>The event severity. A value from 1 to 40.</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>The message to be displayed under the Message column in the Events tab.</td>
</tr>
<tr>
<td>akp</td>
<td>String</td>
<td>Name of the action script to launch as a response to this event. You would normally create a Script Parameter and associate it with a variable named $Akpid as part of your script. When the job is dropped and you select an action, the UI will fill in the $Akpid variable with the action name. You will just need to pass in the $Akpid variable to the script.</td>
</tr>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised. This value will determine which object in the TreeView pane to blink. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;UNIX_DiskObject = /mnt/cdrom&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. the machine name.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value to raise the event. This parameter is currently not used. Set to 0.0.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the detail message or a file name that contains the detail message. The detailed message is displayed in the Message tab of the Event Property dialog box. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Reserved for future use. Set to &quot;&quot;.</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Reserved for future use. Set to 0.</td>
</tr>
</tbody>
</table>
The program logic

The **Action_UNIXCommand** is a very simple, yet very powerful Knowledge Script. Using **Action_UNIXCommand**, you can run *anything* that can be run on the command line that does not require user interaction. This could, for example, include Perl script or shell scripts, as long as the computer running **Action_UNIXCommand** can access the Perl or shell script.

**Note** Even though **NetIQ::Nqext::ExecCmd** can return the **stdout** and/or **stderr**, this script contains no code to test whether execution on the command line succeeds. That is because the user will decide what command to run and the script developer does not know what that will be.

The code does just two things:

1. It calls **NetIQ::Nqext::ExecCmd**. The Script Parameter `$cmd` is the user-defined string that is the command to run.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg parameter is a file name or the detailed message itself. If it is a file name, then the contents of the file are read and passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype != 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>

CreateEvent returns nothing.
It calls `NetIQ::Nqext::CreateEvent` to inform the AppManager console that the script has been run (the action is complete).

```perl
#main
use strict;
use NetIQ::Nqext;
our $msg;
our $resmsg = "";

NetIQ::Nqext::ExecCmd("$Cmd",2);
$msg = "Action Completed";

NetIQ::Nqext::CreateEvent(22, $msg,
    "AKP_COMPLETE",
    $resmsg, 0, "",
    "",
    "",
    0, 0);

#end of main
```

This `CreateEvent` call differs from calls in response to an error or failure, where the `evtmsg` parameter (the second parameter) contains an error message. In this case, since the action completed successfully, you should pass in an empty string ("")) instead of an error message. As a result, the user will see "Action Complete" for the action status in the Event Properties window:

![Event Properties Window](image)
Chapter 9  •  Modifying an action script written in Perl  199

The modified script, Action_UXCommandEx.qml

It was noted previously that Action_UXCommand does not provide any error handling, and does not report on the success or failure of the command that is run on the command line. It is impossible to provide for error handling in Action_UXCommand because we have no idea what command a user will choose to execute. Action_UXCommandEx partially compensates for the lack of error handling by writing to a log file. The log file will not tell us if the command line command succeeded, but it will at least tell us why the action script was run.

In Action_UXCommandEx.qml, we add the ability to write to a file. This includes the addition of two new Script Parameters: $LogToFile and $filename. $LogToFile can take the values “y” or “n” (default). When the value is “y”, the script will write to a log file. In this case, $filename, the path and name of the file to be written, must also be provided.

Recall that the running script will include the user-defined Script Parameters as variables with defined values. For example, if the user accepts the defaults, the following will be pre-pended to the script’s code (with the UNIX machine name filled in by AppManager):

#### Begin KSID Section
our $AppManID = "4.5.78.0.8";
our $KSVerID = "1.0";
#### End KSID Section

#### Begin Type Section
our $UNIX_MachineFolder = "";
#### End Type Section

#### Begin KPP Section
our $Cmd="rm /tmp/foo";
our $LogToFile="n";
our $Filename="";
#### End KPP Section

### Begin KPS Section
Following this is the code portion of the Knowledge Script. The changes from Actions_UXCommand.qml are shown in bold and in a larger font.

```perl
#main
use strict;
use NetIQ::Nqext;
our $msg;
our $resmsg = "";
our $file_ok = 0;

# Check to see
if ($LogToFile eq 'y') {
    if ($Filename eq '') {
        NetIQ::Nqext::CreateEvent(22, "No file name was specified", "AKP_COMPLETE", $resmsg, 0, "", "", 0, 0);
    }
    $file_ok = open (LOG, ">>$Filename");
    unless ($file_ok) {
        NetIQ::Nqext::CreateEvent(22, "Failed to open file $Filename for writing.", "AKP_COMPLETE", $resmsg, 0, "", "", 0, 0);
    }
}
NetIQ::Nqext::ExecCmd("$Cmd",2);
$msg = "Action Completed";
if ($file_ok) {
    print LOG "Job ID = $JobID\n";
    print LOG "Severity = $Severity\n";
    print LOG "Object List = $ObjList\n";
```
print LOG "Machine Name = $MachineName\n";
print LOG "KP Name = $KPName\n";
print LOG "Event Msg = $EventMsg\n";
print LOG "Agent Msg = $AgentMsg\n";
print LOG "Command = $Cmd\n";
close (LOG);

NetIQ::Nqext::CreateEvent(22, $msg, "AKP_COMPLETE",
                          $resmsg, 0, "", "", 0, 0);

#end of main
### End KPS Section
Chapter 10

Modifying a report script written in VBScript

This chapter describes how to modify a report script to customize it. All report scripts are exclusively run on Windows computers and are always written in VBScript.

Unlike other types of scripts, report scripts are written with the expectation that they will be customized by the user for a wide variety of different reporting needs. Report scripts are quite complex and contain a very large number of Script Parameters (typically more than 70). Changing the values of these Script Parameters offers a great deal of flexibility. Therefore, modifying a report script will most likely involve changing its value set (Script Parameters) rather than changing its code.

The first part of this chapter shows how to copy a basic report script (ReportAM_AvgValueByDay), change its value set, and rename it as a specialized report script (MyReports_AvgMemByDay).

There are a few situations in which you will need to make minor alterations to the code or to the non-code XML elements of a report script in order to accomplish your goals. In the second part of this chapter, the report script MyReports_AvgMemByDay is modified to report over a time period (MyReports_AvgMemByMonth) that is not available simply by changing the value of Script Parameters.

The following topics are covered in this chapter:

● About report scripts
● Discovering the Report agent
● Altering the value set of an existing script
● Modifying the code of an existing script
About report scripts

Report scripts are similar to other types of Knowledge Scripts (for example, monitoring scripts) insofar as they provide a similar framework for implementing the script. In creating both types of scripts, you can use the Developer Console to define header information, define the type of object on which the script can run, set a default schedule, define Script Parameters for the script, and define actions associated with events raised by the script.

Report scripts differ, though, in the nature of the logic. The logic of a monitoring script is generally geared toward calling the appropriate managed object to extract system or application data from a performance object and then measuring that data against a threshold. The logic of a report script is geared toward getting raw information from a database via a stored procedure, using COM objects to manipulate that information into some meaningful form, and presenting it in a graphical context like a table or chart.

Report scripts employ a number of COM objects that help you retrieve, manipulate, and display data:

- The ADODataSource object is used to connect to a SQL database, query the database, and return a recordset.

- The filter objects (CROSSTAB, HISTOGRAM, STATISTICS, TIMEFILTER, PERCENT) are used to manipulate the data in the recordset returned by ADODataSource, for example, to provide an average hourly value of the data.

- The Report object is used to format the data returned by the filter objects, generate an HTML report, and render the charts in a report.

By employing the filter objects, the manipulation of data has been moved from the stored procedure to the client side of the transaction, freeing up your SQL Server resources.

Specific information about the properties and methods for the COM objects used in report scripts can be found in the
appManager\documentation\development_tools directory on your AppManager CD. There you will find information about the following objects:

- AMChart
- AMLayout
- Report (including ADODataSource and the filter objects)
- NetIQOLE (used to run AppManager from the command line)

**What approach do I take?**

You have two options for modifying a report script:

- Make a copy of an existing script and use the Properties dialog box to set the default Script Parameter values to meet your specific purpose.
- Modify the properties and logic of an existing script.

**Discovering the Report agent**

Before you begin to work with report scripts, make sure that the scripts you want are visible in the Scripts pane of the AppManager Console. There are four groups of report scripts:

- AppManager repository ("ReportAM")
- Analysis Center ("ReportAC")
- Active Directories ("ReportADSI")
- SAP Proxy Server ("ReportSAP")

All users should be able to see the ReportAM Knowledge Script Group. If you are part of an Active Server domain, you should also be able to see the ReportADSI group. To see the ReportAC, you must have installed NetIQ Analysis Center, and to see the ReportSAP group, a SAP Proxy Server must be reporting to your AppManager repository.

Before any of these Knowledge Script Groups will be visible in the Scripts pane of your Operator Console, you must discover them. Do
this by dragging the Discovery_ReportAgent script to your computer's icon in the TreeView pane. The Properties for Discovery_ReportAgent dialog box will open. Select the Values tab and enter “y” for each group you want to discover:

Running the Discovery_ReportAgent script with all values = “y” will discover all the report script groups that are available to you. You will get an event with a message informing you of failure for the groups that are not available.
Altering the value set of an existing script

Say, for example, you want a report to give you the average daily value for physical memory usage by a group of your SQL Servers. (This presumes that you have SQL Servers organized into one or more Server Groups and are using the *NT_MemUtil* script to collect memory usage data from those servers.) You can make a copy of `ReportAM_AvgValueByDay`, and set new default values for the Data source Script Parameters to specify which data is included in the report, and the method by which that data is aggregated.

Making a copy of the script

First, make a copy of `ReportAM_AvgValueByDay`:

1. Open the AppManager Console and select the `ReportAM` tab in the Scripts pane.
2 Right-click ReportAM_AvgValueByDay, and click **Copy Knowledge Script**.

![Copy Knowledge Script dialog](image)

3 Type a new name and description for the script (by default, the new script is named *ReportAM_CopyOfAvgValueByDay*).

<table>
<thead>
<tr>
<th>To</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the script in the same tab as the original</td>
<td>Keep the ReportAM_prefix.</td>
</tr>
<tr>
<td>Create a new tab for your custom report script</td>
<td>Use a new prefix. For example, <strong>MyReports_</strong>. This example uses <strong>MyReports_SQLAvgMemByDay</strong> as the new name for the script.</td>
</tr>
</tbody>
</table>

In this case, rename the script as **MyReports_AvgMemByDay**, and change the description to “Average Memory By Day Report.”

4 Click **OK**.

**Selecting the data streams for the new report**

After you've made a copy of the script, configure it to report on a specific set of data streams:

1 In the Knowledge Script pane, double-click the icon for the new script **MyReports_SQLAvgMemByDay**. The **Properties for MyReports_AvgMemByDay** dialog box will open.
2 Choose the **Values** tab.

3 Under **Data Source**, click the **Browse [...]** button next to the **Select data wizard** Script Parameter.
4 In the Select An AppManager DataSource browser, select the AppManager repository that holds the data for your SQL Server group, and click OK.

5 On the first page of the data wizard, select the Master view and the Server Group filter, and click Next.
6 Select the Server Group for which you want reports, and click Next.

7 Select **By Data Stream**, and click **Next**.
Select NT_MemUtil – MemPhysUsage %, and click Finish.

When you run the script, it will query the AppManager repository for the information in this data stream as it was collected from all computers in your Server Group.

The selections you make using the data wizard are used to provide values for some of the Script Parameters in the stored procedure used by this script.
Selecting the way data is presented in the new report

You select the style of data presentation with the Values tab Properties for MyReports_AvgMemByDay dialog box.

Configure the Select the style Script Parameter to accommodate the way you intend to use the report. For example, if you want a separate page of the report devoted to each computer, use the By computer style. If you want to make a comparison of memory use across all computers in the group, use the All data streams on one page style.

For this example, select All data streams on one page.

The style of presentation you select for the report is used to provide a value for one of the Script Parameters in the stored procedure used by this script.
Selecting the time range for the new report

You select the time range of data presentation with, once again, the Values tab of the Properties for MyReports_AvgMemByDay dialog box.
For this example, we're going to configure the report to include a week's worth of data.

Configure the Select time range Script Parameter to use a Sliding date/time range of 7 Days.

Do not use the End now Script Parameter.

With this configuration, each time the report runs, it will include seven whole days' worth of data (for example, if you run the report each Saturday, it will include data from midnight of the previous Saturday to 11:59:59 P.M. Friday).

The time range setting is used to provide a value for one of the Script Parameters in the stored procedure used by this script.
Selecting days of the week to include in the report

Again, you use the Values tab of the Properties for MyReports_AvgMemByDay dialog box to select the days of the week.

For this example, use the default setting for the Select peak weekday(s) Script Parameter. The default setting includes all seven days of the week.

The settings for this Script Parameter are used to pass information to the TIMEFILTER object via the DayOfWeekFilter property. The TIMEFILTER object helps determine which data returned by the stored procedure is used in the report. The DayOfWeekFilter property is used to specify data from particular weekdays.
Selecting the aggregation interval

You select the aggregation interval with the Values tab of the same Properties for MyReports_AvgMemByDay dialog box (see the preceding graphic). For this example, use the default setting for the Aggregation interval Script Parameter. The default setting is 1 day.

The setting for this Script Parameter is used to pass information to the TIMEFILTER object via the TimePeriodPerPoint property. The TIMEFILTER object helps determine which data returned by the stored procedure is used in the report. The TimePeriodPerPoint property is used to determine the time range by which data is grouped (in this case, 1 day).

Modifying the Report settings and Event Script Parameters

Using the same dialog box, you can set both report settings and event notification Script Parameters. Use the following Script Parameters to include or exclude a parameter help card, table, and chart:

- Include parameter help card?
- Include table?
- Include chart?

Note The “parameter help card” is a table of the script value settings that you can optionally add to your report.
Use the **Select chart style** Script Parameter to set the graphical properties for charts in the report (for example, rotation, series style, and threshold indicator).

Use the **Select output folder** Script Parameter to set the type of output folder (unique, unique with specified prefix, specific name), and the output path.

Use the **Select properties** Script Parameter to set grouping properties and the title and description of the report.

See the *Reporting Guide* for more information about these Script Parameters.

The same dialog box allows you to modify the event notification Script Parameters to raise events associated with generation of the report, and to change event severity levels.

See the *Reporting Guide* for more information about these Script Parameters.

**Saving your new report script**

Once you have defined the Script Parameter settings for your new report, click **OK** to close the **Properties** dialog box and save the settings.

You can now drop the report script and have it generate exactly the report you want without having to configure any of the Script Parameters.
Modifying the code of an existing script

Another method for creating a custom report script is to modify the non-code XML elements or code of an existing script.

In the following example, we'll modify the script we created in the previous example, MyReports_SQLAvgMemByDay, to create a report that gives us an average monthly value for physical memory usage by a group of SQL Servers. The aggregation interval value in the Values tab of the Properties for MyReports_AvgMemByDay dialog box is restricted to days by default. Looking through the various scripts in the ReportAM group, we see that there are report scripts for average values by minute, hour, and day but not for longer intervals. If we want to aggregate data for a week or a month, we will have to modify an existing script.

This new report uses the average daily values collected during a month to figure the month's average, regardless of how many daily averages are collected (for example, if you collect 15 days worth of data one month, and 30 days the next, the monthly averages are based on 15 values and 30 values respectively).

In order to modify your example script from reporting on daily averages to reporting on monthly averages, you need to make minor modifications to several different parts of the script, including:

- The non-code XML elements
- The script properties
- The script logic
- The Values tab in the UI

Modifying the non-code XML elements of the script

The only change that you must make to the non-code XML elements of the script is to change the aggregation interval from days to months. To do this:
1 Open the script MyReports_SQLAvgMemByDay in the Developer's Console.

2 Choose Properties from the View menu.

3 Choose the Parameters tab and highlight the Script Parameter called PRM_TIMEPERIOD.
Click Modify.

In the **Units** field, change Day(s) to Month(s) and then click **Save**.

In the **Script Properties** dialog box, click **OK**.

**Save** the script as **MyReports_SQLAvgMemByMonth**.

**Note** In the procedure we just completed, we changed the basic *units* of a Script Parameter—this is a modification to the script. In the first part of the chapter, we changed the *values* of Script Parameters, but did not alter the script itself.

**Modifying the script properties**

A number of other script properties must be modified to change the text that appears in the report.

1. Open **MyReports_SQLAvgMemByMonth** in the Developer’s Console.
2. Click **View > Properties**.
3 On the Header tab, change the Knowledge Script description to **Average Memory Value By Month Report**.

4 On the Parameters tab, change the Script Parameters description to **Displays the average values by month of physical memory use on SQL Servers**.
5. Select the Script Parameter `PRM_CHARTTITLE`, then click **Modify**.

6. In the **Default value** field, type **Average By Month**, then click **Save**.

7. Select the Script Parameter `PRM_FOLDERDISPLAY`, then click **Modify**.

8. In the **Default value** field, type **AvgValueByMonth And unique folder name**, then click **Save**.

9. Select the Script Parameter `PRM_LAYOUTFOLDER`, then click **Modify**.

10. In the **Default value** field, type **AvgValueByMonth**, then click **Save**.

11. Select the Script Parameter `PRM_FOLDERPREFIX`, then click **Modify**.

12. In the **Default value** field, type **AvgValueByMonth**, then click **Save**.

13. Select the Script Parameter `PRM_INDEXREPORTTITLE`, then click **Modify**.

14. In the **Default value** field, type **SQL Average Memory By Month**, then click **Save**.

15. Select the Script Parameter `PRM_INDEXDESCRIPTION`, then click **Modify**.

16. In the **Default value** field, type **Displays the average value by month of physical memory use on SQL Servers**, then click **Save**.

17. Click **OK** to close the Script Properties dialog box.

**Modifying the code**

After modifying the Script Parameters, you need to make some changes to the VBScript portion of the script.
To properly manipulate the data for this report, you need to create an additional instance of the **STATISTICS** object and the **TIMEFILTER** object.

**Adding variables**

Two new local variables must be added to the main routine.

Local variables are declared just after the main routine is declared:

```vba
Sub Main()
    Dim Detailmsg
    Dim ReportObj
    Dim IncludeType
    Dim CrossTableObj
    Dim StatsFilterObj
    Dim TimeFilterObj
    Dim DataSourceObj
    Dim Displaytype

    Add the following two variables:
    Dim StatsFilterObj2
    Dim TimeFilterObj2
```

**Manipulating data**

The next modification to the script logic involves setting up the filter objects that manipulate the data that is returned by the **ADODataSource** object.

As the script is currently written, the filter objects are set up to return average daily values. You need to make an additional calculation that takes the average daily values for a month and finds their average value. This second calculation is why you created additional variables for the **STATISTICS** and **TIMEFILTER** objects.

Find the following section of the code:

```vba
Set CrossTableObj = CreateObject("NETIQFILTERS.CROSSTAB")
Set StatsFilterObj = CreateObject("NETIQFILTERS.STATISTICS")
Set TimeFilterObj = CreateObject("NETIQFILTERS.TIMEFILTER")

With TimeFilterObj
    .TimePeriodPerPoint  = PRM_TIMEPERIOD * 24 * 3600
```

```vba
End With
```
If (PRM_PEAKHRSTART <> "") And (PRM_PEAKHREND <> "") Then
  .AddTimeOfDayFilter CDate(PRM_PEAKHRSTART),
  CDate(PRM_PEAKHREND)
End If
.SetDaysOfWeekFilter intWDSun, intWDMon, intWDTue, _
  intWDWed, intWDTue, intWDFri, intWDSat
 .PreFilter           = CrossTableObj
End With

with StatsFilterObj
 .Output("AVG")      = True
 .Grouping           = True
 .PreFilter          = TimeFilterObj
End with

This section of the code is used to calculate the average value per day of the data returned in the recordset.

The first three lines create the CROSSTAB, STATISTICS, and TIMEFILTER objects.

The next section of code, from with TimeFilterObj to the first End With statement, prepares the TimeFilter object and its properties and methods to manipulate data passed from the CROSSTAB object.

The TimePeriodPerPoint property defines the time period by which data is grouped. In this case, it’s one day (24 hours x 3600 seconds).

The next few lines are an If statement that checks to see if the daily peak time range option is in effect, and if it is, what the daily time range is.

The next line checks to see which days of the week are included in the report.

The next line identifies the CROSSTAB object as the first filter used on the recordset. The CROSSTAB object changes a row-oriented recordset to a column-oriented recordset. Once the data is reoriented, it is then passed to the TIMEFILTER object.

At this point, the TIMEFILTER object is prepared to give each data point collected during the same day the same time stamp.
The next section of code, from `With StatsFilterObj` to the next `End With` statement, prepares the `STATISTICS` object and its properties to manipulate data passed from the `TIMEFILTER` object.

The `Output` property determines which type of data is included in the report. In this case, it is an average value of the daily values.

The `Grouping` property groups like types of data so that calculations can be performed. For example, the `Grouping` property groups all data labeled June 1, 2002 so that an average can be found for that data, and groups all data labeled June 2, 2002 so that an average can be found for that data.

After the `End With` statement, add the following bit of logic:

```vbnet
With StatsFilterObj2
   .Output("AVG") = True
   .Grouping = True
   .PreFilter = TimeFilterObj2
End With
```

The first new section of code, from `With TimeFilterObj2` to the next `End With` statement, prepares the `TIMEFILTER` object to take the last output of the `STATISTICS` object (average daily values) and aggregate those values by month (give all values for the same month the same time stamp).

The next section, from `With StatsFilterObj2` to the next `End With` statement, prepares the `STATISTICS` object to take the last output of the `TIMEFILTER` object (values aggregated by month) and find an average of each set of monthly values.

The following lines of code implement the filtering of data, and use the last output of the `STATISTICS` object (monthly average values) to create the charts and tables in the report:

```vbnet
If (PRM_DISPLAYTYPE <> "All data streams on one page") Then
```
.Filter = StatsFilterObj
bHasData = .MakeDrillDownReportV1
(DataSourceObj.RecordSet, IncludeType)
Else
    StatsFilterObj.Recordset = DataSourceObj.Recordset
    bHasData = .MakeChartAndTable _
    (StatsFilterObj.Recordset, IncludeType)
End If

Releasing references to the two additional objects you created

Because you created two additional objects for this report
(StatsFilterObj2 and TimeFilterObj2), you will need to release the
references to those objects. Just before the end of the main routine are
the following lines of code that release references to the other objects
used for this report:

Set DataSourceObj = Nothing
Set CrossTableObj = Nothing
Set StatsFilterObj = Nothing
Set TimeFilterObj = Nothing
Set ReportObj = Nothing

Add the following two lines to this section:

Set StatsFilterObj2 = Nothing
Set TimeFilterObj2 = Nothing

Saving the new report script

Once you’ve made the modifications to the script, save it as
MyReports_SQLAvgMemByMonth, and check it in to the AppManager
repository.

Setting a new time range

The last modification you need to make to this script is made through
the user interface:

1  In the Knowledge Script pane of the Operator Console, right-click
MyReports_SQLAvgMemByMonth, then click Properties.
2 In the Select time range Script Parameter, click the **Browse [...]** button.

3 Select **Sliding date/time range**.

4 Set the time range to **3 Months** (or any number of months that suits your reporting needs).

5 Click **Next**, then set the daily time range as needed.

6 Click **Finish**.

7 Click **OK** to close the **Properties** dialog box.

Your new script is now configured to report on the average monthly values for physical memory use by computers on which you are running SQL Server.
Chapter 11

AppManager Callbacks for Summit BasicScript and VBScript

This chapter describes the Callback functions that you can use in AppManager Knowledge Scripts written in either Summit BasicScript or VBScript.

The syntax for calling these functions differs for the two languages. In Summit BasicScript, you simply call the function with the syntax shown in this chapter. In VBScript, you must call the functions via the NQEXT COM object. For example:

- In Summit BasicScript:
  
  ```
  bVar = DynaDataLog(stream_id, legend, value, agentmsg)
  ```

- In VB Script:
  
  ```
  bVar = NQEXT.DynaDataLog(stream_id, legend, value, agentmsg)
  ```

**Note String length limits.** The AppManager agent cannot return a message or data string to the management server that exceeds 2.0MB for AppManager version 4.3 or 5.0MB for AppManager versions 5.0 and later. In addition, Summit BasicScript has an intrinsic string limit of 32 KB.

The following functions are discussed:

- AbortScript
- CreateData
- CreateEvent
- DataHeader
- DataLog
- DynaCollectData
- DynaDataLog
- GetAgentInfo
- GetContextEx
- GetJobID
- GetKPIInterval
- GetMachName
- GetProgID
- GetSecurityContext
- GetTempFileName (VBScript only)
- GetVersion
- Item (VBScript only)
- ItemCount (VBScript only)
- IterationCount
- LongDataHeader
- LongDataLog
- LongDynaDataLog
- MCAbort
- MCEnterCS
- MCExitCS
- MCGetMOID
- MCVersion
- MCWaitForObject (Summit BasicScript only)
- MCWaitForObjectEx (Summit BasicScript only)
- MSActions
- MSLongActions
- NQSleepe
- QTrace
- **WaitForObject**

Most Callback functions can be used in both Summit BasicScript and VBScript. The code examples for these functions are written in Summit BasicScript.
AbortScript

Requests the AppManager agent to abort the current KS execution.

Syntax
AbortScript [objlist, abortmsg, sev [,raise_err]]

Parameters and settings


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objlist</td>
<td>String</td>
<td>Object name.</td>
</tr>
<tr>
<td>abortmsg</td>
<td>String</td>
<td>Abort event detailed message.</td>
</tr>
<tr>
<td>sev</td>
<td>Long</td>
<td>Customized abort event severity. See note under Remarks.</td>
</tr>
<tr>
<td>raise_err</td>
<td>Bool</td>
<td>Optional. When True (default), sets the job status to Error. When False, sets the job status to Stop.</td>
</tr>
</tbody>
</table>

Return value
None.

Remarks
When used by itself without any arguments, the AppManager agent will simply abort the script execution without sending an event. If you specify any of the parameters, you are requesting the AppManager agent to construct and send an event to the AppManager management server.

Note There is an AppManager management server registry setting (“config\MC job abort event Sev”) that overrides any value that you assign to sev, as long as the registry setting is non-zero. “config\MC job abort event Sev” is normally set to 10, and the abort event severity will therefore be 10, no matter what value you give to sev. If “config\MC job abort event Sev” is set to zero, then the registry value will no longer override sev.
Example

This function is used to abort the script when there is an error:

```vbscript
with Err
    'Assemble Error statement
    strErrStatement = "Number: " & CStr(.Number) & ";  
        Description: " & Trim(.Description) & "; Comment: " & _
            Trim(strAddComment) & "; Source: " & .Source

    'Log Error Statement into Error Log File
    resmsg = "REPORT_AGENT = " & REPORT_AGENT
    NQEXT.CreateEvent (PRM_SEVERITYFAIL, PRM_CREATEFAILED, _
        AKPID, resmsg, 0, strErrStatement, "NetIQ AppManager _
        AMAdmin", 1002, 0, False)
    With objrLayout
        .LogMessage strErrStatement
        .HasData  = False
    End With
    NQEXT.AbortScript resmsg, strErrStatement

    'Clear Error Object Properties
    .Clear
End with
```
CreateData behaves the same as DynaDataLog, except that it provides more configuration information for the data header and data points.

**Syntax**

CreateData streamId, legend, dynaleg, objlist, val, agentmsg, msgtype [,schema] [,loglimit] [,lowWM] [,hiWM] [,deletefile]

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>String</td>
<td>The data stream ID. For each unique stream ID in a script, it will generate a Data Source in the AppManager database. Subsequent calls to CreateData using the same stream ID will insert data points to the same Data Source. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>The data stream legend. This value will show up under the Legend column and in the graphs. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>dynaleg</td>
<td>String</td>
<td>The data stream dynamic legend. Contains the dynamic information that can be used for reporting. If a portion of your legend changes often, then pass that text into this parameter. Otherwise leave it blank.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Corresponding object name where the data is collected on. This value is used for graphing and reporting. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;NT_DiskObject = D:&quot;. The objectvalue can normally be obtained by the drop object variable, e.g. NT_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The data point value.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the data detail or a file name that contains the data detail. The data detail is basically an annotation of each data point, giving more information about the data point since the data point is just a numeric value. For example, the data point value may be 5 for the number of processes running, while the data detail may list the processes that are running. The detailed message is displayed in the Graph Data Detail dialog box for each data point. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg is a file name or the detailed message itself. If it is a file name, then the contents of the file are passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Optional. XML schema for dynamic table creation in RDB. Default is an empty string.</td>
</tr>
<tr>
<td>loglimit</td>
<td>Long</td>
<td>Optional. The number of days to keep this data point in the database. Default -1, keep forever. The data points can be removed from the database by other means.</td>
</tr>
<tr>
<td>lowWM</td>
<td>Double</td>
<td>Optional. Low watermark. Default is -1.0.</td>
</tr>
<tr>
<td>hiWM</td>
<td>Double</td>
<td>Optional. High watermark. Default is -1.0.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype != 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
Return value

None.

Remarks

DynaDataLog sends data points for dynamic data streams. This function allows you to collect data for data streams that may be instantiated at each iteration.

Example

Here is an example taken from the `Exchange2000_QueueStatus` Knowledge Script:

```vbs
...  Dim   resname
Const UNITNUMBER = "^^#"
...  Sub Main ()
...  Dim gpocount
Dim detailmsg
...
resname = "NT_GroupPolicyFolder    = " & NT_GroupPolicyFolder
...
retval = OBJ.GetGroupPolicy(computer, gpolist, gpocount, errormsg)
...
detailmsg  = "List of GPO linked to the machine :")
For j = 1 To NQEXT.ItemCount(gpolist,",")
gpo = NQEXT.Item(gpolist,j,",")
If (j < NQEXT.ItemCount(gpolist,",")) Then
detailmsg = detailmsg & Chr(10) & cstr(j) & ") " & 
gpo
End If
Next
...
If (DO_DATA = "y") Then
 NQEXT.CreateData 0, "Group Policy list" & UNITNUMBER, ",",_  resname, gpocount, detailmsg, 0
End If
```
CreateEvent

Used by a Knowledge Script to send an event to the AppManager agent. The AppManager agent will apply additional rule processing and will determine whether to send a new event or a duplicated (collapsed) event to the AppManager management server.

Syntax

CreateEvent sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [,deletefile]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>The event severity. A value from 1 to 40.</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>The message to be displayed under the Message column in the Events tab.</td>
</tr>
<tr>
<td>akp</td>
<td>String</td>
<td>Name of the action script to launch as a response to this event. You would normally create an AKPID parameter as part of your script. When the job is dropped and you select an action, the UI will fill in the AKPID variable with the action name. You will just need to pass in the AKPID variable to the script.</td>
</tr>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised. This value will determine which object in the TreeView pane to blink. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;UNIX_DiskObject = /mnt/cdrom&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. UNIX_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value to raise the event. This parameter is currently not used. Set to 0.0.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the detail message or a file name that contains the detail message. The detailed message is displayed in the Message tab of the Event Property dialog box. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
</tbody>
</table>
Developing Custom Knowledge Scripts

Return value

None.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Not used. Should always be empty.</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Not used. Should always be 0.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg parameter is a file name or the detailed message itself. If it is a file name, then the contents of the file are read and passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype ! = 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
Example

Here is an example taken from the Exchange2000_QueueStatus Knowledge Script:

...  
Severity = 10  
AKPID = AKP_NULL  
Dim resname  
...  
Sub Main ()  
...  
Dim shortmsg  
...  
Dim detailmsg1  
...  
resname = "NT_GroupPolicyFolder = " & NT_GroupPolicyFolder  
...  
shortmsg = "Number of Group Policies : " & cstr(gpocount) & " exceeds threshold."  
If (DO_EVENT = "y") And (gpocount > Threshold) Then  
detailmsg1 = "Total number of Group Policies associate with the machine = " & cstr(gpocount) & Chr(10) & "Threshold of number of GPOs = " & Threshold & Chr(10) & detailmsg  
    NQEXT.CreateEvent Severity, shortmsg, AKPID, resname, _  
        0.0, detailmsg1, ",", 1000, 0  
End If
**DataHeader**

Sends the data header for logging and graphing data streams (short form). A `DataHeader` call is made in the first execution interval of a job for each data stream to be collected. Each data header provides an appropriate description of the information collected in the data stream. Most Knowledge Scripts that collect data include this call.

**Syntax**

```
DataHeader legend, graph_id, stream_id [,objlist]
```

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>legend</td>
<td>String</td>
<td>Graphing legend displayed in the List and Graph panes. For example, the legend for one data stream created by <code>NT_CpuResource</code> is <code>user CPU</code>. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>graph_id</td>
<td>Long</td>
<td>Graph ID. This parameter is not currently used. It is always set to the value 0 (see example).</td>
</tr>
<tr>
<td>stream_id</td>
<td>Long or String</td>
<td>Data stream identifier. This identifier should be unique for each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Optional. Matching object where the data is collected.</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Remarks**

The data stream identifier `stream_id` is used to link `DataLog` calls for individual data points to the appropriate `DataHeader` that describes the data stream. The `stream_id` parameter should be unique for each
data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.

To allow your custom Knowledge Scripts to collect data, you need to include calls to both `DataHeader` and `DataLog`.

The `DataHeader` function initiates the collection of a data stream. You must include this function call once for each data stream collected by each job before sending any data points. Therefore, you should add the `DataHeader` call so that it runs in the first execution interval.

Once a data stream is initiated with the `DataHeader` call, the `DataLog` function sends the actual data point value back to the management server at each interval. The `DataLog` function needs to be called for each data stream being collected. Each `DataLog` call is associated with one `DataHeader` call through the same `streamid`.

**Example**

In the `NT_LogicalDiskSpace` script, the `DataHeader` call initiates the collection of two data streams (used percentage and available MB) for each logical disk when the user elects to collect data (`DO_DATA = "y"`):

```vbnet
If IterationCount() = 1 And DO_DATA = "y" Then
    DataHeader "Ldsk: & objname & "USED%", 0, I
    DataHeader "Ldsk: & objname & "AVAIL MB", 0, I+1000
End If
```

Once `DataHeader` is used to establish the data stream, the `DataLog` call is used to collect a data point value for each data stream at each interval the job is run:

```vbnet
If DO_DATA = "y" Then
    . . .
    DataLog I, Dutil, datapoint
    DataLog I+1000, Dfree, datapoint
    . . .
End If
```
**DataLog**

Sends data points back for logging and graphing. Most Knowledge Scripts that collect data include this call. This call is always used in conjunction with a `DataHeader` call.

**Syntax**

```
DataLog stream_id, data, datapointmsg
```

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream_id</code></td>
<td>Long or String</td>
<td>Data stream identifier. This identifier should be the same identifier used in the associated DataHeader call for each data stream. The string length limit is 64 characters.</td>
</tr>
<tr>
<td><code>data</code></td>
<td>Double</td>
<td>Data point value.</td>
</tr>
<tr>
<td><code>datapointmsg</code></td>
<td>String</td>
<td>Detail message from the AppManager agent(s) displayed in the Graph Data Detail dialog. The maximum size for this string is 32K.</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Remarks**

The data stream identifier `stream_id` is used to link `DataLog` calls for individual data point collection to their associated `DataHeader` calls. The `stream_id` parameter should be unique for each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.
Example

This code sample shows three data streams sent via the **DataLog** routine. Each has a separate message consisting of the data stream ID (see **DataHeader**), the data value and a message (defined in a separate routine):

```vbscript
If DO_DATA = "y" Then
  Dim Msg0$  
  Dim Msg1$  
  Dim Msg2$  
  Msg0 = OBJ.PhysUsageAgtMsg(True)  
  Msg1 = OBJ.VirtualUsageAgtMsg(True)  
  Msg2 = OBJ.PGFileUsageAgtMsg(True)  
  DataLog 0, Dval0, Msg0  
  DataLog 1, Dval1, Msg1  
  DataLog 2, Dval2, Msg2
End If
```
DynaCollectData

This function works the same as “DynaDataLog” on page 246 except that it provides more parameters to specify configuration information for the data header and data point.

Syntax


Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamId</td>
<td>Long, string</td>
<td>Data stream ID. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>Legend</td>
<td>String</td>
<td>Data stream legend. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>Dynaleg</td>
<td>String</td>
<td>Dynamic legend, contain the dynamic information that can be used for reporting.</td>
</tr>
<tr>
<td>Objlist</td>
<td>String</td>
<td>Corresponding object name where the data is collected.</td>
</tr>
<tr>
<td>Val</td>
<td>Double</td>
<td>Current data point value.</td>
</tr>
<tr>
<td>Agentmsg</td>
<td>String</td>
<td>Contain either a plain text or a message file name.</td>
</tr>
<tr>
<td>Msgtype</td>
<td>Long</td>
<td>Related to agentmsg, 0 for plain text, 1 for message file.</td>
</tr>
<tr>
<td>Schema</td>
<td>String</td>
<td>Optional. This parameter should not be used. Default is an empty string.</td>
</tr>
<tr>
<td>Loglimit</td>
<td>Long</td>
<td>Optional. Datalog limit in # of days. Default is -1.</td>
</tr>
<tr>
<td>LowWM</td>
<td>Double</td>
<td>Optional. Low watermark. Default is -1.0.</td>
</tr>
<tr>
<td>HiWM</td>
<td>Double</td>
<td>Optional. High watermark. Default is -1.0.</td>
</tr>
<tr>
<td>Deletefile</td>
<td>Bool</td>
<td>Optional. Used only when msgtype=1, default is True.</td>
</tr>
<tr>
<td>LogOnHeaderCreate</td>
<td>Bool</td>
<td>Optional. If omitted, defaults to True. If True, data point logged when data header created.</td>
</tr>
</tbody>
</table>
**Return value**

Boolean. **True** if the data point is returned successfully, **False** otherwise.

**Remarks**

The data stream identifier `stream_id` can be a numeric identifier or a data stream name. The `stream_id` parameter should be unique for each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.

**Example**

The following code fragment is from **Oracle_CallsPerTransaction**:

```vbnet
If DO_DATA = "y" Then
    DynaCollectData (TargetObj(lIndex).m_sDBName & _
    TargetObj(lIndex).m_sVer ,"Calls Per Transaction " & _
    TargetObj(lIndex).m_sDBName & "@" & _
    TargetObj(lIndex).m_sVer , _, _
    ,"",sResName,dblResult,"",0
End If
```
**DynaDataLog**

Sends data points for dynamic data streams. This function allows you to collect data for data streams that may be instantiated at each iteration. For example, the Knowledge Script `Exchange_MTAQueueLen` can dynamically enumerate new Exchange connectors at each interval. This extension creates a data stream for each connection and continues to collect data for each stream by stream name.

**Syntax**

```
DynaDataLog stream_id, legend, value, agentmsg [, objlist]
```

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream_id</td>
<td>Long or String</td>
<td>Data stream identifier. The identifier can be a numeric identifier or a stream name. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>Graph legend displayed in the List and Graph panes. For example, the legend for one data stream created by <code>nt_cpuresource</code> is User CPU. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Data point value.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Detail message from the AppManager agent(s) displayed in the Graph Data Detail dialog. The maximum size for this string is 32K.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Optional. Matching object where the data is collected.</td>
</tr>
</tbody>
</table>

**Return value**

Boolean. `True` if the data point is returned successfully, `False` otherwise.
Remarks

The data stream identifier `stream_id` can be a numeric identifier or a data stream name. The `stream_id` parameter should be unique for each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.

Note Unlike the `DataLog` extension, `DynaDataLog` does not require a `DataHeader` call to establish a data stream.

Example

This code fragment illustrates the use of `DynaDataLog` to return data for dynamically discovered instances of MTA connectors:

```vbscript
If DO_DATA = "y" Then
    AgtMsg = AgtMsg & dval & chr$(9) & chr$(9) & Inst & Chr$(10)
    rc = DynaDataLog (Inst & " QueueLen", Inst & " QueueLen", _
        dval, dval & chr$(9) & chr$(9) & Inst & Chr$(10) )
End If
```

In VB Script, these lines are modified to call the function through the `NQEXT` object:

```vbscript
If DO_DATA = "y" Then
    AgtMsg = AgtMsg & dval & chr(9) & chr(9) & Inst & Chr(10)
    rc = NQEXT.DynaDataLog (Inst & " QueueLen", Inst & " QueueLen", _
        dval, dval & chr(9) & chr(9) & Inst & Chr(10) )
End If
```
GetAgentInfo

This function will provide the current AppManager agent information to the script, including the framework type, the agent version, and the path of the installation directory.

Syntax
GetAgentInfo prodtype, agtver, installdir

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prodtype</td>
<td>String</td>
<td>The returned framework type, either &quot;OM&quot; or &quot;AM&quot;</td>
</tr>
<tr>
<td>agtver</td>
<td>String</td>
<td>The returned agent version string</td>
</tr>
<tr>
<td>installdir</td>
<td>String</td>
<td>The returned product install path</td>
</tr>
</tbody>
</table>

Return value

None.

Example

Here is an example taken from the AMAdmin_DeleteGlobalParams Knowledge Script:

```vbs
' Check that the agent supports server-side job configuration
Sub AssertAgentVersion()
    Dim prodType As String
    Dim agentVer As String
    Dim installDir As String

    GetAgentInfo prodType, agentVer, installDir

    If agentVer < MIN_MC_VERSION Then
        MCAbort "", "This MC does not support server-side _
                   job configuration.", 10, True, False
    End If
End Sub
```

248  Developing Custom Knowledge Scripts
GetContextEx

Returns the value for a specified custom context. This function can be used to get custom properties you have stored as a name-value pair in the repository using the AppManager Security Manager. Only one custom name-value property pair can be entered in a single GetContextEx call.

Syntax

GetContextEx label_name, label_value,
    sublabel_name, sublabel_value,
    val1_name, val1_value

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label_name</td>
<td>String</td>
<td>Label context name (can not be empty). Currently, the predefined name is label. This parameter must be set to &quot;label&quot; in your script.</td>
</tr>
<tr>
<td>label_value</td>
<td>String</td>
<td>Label context value stored in the AppManager repository (can not be empty). Enter the Label value exactly as it has been entered in the Label field in the Security Manager.</td>
</tr>
<tr>
<td>sublabel_name</td>
<td>String</td>
<td>Sub-label context name (can not be empty). Currently, the predefined name is sub-label. This parameter must be set to &quot;sub-label&quot; in your script.</td>
</tr>
<tr>
<td>sublabel_value</td>
<td>String</td>
<td>Custom Sub-label context value stored in the AppManager repository (can not be empty). Enter the Sub-Label value exactly as it has been entered in the Sub-Label field in the Security Manager.</td>
</tr>
<tr>
<td>val1_name</td>
<td>String</td>
<td>Value 1 context name. Currently, the predefined name is val1. This parameter must be set to &quot;val1&quot; in your script.</td>
</tr>
<tr>
<td>val1_value</td>
<td>String</td>
<td>Context property value stored in the AppManager repository. Use an empty string (&quot;&quot;) to return the current value. When this parameter is not an empty string, it is used as a filter to further qualify the output context value.</td>
</tr>
</tbody>
</table>
Return value

Variant. Returns the context value.

Remarks

This function allows you to retrieve custom information you have stored in the AppManager repository using the Security Manager. Within your Knowledge Scripts, the `GetContextEx()` function uses the label and sub-label you enter to locate the appropriate custom context value. For information about entering the custom values into the repository, see the AppManager User Guide.

Example

Assume the following information has been entered using the Security Manager:

- **Label:** MyApplication
- **Sub-Label:** email_address
- **Value 1:** admin@tgif.com

To use the custom values:

- Modify the `NeedKPW` parameter in the KP-Status section of the customized Knowledge Script to look up a value in the KPW table. For example:
  
  ```
  'NeedKPW = 1       ' Look up the value in the KPW table
  ```

- Add the `GetContextEx()` function to look up valid custom values in the repository. For example:

  ```
  GetContextEx("label", "MyApplication", "sub-label", "email_address", "val1", email)
  ```
The following code fragment illustrates how to use this call in
BasicScript:
Sub Main()
Dim email As String
email = ""
'Get the current value
If AdminEmail = "" Then
dreturn = GetContextEx("label", "MyApplication", _
"sub-label", "email_address", "val1", email)
Else
email = AdminEmail
End If
End Sub

If the user doesn’t specify an email address in the Knowledge Script
Properties dialog, the GetContextEx call looks up the email address
stored in the repository and the email parameter returns the value
"admin@tgif.com".

Chapter 11 • AppManager Callbacks for Summit BasicScript and VBScript

251


GetJobID

Gets the job ID for the running Knowledge Script.

Syntax

GetJobID

Parameters and settings

None.

Return value

Long.

Example

The following BasicScript code fragment comes from the NT_ServiceDown Knowledge Script:

```basic
...  
JobId = GetJobId       'Get Job Id for ntserdown.ini file  
For I = 1 To NumServ
    Servname = Item$(RealServices, I,, ",")  
    RegSrv Servname, JobId, I  'RegSrv registers each service  
        'in ntserdown.ini  
Next I
...  
```
GetKPInterval

Returns the execution interval, in seconds, for the running Knowledge Script.

Syntax
GetKPInterval

Parameters and settings
None.

Return value
The execution interval, in seconds.

Example

Here is an example taken from the Domino_UserSessions Knowledge Script:

```vbscript
Dim Time_Out As Long
Time_Out = TIMEOUT
If (Time_Out > GetKPInterval()) Then
    resmsg = "MC Abort"
    longmsg = "Please enter a timeout that is less than KP interval"
    longmsg = longmsg & Chr$(13) & Chr$(10)
    longmsg = longmsg & "Current condition: Timeout " & CStr(Time_Out) & " > KP Interval " & CStr(GetKPInterval()) & " \n    MCAbort resmsg, longmsg
End If
```

GetMachName

Returns a managed computer machine name (host name). This is useful for including the name of the computer causing an event in a message.

Syntax
GetMachName

Parameters
None.

Returns
Name of managed computer.

Example
The following BasicScript code fragment comes from the \texttt{NT\_RemoteServiceDown} Knowledge Script:
\begin{verbatim}
If IterationCount() = 1 Then
    NumServ = ItemCount(Services, ",,"
    If NumServ = 0 Then
        Err.Description = "No Service is given"
        Err.raise 4002
    End If
    NumMach = ItemCount(MachineList, ",,"
    If NumMach = 0 Then
        Machines = GetMachName()
        NumMach = 1
    Else
        Machines = MachineList
    End If
End If
\end{verbatim}
GetProgID

Requests the AppManager agent to return the versioned Prog ID that matches the version of the current script. For example, a version 4.0 KS may call this function to construct and return the version 4.0 prog ID for the NT MO, such as "NetIQAgent.NT.4"

Syntax

GetProgID progid, scriptver

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>progid</td>
<td>String</td>
<td>Version independent MO COM progid</td>
</tr>
<tr>
<td>scriptver</td>
<td>String</td>
<td>The associated KS script version string</td>
</tr>
</tbody>
</table>

Return value

LPTSTR. The versioned Prog ID.

Example

Here is an example taken from the Win2000_GroupPolicyCount Knowledge Script:

...<Version>
  <AppManID>4.5.78.0.</AppManID>
  <KSVerID>1.1</KSVerID>
</Version>

If NQEXT.IterationCount() = 1 Then
  progid = NQEXT.GetProgId("NetIQAgent.NT", AppManID)
  Set NT = CreateObject(progid)
  Set OBJ = NT.GroupPolicy
End If
GetSecurityContext

Return the value for the specified KPW.

Syntax

GetSecurityContext label_val, sublabel_val, name, value

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label_val</td>
<td>String</td>
<td>Label value</td>
</tr>
<tr>
<td>sublabel_val</td>
<td>String</td>
<td>Sub-label value</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Value name</td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>The returned value for the specified name</td>
</tr>
</tbody>
</table>

Return value

Boolean. True if the operation succeeds, False otherwise.
**GetTempFileName (VBScript only)**

Requests the AppManager agent to construct a temp file name based on the input criteria. The file name is concatenated from path, prefix string, and a hex string formed from the unique ID, and a ".tmp" extension.

**Syntax**

GetTempFileName path, prefix, uniqid

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>String</td>
<td>Temp file path</td>
</tr>
<tr>
<td>prefix</td>
<td>String</td>
<td>File name prefix string</td>
</tr>
<tr>
<td>uniqid</td>
<td>Long</td>
<td>An unique ID</td>
</tr>
</tbody>
</table>

**Return value**

LPTSTR. The constructed temp file name and path.
GetVersion

Asks the AppManager agent to obtain the version string for the specified file name. It can be used to retrieve the version of a NetIQ file or any file on the system. If file is simply a file name (with no path specified), the AppManager agent will pre-append the NetIQ install path.

Syntax

GetVersion file, verstr

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>String</td>
<td>NetIQ or any file name</td>
</tr>
<tr>
<td>verstr</td>
<td>String*</td>
<td>The returned corresponding version string (passed by reference).</td>
</tr>
</tbody>
</table>

Return value

None.

Remarks

When looking up the version, the AppManager agent will automatically perform the wildcard operation and retrieve the latest version. For example, if the specified component is qsqla.dll and there exists both qsqla3.dll and qsqla4.dll, MC will return the version of qsqla4.dll.
Example

Here is an example taken from the IIS\_CacheHitRatio Knowledge Script:

```
Dim NT As Object
Dim OBJ As Object
Dim IIS As Object
Private IISVersion As Long
Private Counter As String ' delay counter value until version ' is found
...
Sub Main()
  Dim progid As String
  Dim IISprogid As String
  ...  
  progid = MyGetProgId("NetiQAgent.NT")
  Set NT = CreateObject(progid)
  Set OBJ = NT.System
  IISprogid = MyGetProgID("NetiQAgent.IIS")
  Set IIS = CreateObject(IISprogid)
  IISVersion = IIS.GetVersion()
  ...  
  If (IISVersion < 5) Then
    Counter = "Cache Hits %"
  Else
    Counter = "URI Cache Hits %"
```
**Item (VBScript only)**

Executes the Summit BasicScript built-in function Item.

**Syntax**

Item list, idx, delim

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Long</td>
<td>A list of strings.</td>
</tr>
<tr>
<td>idx</td>
<td>Long</td>
<td>Returned string index within the list.</td>
</tr>
<tr>
<td>delim</td>
<td>String</td>
<td>String delimiter.</td>
</tr>
</tbody>
</table>

**Return value**

Returns the string at the specified index position within the input list.
Example

Here is an example taken from the WIN2000_GroupPolicyCount Knowledge Script:

```
<Script language="VBScript">

Dim NT
Dim OBJ' Keep the reference count of this DLL
...
Sub Main ()
Dim gpolist
Dim computer
Dim errormsg
Dim gpocount
Dim progid
Dim j
Dim detailmsg
Dim retval
...
progid = NQEXT.GetProgId ("NetiQAgent.NT", AppManID)
Set NT = CreateObject (progid)
Set OBJ = NT.GroupPolicy
...
computer = ""
retval = OBJ.GetGroupPolicy(computer, gpolist, _
gpocount, errormsg)
...
detailmsg = "List of GPO linked to the machine:"
For j = 1 To NQEXT.ItemCount(gpolist,"") 'There is an
' extra comma after the list
  gpo = NQEXT.Item(gpolist,j,"")
  If (j < NQEXT.ItemCount(gpolist,"")) Then
    detailmsg = detailmsg & Chr(10) & cstr(j) & ") " _
    & gpo
  End If
Next
```
**ItemCount (VBScript only)**

Executes the Summit BasicScript built-in function `ItemCount`.

**Syntax**

`ItemCount list, delim`

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Long</td>
<td>A list of strings.</td>
</tr>
<tr>
<td>delim</td>
<td>String</td>
<td>String delimiter.</td>
</tr>
</tbody>
</table>

**Return value**

Long. Number of strings in the input list.
Example

Here is an example taken from the WIN2000_GroupPolicyCount Knowledge Script:

```vbscript
<Script language="VBScript">
...
Dim NT
Dim OBJ' Keep the reference count of this DLL ...
Sub Main ()
Dim gpolist
Dim computer
Dim errormsg
Dim gpocount
Dim progid
Dim j
Dim detailmsg
Dim retval
...
progid = NQEXT.GetProgId ("NetiQAgent.NT", AppManID)
Set NT = CreateObject (progid)
Set OBJ = NT.GroupPolicy ...
computer = ""
retval = OBJ.GetGroupPolicy(computer, gpolist, _
gpocount, errormsg)
...
detailmsg = "List of GPO linked to the machine:")
For j = 1 To NQEXT.ItemCount(gpolist,"") ' List ends ' with extra comma
gpo = NQEXT.Item(gpolist,j,"")
If (j < NQEXT.ItemCount(gpolist,"")) Then
detailmsg = detailmsg & chr(10) & cstr(j) _
& ") " & gpo
End If
Next
```

...
IterationCount

Determines the number of times the calling Knowledge Script has run. Most Knowledge Scripts that collect data include this call.

Syntax
IterationCount

Parameters and settings
None.

Return value
Long representing the current iteration count.

Example
This routine is called to check the current iteration count for a Knowledge Script. The first time a Knowledge Script runs, the iteration count is 1. If the iteration count is 1, the DataHeader call is made to allow the Knowledge Script to collect a data stream.

The following BasicScript example from NT_CpuResource illustrates this function with four DataHeader calls (four data streams):

```basic
If IterationCount() = 1 Then
    If DO_DATA = "y" Then
        DataHeader "USER Cpu" & UNITPERCENT, 0, 0
        DataHeader "Number of Processes" & UNITNUMBER, 0, 1
        DataHeader "All Threads" & UNITNUMBER, 0, 2
        DataHeader "Interrupts" & UNITNUMBER, 0, 3
    End If
End If
```

End If
LongDataHeader

Requests the AppManager agent to send a data header to the AppManager management server. You can use this function to specify all the listed configuration information, such as high/low watermark, etc.

Syntax

LongDataHeader legend, graphId, streamId, logLimit, color, style, MaxVal, minVal, hiWM, loWM [,objlist]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>legend</td>
<td>String</td>
<td>Legend name. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>graphId</td>
<td>Long</td>
<td>Graph ID.</td>
</tr>
<tr>
<td>streamId</td>
<td>Variant</td>
<td>Data stream ID. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>logLimit</td>
<td>Long</td>
<td>Datalog limit in # of days.</td>
</tr>
<tr>
<td>color</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td>style</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td>MaxVal</td>
<td>Double</td>
<td>Maximum allowed data point value.</td>
</tr>
<tr>
<td>minVal</td>
<td>Double</td>
<td>Minimum allowed data point value.</td>
</tr>
<tr>
<td>hiWM</td>
<td>Double</td>
<td>High watermark.</td>
</tr>
<tr>
<td>loWM</td>
<td>Double</td>
<td>Low watermark.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Optional. Matching object where the data is collected.</td>
</tr>
</tbody>
</table>

Return value

None.
Remarks

The previous Callback function, DataHeader, just uses the default values for these configurations.

Example

Here is an example taken from the NT_CpuLoaded.qml:

```plaintext
Dim NT As Object
Dim OBJ As Object
Dim resmsg$
Dim resarg$
...
Sub Main()
  Dim Duser#, Dpriv#, Dtotal#
  Dim progid$
  ...
  progid = MyGetProgId ("NetIQAgent.NT")
  Set NT = CreateObject(progid)
  Set OBJ = NT.CPU
  ...
  resarg = ""
  ...
  Dpriv = OBJ.UtilValue("PRIVILEGED", resarg)
  Duser = OBJ.UtilValue("USER", resarg)
  If Dpriv = -1 Or Duser = -1 Then
    Err.Description = "Failed on CPU MO."
    Err.raise 4101 'raise error to terminate this KS
  End If
  ...
  Dtotal = Dpriv + Duser
  ...
  If DO_DATA = "y" Then
    If IterationCount() = 1 Then
      LongDataHeader OBJ.UtilLegend("PROCESSOR", resarg),
                      0, 0, CpuUtil_DataPoints, 0, 0, 0, 0, 0
    End If
    ...
    longm = "Privileged " & Format$(Dpriv, "0.00") & chr$(10) & "User " & Format$(Duser, "0.00") & chr$(10) & "Total " & Format$(Dtotal, "0.00")
    DataLog 0, Dtotal, longm
  End If
End Sub
```
LongDataLog

This function requests the AppManager agent to send a data point to management server. Unlike “DataLog” on page 242, the AppManager agent will read the detail message from the specified file and return the message as part of data point. This function is useful for returning detail messages larger than 32 KB.

Syntax

LongDataLog streamId, value, msgfile [,deletefile]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>Long, string</td>
<td>Data stream ID. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Current data point value.</td>
</tr>
<tr>
<td>msgfile</td>
<td>String</td>
<td>The name of file that contains the detail message for the current data point.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Optional. True to delete the file after it is read. False to retain.</td>
</tr>
</tbody>
</table>

Return value

None.

Example

This code fragment from the AD_NumberOfComputers Knowledge Script reads the detail message from strOutFile:

```vbscript
... If DO_DATA = "y" Then
    LongDataLog streamid, NumUsers, strOutFile
End If
...```

LongDynaDataLog

This function works the same as “DynaDataLog” on page 246 except the AppManager agent reads the detail message from a specified file and returns the message as part of data point. This function is useful for returning detail messages larger than 32 KB.

Note **LongDynaDataLog** does not require a DataHeader call to establish a data stream.

Syntax

`LongDynaDataLog streamId, legend, value, msgfile [,deletefile] [,objlist]`

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>Long, string</td>
<td>Data stream ID. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>Legend name. The string length limit is 128 characters.</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Data point value.</td>
</tr>
<tr>
<td>msgfile</td>
<td>String</td>
<td>The name of file that contains the detail message for the current data point.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Bool</td>
<td>Optional. True to delete the file after it is read. False to retain.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Optional. Matching object where data is collected on.</td>
</tr>
</tbody>
</table>

Return value

None.

Remarks

The data stream identifier stream_id can be a numeric identifier or a data stream name. The stream_id parameter should be unique for
each data stream collected by a single Knowledge Script. The identifier does not need to be unique across Knowledge Scripts.

**Example**

This code fragment from `Win2000_DiskQuotaStatus` constructs the stream ID and reads the detail message from `overlimitF`:

```vbscript
... If DO_DATA_OVERLIMIT = "y" Then
    brc = LongDynaDataLog(MachName & "/" & drive & "/ _
                          #overlimit", "Number users on " & drive & " _
                          over quota limit" & UNIT, overlimit, overlimitF _
                          & "2")
End If
...```

...
**MCAbort**

Allows a Knowledge Script to abort its current operation. When invoked, a severity 40 event is raised on the resource objects specified and the Knowledge Script job is signaled to terminate the current operation. The AppManager agent and other jobs are not affected.

**Syntax**

\[
\text{MCAbort objlist, agentmsg [,sev] [,toretevt] [,raise_err]}
\]

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objlist</td>
<td>String</td>
<td>Objects that report the event (represented by icons in the Operator Console's TreeView pane).</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Message to accompany the aborted operation event.</td>
</tr>
<tr>
<td>sev</td>
<td>Long</td>
<td>Optional. Specifies a custom event severity.</td>
</tr>
<tr>
<td>toretevt</td>
<td>Bool</td>
<td>Optional. True to generate an event, False to not generate an event.</td>
</tr>
<tr>
<td>raise_err</td>
<td>Bool</td>
<td>Optional. Sets the job status to error or stop. True to set job status to Error, False to set it to Stop. Default is True.</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Example**

The following BasicScript code fragment is from the **General_AsciiLog** Knowledge Script:

\[
\ldots
\text{If ErrorCode = -1 Then}
\]

\[
\text{MCAbort resname, ErrorMsg}
\]

\[
\text{End If}
\]

\[
\text{If ErrorCode = -2 Then}
\]

\[
\ldots
\]

Developing Custom Knowledge Scripts
**MCEnterCS**

Enter the AppManager agent-defined **critical section**.

**Syntax**

\[ \text{MCEnterCS} \]

**Parameters and settings**

None.

**Return value**

Long. Returns 0 if the lock was acquired; 1 if the job was stopped while waiting for the job (in which case, the job did not acquire the lock).

**Example**

The following BasicScript code fragment comes from **ARCserve_CanceledJobs**:

```basic
If IterationCount() = 1 Then
  Dim i%
  On Error GoTo ErrorOut
  proid = MyGetProgId("NetIQAgent.ARCserve")
  Set ASMO = CreateObject(proid)
  ... Filter=0
  If DO_ERR="y" Then Filter=1
  If DO_WARN="y" Then Filter=Filter+2
  MCEnterCS
  ErrorCode=ASMO.PreProcess(FileName, ErrorMsg, _
    StartOffset, FileCreateTime)
  MCExitCS
  If ErrorCode = -1 Then
    MCAbort resname, ErrorMsg
  End If
  ...
```

Chapter 11 • AppManager Callbacks for Summit BasicScript and VBScript 271
MCExitCS

Exit the AppManager agent-defined critical section.

Syntax

MCExitCS

Parameters and settings

None.

Return value

None.

Example

The following BasicScript code fragment comes from ARCserve_CanceledJobs:

... If IterationCount() = 1 Then
    Dim i%
    ...
    Filter=0
    If DO_ERR="y" Then Filter=1
    If DO_WARN="y" Then Filter=Filter+2
    MCEnterCS
    ErrorCode=ASMO.PreProcess(FileName, ErrorMsg, _
    StartOffset, FileCreateTime)
    MCExitCS
    If ErrorCode = -1 Then
        MCAbort resname, ErrorMsg
    End If
    ...

...
MCGetMOID

Retrieves version information for the managed object installed on the computer where the Knowledge Script is running. This extension is used to ensure that a particular version of a Knowledge Script calls a suitable version of a managed object.

For example, if the input parameters specify NetIQAgent.NT and 3.0.346, the returned string would be NetIQAgent.NT.3 and the appropriate DLL (for example, qnta3.d11) is loaded into memory.

Note A managed object may have multiple program ID entries in the registry. For example, when upgrading from 3.0 to 4.0, there may be three program ID entries in the registry, NetIQAgent.NT, NetIQAgent.NT.3, and NetIQAgent.NT.4. However, the version independent program ID (NetIQAgent.NT) always corresponds to the latest versioned program ID (for example, NetIQAgent.NT.4).

Syntax

MCGetMOID programid, version

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>programid</td>
<td>String</td>
<td>Managed object program identifier. For example, NetIQAgent.NT.</td>
</tr>
<tr>
<td>version</td>
<td>String</td>
<td>Knowledge Script version (for example, AppManID or KSVerID parameter).</td>
</tr>
</tbody>
</table>

Return value

String representing the managed object version.
Example

The following code fragment illustrates the call in BasicScript:

```vba
Function MyGetProgID (progid As String) As String
    Dim version As String
    MCVersion "netiqmc.exe", version
    If version < "3.0" Then
        MyGetProgID = progid
    else
        MyGetProgID = MCGetMOID (progid, AppManID)
    End If
End Function
```

```vba
Dim NT As Object
Dim progid As String
Sub Main()
    If IterationCount() = 1 Then
        progid = MyGetProgId ("NetiQAgent.NT")
        Set NT = CreateObject(progid)
    End If
    ...
End Sub
```
MCVersion

Requests the AppManager agent to obtain the version string for the specified component file name.

Syntax

MCVersion component, verstr [,fullpath]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>String</td>
<td>Managed object component file name.</td>
</tr>
<tr>
<td>verstr</td>
<td>String</td>
<td>The returned corresponding version string.</td>
</tr>
<tr>
<td>fullpath</td>
<td>Bool</td>
<td>Optional. If True, component contains the full path to the filename; if False, the component's location is relative to the AppManager\bin directory. Default is False</td>
</tr>
</tbody>
</table>

Return value

None.

Remarks

The AppManager agent will automatically perform the wildcard operation and retrieve the latest version. For example, if the specified component is qsqla.dll and there exists both qsqla3.dll and qsqla4.dll, the AppManager agent will return the latter.

Example

The following code example from the Discovery_SQL Knowledge Script gets the latest version for the dynamic link library used in monitoring SQL Server:

```vbscript
' Get the qsqla.dll version
version = ""
MCVersion "qsqla.dll", version
```
MCWaitForObject (Summit BasicScript only)

Simulates the Win32 API function `WaitForMultipleObjects` to wait for objects to be signalled before continuing execution.

**Syntax**

`MCWaitForObject waitall, obj1 [, obj2, obj3, ...., obj10]`

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>waitall</code></td>
<td>Boolean</td>
<td>Set to True to wait for all objects. Set to False to wait for one or more specific objects (objn).</td>
</tr>
<tr>
<td><code>objn</code></td>
<td>Long</td>
<td>Object to wait for. You must identify at least one object. You can wait for up to 10 objects. For example: <code>MCWaitForObject(True, obj1, obj2, ..., obj10)</code></td>
</tr>
</tbody>
</table>

**Return value**

Long. The number of objects signalled or the object index.

**Remarks**

If the `waitall` parameter is True, this call waits until all objects are signalled, then returns the number of objects signalled. If the `waitall` parameter is False, this call returns when any specified object is signalled and the return value indicates which object in the input parameter list has been signalled. For example, if you use the following call:

```
ret = MCWaitForObject(False, obj1, obj2, obj3)
```

when the second object (obj2) is signalled, `MCWaitForObject` returns the value 2.

Unlike the `WaitForMultipleObjects` function in the Win32 API, the `MCWaitForObject` caller thread will not block other Knowledge Script
threads on the managed computer, allowing for parallel processing of all running jobs.

**Example**

The following BasicScript code segment illustrates waiting for any of five objects to be signalled before returning (return value dependent on the object signalled):

```basic
ret = MCWaitForObject(False, h1, h2, h3, h4, h5)
```

The following BasicScript code fragment is from **General_PingMachine**:

```basic
...  
ret = MCWaitForObject (True, pInfo.hProcess)  
If (ret = 0) Then  
    retmsg = "Failed to wait process with " & ret  
    GOTO MyExit  
End If  
...  
```
MCWaitForObjectEx (Summit BasicScript only)

Simulates the Win32 API function `WaitForMultipleObjects` to wait for objects to be signaled before continuing execution. This function allows you to specify a maximum waiting period.

**Syntax**

```plaintext
MCWaitForObjectEx waitall, waitinterval, obj1 [, obj2, obj3, ...., obj10]
```

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitall</td>
<td>Boolean</td>
<td>Set to True to wait for all objects. Set to False to wait for one or more specific objects (objn).</td>
</tr>
<tr>
<td>waitinterval</td>
<td>Long</td>
<td>Maximum number of milliseconds to wait for objects to be signaled.</td>
</tr>
<tr>
<td>objn</td>
<td>Long</td>
<td>Object to wait for. You must identify at least one object. You can wait for up to 10 objects.</td>
</tr>
</tbody>
</table>

**Return value**

Long. The number of objects signaled or the object index. If the wait interval expires or objects are not signaled, the function returns a value of -1.

**Remarks**

If the `waitall` parameter is True, this call waits until all objects are signalled, then returns the number of objects signalled. If the `waitall` parameter is False, this call returns when any specified object is signalled and the return value indicates which object in the input parameter list has been signalled. For example, if you want to wait 10 seconds for two processes you can use a call such as:

```plaintext
ret = MCWaitForObjectEx (True, 10000, Process1, Process2)
```
The wait interval is specified in milliseconds. For example, you would set the interval to 10000 to wait for 10 seconds. When the second object (pFlag.hProcess2) is signalled, MCWaitForObjectEx returns the value 2. If this job is waiting for objects to be returned when a user stops another job, all the jobs this agent is running stop until this wait interval expires. If the object will never come back because of some other problem, then all the jobs would be stuck. Therefore you should use one of these options:

- Use a reasonably short interval of a few minutes or less.
- Use the WaitForObject function in a VB script.

Unlike the WaitForMultipleObjects function in the Win32 API, the MCWaitForObjectEx caller thread will not block other Knowledge Script threads on the managed computer, allowing for parallel processing of all running jobs.

Example

The following code segment illustrates waiting for all three objects to be signalled before returning (return value 3) in BasicScript:

\[ \text{ret} = \text{MCWaitForObjectEx(True,100,h1,h2,h3)} \]

The following BasicScript code segment illustrates waiting for any of five objects to be signalled before returning (return value dependent on the object signalled):

\[ \text{ret} = \text{MCWaitForObjectEx(False,100,h1,h2,h3,h4,h5)} \]
MSActions

Allows a Knowledge Script to report events and initiate actions. Knowledge Scripts that trigger events include this call.

**Note** If you are writing your Knowledge Script in VB Script and using NQEXT, you cannot pass multiple detail message strings using NQEXT.MSActions or NQEXT.CreateEvent. To pass long messages, either concatenate the strings or write them to a file, then use the NQEXT.MSLongActions call to return the contents of the file.

**Syntax**

MSActions severity, shortmsg, akpid, objlist, detailmsg [, detailmsg2, detailmsg3, ......,detailmsg6] [, value]

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>Long</td>
<td>Severity of the event.</td>
</tr>
<tr>
<td>shortmsg</td>
<td>String</td>
<td>Event message displayed in the List pane.</td>
</tr>
<tr>
<td>akpid</td>
<td>String</td>
<td>Action name or identifier for the action to be taken.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Objects that report the event (represented by blinking icons in the Operator Console's TreeView pane).</td>
</tr>
</tbody>
</table>
One key part of building custom Knowledge Scripts is the ability to generate events and initiate actions requested by the user. To allow your custom Knowledge Scripts to trigger events and actions, you need to include the special \texttt{MSActions} call in the main script logic.

The \texttt{MSActions} call controls how events are displayed in the Operator Console and the information stored in the AppManager repository.

You need to include an \texttt{MSActions} call for each event to be raised. Therefore, if a Knowledge Script is intended to trigger different events for different conditions (for example, a severe event when a check fails and an information event when successful), you need to include multiple \texttt{MSActions} function calls.

The following table describes the parameters for the \texttt{MSActions} function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>detailmsg</td>
<td>String</td>
<td>Detail message from the AppManager agent(s) displayed in the event's Properties dialog. At least one detailmsg is required. The maximum size of the string is 32K. To pass additional information beyond the 32 K, you can specify up to 6 message strings, each with a maximum size of 32K, to define the entire detail message for an event. For example, if the message you want to return is 64K, the message would be stored in two strings: \texttt{MSActions Severity, &quot;High&quot;, AKPID, &quot;,&quot;, detailmsg, detailmsg2}</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Optional. The current value to raise an event.</td>
</tr>
</tbody>
</table>

Return value

None.

Remarks

One key part of building custom Knowledge Scripts is the ability to generate events and initiate actions requested by the user. To allow your custom Knowledge Scripts to trigger events and actions, you need to include the special \texttt{MSActions} call in the main script logic.

The \texttt{MSActions} call controls how events are displayed in the Operator Console and the information stored in the AppManager repository.

You need to include an \texttt{MSActions} call for each event to be raised. Therefore, if a Knowledge Script is intended to trigger different events for different conditions (for example, a severe event when a check fails and an information event when successful), you need to include multiple \texttt{MSActions} function calls.
Example 1

For example, in the **NT_FilesOpen** Knowledge Script:

```basic
... If Dval1 > TH_FILES Then
    longm = "NT # of files open is " & Cstr(Dval1) & "; _
    >TH = " & Cstr(TH_FILES)
    MSActions Severity, "No of Files Open High", AKPID, _
    "", longm
End If
```

The following BasicScript example illustrates the use of multiple detail messages added to the MSActions call (although in this case they are not necessary because the messages passed are less than 32K):

```basic
detailmsg1 = "CPU load is " & Cstr(Dval1) & "; The Threshold is for CPU usage" & Cstr(TH_USE)
MSActions 5, " CPU Load", AKPID, resource, _
   detailmsg1,"detail2","detail3"
```

**Note** If you are writing your Knowledge Script in VB Script and using NQEXT, you cannot pass multiple detail message strings using NQEXT.MSActions or NQEXT.CreateEvent. To pass long messages, either concatenate the strings into a single message, or write the strings to a file, then use the NQEXT.MSLongActions call to return the contents of the file.
Example 2

As illustrated in this BasicScript example from *NT_LogicalDiskSpace*, the *MSActions* call is used to trigger an event when the disk space used or free space available exceeds a threshold:

```
If DO_EVENT = "y" Then
    If Dutil > TH_UTIL Or Dfree < TH_FREE Then
        Dim eventmsg$
        Dim detailmsg$
        Dim resname$
        eventmsg = "Disk " & objname & " Full"
        detailmsg = "Disk " & objname

        If Dutil > TH_UTIL Then
            detailmsg = detailmsg & " Used % is " & CStr(TH_UTIL)
        End If

        If Dfree < TH_FREE Then
            detailmsg = detailmsg & "Free space MB is " & Str(TH_FREE)
        End If

        resname = "NT_LogicalDiskObj = " & objname
        MSActions Severity, eventmsg, AKPID, resname, detailmsg
    End If
End If
```
MSLongActions

This function works the same as “MSActions” on page 280 except the AppManager agent will read the event detail message from the specified file.

Syntax

\[ \text{MSLongActions} \text{ sev, shortmsg, akp, objlist, msgfile} \\
\quad [, deletefile] [, value] \]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>Long</td>
<td>Severity of the event.</td>
</tr>
<tr>
<td>shortmsg</td>
<td>String</td>
<td>Event message displayed in the List pane.</td>
</tr>
<tr>
<td>akpid</td>
<td>String</td>
<td>Action name or identifier for the action to be taken.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Objects that report the event (represented by blinking icons in the Operator Console’s TreeView pane).</td>
</tr>
<tr>
<td>msgfile</td>
<td>String</td>
<td>The name of file that contains the detail message for the current event.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Boolean</td>
<td>Optional. True to delete the file after it is read and False to retain.</td>
</tr>
<tr>
<td>value</td>
<td>Double</td>
<td>Optional. The current value to raise an event.</td>
</tr>
</tbody>
</table>

Return value

None.

Example

This code fragment from BackupExec_FailedJobs reads the detail message from FName2:

\[
\text{If } \text{DOEVENT} = "y" \text{ And Dtotal} > \text{TH_USAGE Then} \\
\quad \text{MSLongActions SEVERITY, "The total number of failed _ backup jobs exceeds the threshold", _ AKPID, resname, strFilename} \\
\text{End If}
\]
NQSleep

Requests the AppManager agent to sleep for the specified interval on behalf of the KS.

Syntax

NQSleep intv [, noabort]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intv</td>
<td>Long</td>
<td>Sleep interval in msec</td>
</tr>
<tr>
<td>noabort</td>
<td>Bool</td>
<td>Optional. Request AppManager agent not to abort sleep in any condition. Default is False</td>
</tr>
</tbody>
</table>

Return value

Long. 1=sleep completes, -1=sleep aborted

Remarks

The parameter noabort requests the AppManager agent not to abort the sleep—even if the current script is being stopped.

Example

Sub Main()
    Dim szAdminCmd, LogDirName, FileUTC
    Dim LatestJobID, JobFileName, JobDetailFile
    ...
    JobDetailFile = JobFileName & "." & FileUTC & ".t"
    '5 retrys, if after 5 secs file still doesn't exist 'call it quits
    Cnt = 0
    Do
        NQEXT.NQSleep (5000)
        Cnt = Cnt + 1
        Loop While FSO.FileExists(JobDetailFile) = False And _
            Cnt < 5
    Loop
    ...

Chapter 11 • AppManager Callbacks for Summit BasicScript and VBScript 285
QTrace

Records a Knowledge Script trace message to a log file. This trace message is stored in the log file `mctrace.log` in the temporary directory located under the AppManager installation directory on the managed computer.

**Syntax**

QTrace msg

**Parameters and settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>String</td>
<td>Trace message to be logged.</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Remarks**

For example, if your AppManager installation directory is `C:\NetIQ\AppManager` on the managed computer and the management server is `Tango`, the path to the trace log is `C:\NetIQ\Temp\NetIQ_debug\Tango\mctrace.log`. 
Example

In the **Action_DosCommand** Knowledge Script, this function is used to log the result of the command as follows:

```vbscript
... 
Sub Main()
    Dim pInfo As PROCESS_INFO
    Dim sInfo As STARTUPINFO
    Dim sNull As String, errmsg$
    Dim success As Boolean, allowed As Boolean
    Dim ret&

    ... 
    success = CreateProcess (sNull, DOScmd, 0&, 0&, _
    0&, NORMAL_PRIORITY_CLASS, _
    0&, sNull, sInfo, pInfo)

    QTrace "CreateProcess <" & DOScmd & "> return " & success 
    ... 
End Sub
```
WaitForObject

Requests the AppManager agent to check the input object handle status on the behalf of current script.

Syntax

waitForObject hobj, intv [, noabort]

Parameters and settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hobj</td>
<td>Long</td>
<td>Input object handle to be waited on</td>
</tr>
<tr>
<td>intv</td>
<td>Long</td>
<td>Sleep interval in msec</td>
</tr>
<tr>
<td>noabort</td>
<td>Bool</td>
<td>Optional. Request MC not to abort sleep in any condition. Default is False</td>
</tr>
</tbody>
</table>

Return value

Long. 1=object signalled, 0=timer expired, -1=wait aborted.

Remarks

Set noabort if you do not want the wait to be interrupted by a user stop job request.
Example

Here is an example taken from the Async_FilesChanged Knowledge Script:

Sub Main()
    ...
    Set gObjNtFiles = CreateObject("NetIQAgent.NtFiles")
    ...
    hEvent = gobjNtFiles.GetEvent()

    ret = NQEXT.WaitForObject(hEvent, 0)
    If ret <> 1 Then
        If ret = 0 Then
            ' timeout
            NQEXT.AbortScript gsResName, "WaitForObject timeout"
            Exit Sub
        ElseIf ret = -1 Then
            ' job stop
            Exit Sub
        Else
            NQEXT.AbortScript gsResName, "WaitForObject error: _
            " & CStr(ret)
            Exit Sub
        End If
    End If
    ...
End Sub
Chapter 12

AppManager Callbacks for Perl

This chapter describes the calls made to the AppManager agent from Knowledge Scripts written in Perl. A call can be a function that returns a result, or a subroutine that does not return a result.

These Perl Callbacks are defined in the Perl module NetIQ::Nqext. The module must be loaded before any Callbacks are made. To load the module, include this statement:

```
use NetIQ::Nqext;
```

The following Callbacks are described:

- `AbortScript()`
- `CounterValue()`
- `CreateData()`
- `CreateEvent()`
- `ExecCmd()`
- `ExportData()`
- `ExportHugeData_pl()`
- `GetJobID()`
- `GetMachName()`
- `GetScriptInterval()`
- `GetTempFileName()`
- `ImportData()`
- `ImportHugeData_pl()`
- `IterationCount()`
AbortScript( )

Instructs the AppManager agent to abort execution of the Knowledge Script.

**Syntax**

```c
NetIQ::Nqext::AbortScript ([objlist, abortmsg, sev [,raise_err]])
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objlist</td>
<td>String</td>
<td>Object name. This parameter uses the syntax: <code>&lt;objtypename&gt; = &lt;objname&gt;</code>. For example, UNIX_CPUObj = 0.</td>
</tr>
<tr>
<td>abortmsg</td>
<td>String</td>
<td>Event detail message indicating execution of the script was aborted.</td>
</tr>
<tr>
<td>sev</td>
<td>Long</td>
<td>Abort event severity. See note under Remarks.</td>
</tr>
<tr>
<td>raise_err</td>
<td>Long</td>
<td>Optional. Default is 1. Defines job status as Error (1) or Stopped (0).</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Remarks**

When used without arguments, the AppManager agent aborts the script without raising an event.

If you specify any one of the parameters `objlist`, `abortmsg`, or `severity`, the agent will send an event to the AppManager management server.

**Note** There is an AppManager management server registry setting (“config\MC job abort event sev”) that overrides any value that you assign to `sev`, as long as the registry setting is non-zero. “config\MC
Job abort event Sev" is normally set to 10, and the abort event severity will therefore be 10, no matter what value you give to sev. If "config\MC job abort event Sev" is set to zero, then the registry value will no longer override sev.

Use raise_err to define the job status as Error (when set to 1) or Stopped (when set to 0).

Example

use NetIQ::Nqext;
# if the file doesn't exist, then exit the script and stop # the job
if (!(-e "$File_path")) {
  NetIQ::Nqext::AbortScript ($objlist, _
    "Cannot open $File_path -- file does not exist.",
    $Severity);
  die;
}
CounterValue( )

Requests the AppManager agent to retrieve the counter value for the specified object, counter, and instance.

Syntax

NetIQ::Nqext::CounterValue (object, counter, instance)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>String</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>counter</td>
<td>String</td>
<td>Name of the counter.</td>
</tr>
<tr>
<td>instance</td>
<td>String</td>
<td>Name of the instance.</td>
</tr>
</tbody>
</table>

Return value

Double. Returns the counter value for the specified object, counter, and instance. If the counter does not exist or there was an error retrieving the counter value, then -1 is returned.

Example

use NetIQ::Nqext;
...
$value = NetIQ::Nqext::CounterValue ("UX Processor", "%User Time", "")
if ($value = -1) {
  # Raise some kind of error event about not able to find the counter value
  ...
}
else if ($value > $threshold) {
  # Raise an event about % user time is too high
  ...
}
CreateData( )

Requests the AppManager agent to create a data point or a data source.

Syntax

NetIQ::Nqext::CreateData (streamId, legend, dynaleg, objlist, val, agentmsg, msgtype [,schema] [,loglimit] [,lowWM] [,hiWM] [,deletefile])
## Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamId</td>
<td>String</td>
<td>The data stream ID. For each unique stream ID in a script, it will generate a Data Source in the AppManager database. Subsequent calls to nqext_CreateData using the same stream ID will insert data points to the same Data Source. The string length limit is 64 characters.</td>
</tr>
<tr>
<td>legend</td>
<td>String</td>
<td>The data stream legend. This value will show up under the Legend column and in the graphs.</td>
</tr>
<tr>
<td>dynaLeg</td>
<td>String</td>
<td>The data stream dynamic legend. Contains the dynamic information that can be used for reporting. If a portion of your legend changes often, then pass that text into this parameter. Otherwise leave it blank.</td>
</tr>
<tr>
<td>objlist</td>
<td>String</td>
<td>Corresponding object name where the data is collected on. This value is used for graphing and reporting. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;NT_DiskObject = D:&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. NT_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The data point value.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the data detail or a file name that contains the data detail. The data detail is basically an annotation of each data point, giving more information about the data point since the data point is just a numeric value. For example, the data point value may be 5 for the number of processes running, while the data detail may list the processes that are running. The detailed message is displayed in the Graph Data Detail dialog box for each data point. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg is a file name or the detailed message itself. If it is a file name, then the contents of the file are passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Optional. XML schema for dynamic table creation in RDB. Default is an empty string.</td>
</tr>
</tbody>
</table>
**Return value**

None.

**Example**

```perl
use NetIQ::Nqext;

... $cpu_usage0 = nqGetCpuUsage (0);
$cpu_usage1 = nqGetCpuUsage (1);
... # Create a data stream for cpu 0
NetIQ::Nqext::CreateData ("CPU_0", "% CPU Usage", "CPU 0",
"UNIX_CPUObject = 0", cpu_usage, "CPU usage = $cpu_usage",0);

# Create a data stream for cpu 1
NetIQ::Nqext::CreateData ("CPU_1", "% CPU Usage", "CPU 1",
"UNIX_CPUObject = 1", cpu_usage, "CPU usage = $cpu_usage",0);
... 
```

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loglimit</td>
<td>Long</td>
<td>Optional. The number of days to keep this data point in the database. Default -1, keep forever. The data points can be removed from the database by other means.</td>
</tr>
<tr>
<td>lowWM</td>
<td>Double</td>
<td>Optional. Low watermark. Default -1.0.</td>
</tr>
<tr>
<td>hiWM</td>
<td>Double</td>
<td>Optional. High watermark. Default -1.0.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This parameter is ignored if msgtype != 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it wants to send an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
**CreateEvent()**

Requests the AppManager agent to raise an event.

**Syntax**

`NetIQ::Nqext::CreateEvent(sev, evtmsg, akp, obj, val, agentmsg, evtsrc, evtid, msgtype [, deletefile])`
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sev</td>
<td>Long</td>
<td>The event severity. A value from 1 to 40.</td>
</tr>
<tr>
<td>evtmsg</td>
<td>String</td>
<td>The message to be displayed under the Message column in the Events tab.</td>
</tr>
<tr>
<td>akp</td>
<td>String</td>
<td>Name of the action script to launch as a response to this event. You would normally create an AKPID parameter as part of your script. When the job is dropped and you select an action, the UI will fill in the AKPID variable with the action name. You will just need to pass in the AKPID variable to the script.</td>
</tr>
<tr>
<td>obj</td>
<td>String</td>
<td>Corresponding object name where the event is raised. This value will determine which object in the TreeView pane to blink. Format of the value passed in should be &quot;ObjectName = ObjectValue&quot;, e.g. &quot;UNIX_DiskObject = /mnt/cdrom&quot;. The ObjectValue can normally be obtained by the drop object variable, e.g. UNIX_MachineFolder.</td>
</tr>
<tr>
<td>val</td>
<td>Double</td>
<td>The current value to raise the event. This parameter is currently not used. Set to 0.0.</td>
</tr>
<tr>
<td>agentmsg</td>
<td>String</td>
<td>Either the detail message or a file name that contains the detail message. The detailed message is displayed in the Message tab of the Event Property dialog box. If this parameter contains the name of a file, make sure you set the msgtype parameter to 1.</td>
</tr>
<tr>
<td>evtsrc</td>
<td>String</td>
<td>Reserved for future use. Set to &quot;&quot;.</td>
</tr>
<tr>
<td>evtid</td>
<td>Long</td>
<td>Reserved for future use. Set to 0.</td>
</tr>
</tbody>
</table>
Developing Custom Knowledge Scripts

Return value

None.

Remarks

If called by a script running on a 1.0 agent, the agent simply raises the event to the MSU. If called by a script running on a 2.0 agent, the agent will apply an additional rule process to determine whether to send a new event or duplicated (collapsed) event to the AppManager management server.

Example

```perl
use NetIQ::Nqext;
use NetIQ::Oracle;
...

$sp = NetIQ::Oracle::GetLogSpace (...);
...

# Create an event with severity 5 and message "Log space ...
NetIQ::Nqext::CreateEvent (5, "Log space dangerously low", $akpid, "UNIX_MachineFolder = MyServer", 0, Log space is $sp", "NetIQ :: Oracle", 1000, 0);
...

```

Parameter Data type Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgtype</td>
<td>Long</td>
<td>Flag specifying whether the value passed in the agentmsg parameter is a file name or the detailed message itself. If it is a file name, then the contents of the file are read and passed in as the detailed message. Set to 0 to specify that the value in the agentmsg parameter is the detailed message. Set to 1 to specify that the value is the file name containing the detailed message.</td>
</tr>
<tr>
<td>deletefile</td>
<td>Long</td>
<td>Optional. Flag to tell the AppManager agent to delete the event detail message file after it is done reading the contents and passing the event to the MSU. This flag is ignored if msgtype != 1. Set to 1, which is default, to delete the file when msgtype = 1. Set to 0 to not delete the file. Be careful when setting this value to 0, especially if your script generates a message file each time it sends an event because the files will never be removed.</td>
</tr>
</tbody>
</table>
**ExecCmd( )**

The Perl language allows invocation of external commands by using back quotes (` `) to substitute the output of the enclosed command. The AppManager UNIX agent does not support this.

`ExecCmd` instructs the agent to execute an external command on behalf of the Knowledge Script.

**Syntax**

`NetIQ::Nqext::ExecCmd (cmd [, flag])`

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>String</td>
<td>The non-interactive command.</td>
</tr>
<tr>
<td>flag</td>
<td>Long</td>
<td>Optional. 0: the Callback returns the stdout. 1: the Callback returns the temporary file name containing the stdout. 2: the Callback returns the stdout along with the stderr. 3: the Callback returns the temporary file name containing both the stdout and stderr. Default is 0</td>
</tr>
</tbody>
</table>

**Return value**

String. Depending on the flag passed in, this Callback will either return the `stdout` and/or `stderr` results or a filename containing the `stdout`/`stderr` results from executing the external command.

**Remarks**

If `flag` is 1 or 3, then the Knowledge Script must remove the temporary file after it is used.
Example

A Perl script statement invoking an external command should be changed from

```perl
$a = `cmd`;
```

to

```perl
$a = NetIQ::Nqext::ExecCmd("cmd");
```

A Perl script statement that reads from a pipe should be changed from

```perl
open (F, "cmd |");
...
close F;
```

to

```perl
$fh = NetIQ::Nqext::ExecCmd("cmd", 1);
open (F, $fh);
...
close F;
```

unlink $fh;
ExportData( )

Requests the AppManager agent to save the specified string-based scalar script variable along with the value in memory for referencing in subsequent job iterations. If the variable already exists, the value is updated.

Syntax

NetIQ::Nqext::ExportData (name, val)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Variable name for storing val.</td>
</tr>
<tr>
<td>val</td>
<td>String</td>
<td>Script variable value.</td>
</tr>
</tbody>
</table>

Return value

None.

Remarks

Note that ExportData can only work with string scalar variables. For any numeric scalar variables, you will need to convert it into a string before calling ExportData to store it. For array and hash variables, use ExportHugeData.pl.

Scripts can use the ImportData Callback to retrieve the stored value, so that you can define a global variable at the AppManager agent scope, and update and retrieve its value.

Note If either the agent or the job is stopped and restarted, exported data may not persist.
Example

use NetIQ::Nqext;
...
our $Persist_Var3; # not recommended
...
my $var1 = 5;
my $var2 = "6";
...
sub save_data {
    my $tmp = "$var1";
    NetIQ::Nqext::ExportData ("var1", $tmp);
    NetIQ::Nqext::ExportData ("var2", $var2);
}

sub load_data {
    $var1 = NetIQ::Nqext::ImportData ("var1");
    $var2 = NetIQ::Nqext::ImportData ("var2");
}
...
# import the data at the start of the script
load_data();

# $var1 increments by 1 every iteration while $var2 # increments by 2.
$var1 += 1;
$var2 += 2;
$Persist_Var3 = $Persist_Var3 + $var1 + $var2;

save_data();
...
ExportHugeData_pl( )

Instructs the AppManager agent to export a scalar, array, or hash variable associated with the label name.

**Syntax**

NetIQ::Nqext::ExportHugeData_pl (name, val_ref)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The label for this variable. You should use the variable name without the $, @, or % prefix.</td>
</tr>
<tr>
<td>val_ref</td>
<td>Reference to a variable</td>
<td>The reference to the variable that must be persistent. In Perl, the reference to variable $v is specified as $v.</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Remarks**

This Callback is based on the Perl Storable module, which can import huge variables. Unlike ImportData and ExportData, which store the values in memory, ImportHugeData_pl and ExportHugeData_pl store values on disk, and are therefore slower.

**Example**

```perl
use NetIQ::Nqext;
our %P_history;
...
ExportHugeData_pl('P_history', \%P_history);
```
GetJobID( )

Queries the AppManager agent for the AppManager repository job ID that the calling script is running under.

Syntax
NetIQ::Nqext::GetJobID()

Parameters
None.

Return value
Long. Returns the AppManager repository job ID of the calling script.

Example
use NetIQ::Nqext;
...
my $jobid;
...
$jobid = NetIQ::Nqext::GetJobID();
...
GetMachName()  

Queries the AppManager agent for the agent’s computer name.

**Syntax**

NetIQ::Nqext::GetMachName()

**Return value**

String. The agent machine name.

**Example**

```perl
use NetIQ::Nqext;
...
my $machine_name;
...
$machine_name = NetIQ::Nqext::GetMachName();
...```
GetScriptInterval()

Queries the AppManager agent for the current job interval.

Syntax

NetIQ::Nqext::GetScriptInterval()

Parameters

None.

Return value

Long. Returns the number of seconds between job iterations.

Example

use NetIQ::Nqext;
...
my $intv;
...
$intv = NetIQ::Nqext::GetScriptInterval();
...
GetTempFileName()

Instructs the AppManager agent to construct a temporary file name based on input criteria. The file name is concatenated from the path and prefix strings, and a hex string based on uniqid.

Syntax
NetIQ::Nqext::GetTempFileName (path, prefix, uniqid)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>String</td>
<td>Path for created temp file.</td>
</tr>
<tr>
<td>prefix</td>
<td>String</td>
<td>File name prefix string.</td>
</tr>
<tr>
<td>uniqid</td>
<td>Long</td>
<td>A unique ID.</td>
</tr>
</tbody>
</table>

Return value
String. Returns the full path of the temporary file that was created.

Remarks
The file that is created is guaranteed to be a unique file. Therefore, scripts that call GetTempFileName should delete the file after it is used.

Example
use NetIQ::Nqext;
...
$fname = NetIQ::Nqext::GetTempFileName ();
$cmd = "ps -ef | grep nqmagt > $fname";
$stdout_result = ExecCmd ($cmd);
If (-s "fname") {
    open(RESULT_FILE, $fname);
    ...
    close(RESULT_FILE);
    unlink($fname);
}
**ImportData( )**

Requests the AppManager agent to retrieve the value stored under the specified variable name by an ExportData call.

Along with ExportData, Knowledge Script writers can define a global variable at the agent scope, update and retrieve its value.

**Syntax**

NetIQ::Nqext::ImportData (name)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Script variable name.</td>
</tr>
</tbody>
</table>

**Return value**

String.

**Remarks**

ExportData can only work with string scalar variables. Numeric scalar variables must be converted to a string before calling ExportData. However, when you import the variables into the script using ImportData, you do not need to do any conversion from string to a numeric value because Perl will handle that for you.

For array and hash variables, use ExportHugeData_p1 and ImportHugeData_p1.

**Note** If either the AppManager agent or the job is stopped and restarted, exported data may not persist.
Example

use NetIQ::Nqext;
...
our $Persist_Var3; # not recommended
...
my $var1 = 5;
my $var2 = "6";
...
sub save_data {
    my $tmp = "$var1";
    NetIQ::Nqext::ExportData ("var1", $tmp);
    NetIQ::Nqext::ExportData ("var2", $var2);
}

sub load_data {
    $var1 = NetIQ::Nqext::ImportData ("var1");
    $var2 = NetIQ::Nqext::ImportData ("var2");
}
...
# import the data at the start of the script
load_data ();

# $var1 increments by 1 every iteration while $var2
# increments by 2.
$var1 += 1;
$var2 += 2;
$Persist_Var3 = $Persist_Var3 + $var1 + $var2;

save_data ();
...
**ImportHugeData_pl( )**

Instructs the AppManager agent to import a scalar, array, or hash variable associated with the label name.

**Syntax**

NetIQ::Nqext::ImportHugeData_pl (name)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The label for this variable. You should use the variable name without the $, @, or % prefix.</td>
</tr>
</tbody>
</table>

**Return value**

Reference. Returns a Reference to the imported variable.

**Remarks**

This Callback is based on the Perl *Storable* module which can import huge variables. Unlike *ImportData* and *ExportData*, which store values in memory, *ImportHugeData_pl* and *ExportHugeData_pl* store values on disk, and are therefore slower.

**Example**

To import a hash:

```perl
...  %P_file = %{ NetIQ::Nqext::ImportHugeData_pl('P_file') };  ...
```
IterationCount( )

Queries the AppManager agent for the current job iteration value.

Syntax
NetIQ::Nqext::IterationCount()

Parameters
None.

Return value
Long. The number of iterations the job has run since it was started, including the current job.

Example
use NetIQ::Nqext;
...
if (NetIQ::Nqext::IterationCount () == 1) {
    InitializeMyObject ();
}
...

Chapter 13

Testing and debugging

This chapter describes how to open the debuggers for Knowledge Scripts. The following topics are covered:

- Debugging Knowledge Scripts
- Where to debug scripts
- The prepend and append files
- Setting debuggers for VBScript and BasicScript
- Debugging Summit BasicScript scripts
- Debugging VBScript scripts
- Debugging Perl scripts

Debugging Knowledge Scripts

The emphasis in this book is on modifying existing Knowledge Scripts. This means that the changes you make to your scripts will be easily isolated in the event that the scripts do not run properly. Under such circumstances, you are unlikely to need to do difficult debugging.

In the event that you do need to debug your scripts with a debugging program, tools are provided to help you do this. Prior to debugging, you should check the script’s syntax:

- You can check the syntax of scripts written in VBScript or Perl in the Developer's Console. Choose the **VBScript Syntax Check** or **Perl Syntax Check** commands on the **Tools** menu.

- You can check the syntax of scripts written in Summit BasicScript in the Knowledge Script Editor. Choose the **Syntax Check** command on the **Run** menu.
Where to debug scripts

Summit BasicScript Knowledge Scripts must be debugged on the computer where the Knowledge Script Editor is installed. This is probably the same computer that hosts the Developer’s Console. An AppManager agent must be installed on this computer, so that the managed object methods needed by the scripts are installed and registered. The Knowledge Script Editor includes a debugging program, but only for Summit BasicScript.

VBScript Knowledge Scripts must also be debugged on a computer with an AppManager agent installed, so that the managed object methods are available. You will also need to install a Microsoft Windows Script Host on this system—this can be run in debug mode (with parameter = //D) and you can download it from:


Perl scripts must be debugged on a computer that has the Perl interpreter installed. This can be a Windows machine. If you are debugging a script that calls methods from NetIQ Perl modules (most Perl scripts do not require NetIQ modules), they must be available—in this case, you will need to debug the Perl script on a UNIX host that has the AppManager UNIX agent installed and any Perl modules that the script may be calling.

For maximum convenience, you should install the Perl interpreter on the computer that hosts both the AppManager Operator Console and the Developer’s Console. Then you can debug most Perl scripts on the same system where you develop them.

Setting debuggers for VBScript and BasicScript

If you set the appropriate debuggers for Summit BasicScript and VBScript in the Developer’s Console, you will be able to launch them automatically. To set them (you must obtain and install the Windows Script Debugger first), choose Set Debuggers... in the Tools menu.
of the Developer's Console. This will open the **Set Debuggers** dialog box:

If you have used the default installations, the paths to the debuggers will be:

<table>
<thead>
<tr>
<th>Scripting Language</th>
<th>Debugger</th>
<th>Default Path to Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit</td>
<td>NetIQ Knowledge Script Editor</td>
<td>C:\Program Files\NetIQ\AppManager\bin\niqnedt.exe</td>
</tr>
<tr>
<td>VBScript</td>
<td>Microsoft Script Host</td>
<td>C:\WINNT\system32\cscript.exe</td>
</tr>
</tbody>
</table>

**NOTE** To use this Script Host in debug mode, it must be launched with either the //D or //X parameter.

The Microsoft Windows Script Host requires a command line argument, either //D or //X, to run in debug mode. You must enter one of these arguments in the third field of the **Set Debuggers** dialog box. If you use the //D parameter, the debugger will only open if an error occurs. Using the //X parameter starts the debugger and puts a breakpoint on the first executable line of script code.
Developing Custom Knowledge Scripts

The prepend and append files

Prepend files

Knowledge Scripts can call two different types of methods that are not defined in the script: managed object methods and Callback functions. If you are debugging your script on a computer with the appropriate managed objects installed, you can debug without worrying about them. Note that the managed objects can be registered on a computer where an AppManager agent is not present.

Callback functions are available only in the AppManager agent running the script. When you are debugging, your script will be run by the debugger program, not by the agent. Therefore, the script you are debugging will not be able to access the Callback functions. Execution will stop when the debugger reaches a function it cannot call.

Three prepend files, one for each scripting language, contain simplified versions of the standard Callback functions. When these files are added to the beginning of your script, the script can call the simplified Callback functions in lieu of the real Callback functions, and therefore run in the debugger. The simplified Callback functions “return” a message box saying that the function was called successfully (or print to stdout in Perl scripts).

For example the Summit BasicScript prepend file (PrependFile.ebs) contains a subroutine definition for every Callback. Here is the code for MSActions:

```basic
Sub MSActions( sevlevel As Long, shortmsg As String, _
    AKPName As String, objlist As String, _
    agtmsg As String)
If OutputMode = WINDOW_MODE Then
    MsgBox "Call back function MSActions with parameters:_
        & Chr(13) & Chr(10) & "Severity: " & _
        sevlevel & Chr(13) & Chr(10) & _
        "shortmsg: " & shortmsg & Chr(13) & Chr(10) & _
        "AKPName: " & AKPName & Chr(13) & Chr(10) & _
        "objlist: " & objlist & Chr(13) & Chr(10) & _
        "agtmsg: " & agtmsg
```

318  Developing Custom Knowledge Scripts
ElseIf OutputMode = FILE_MODE Then
    Print #1, "Call back function MSActions with _ parameters: "
    Print #1, "Severity: " & sevlevel
    Print #1, "shortmsg: " & shortmsg
    Print #1, "AKPName: " & AKPName
    Print #1, "objlist: " & objlist
    Print #1, "agtmsg: " & agtmsg
End If
End Sub

Append files

VBScript is the default (and only script type) that can be newly created using the Developer Console. When you open the Developer Console, an empty Main subroutine is created as would be needed for a VBScript Knowledge Script. However, if you simply add content and then press F5 to initiate debug, the Main subroutine will never be called.

To correct this problem, pressing F5 will add the contents of an append file (AppendFile.vbs) at the end of the code to call Main. AppendFile.vbs might, for example, contain code like this:

' Execute main until the maximum iteration count is reached
Do While ((NQEXT.m_IterationCount <= NQEXT.ScheduleXNumberOfTimes) Or _
    (NQEXT.ScheduleXNumberOfTimes < 0))
    Main
    If ((NQEXT.m_IterationCount >= NQEXT.ScheduleXNumberOfTimes) And _
        Not (NQEXT.ScheduleXNumberOfTimes < 0)) Then
        Exit Do
    End If
    NQEXT.NextIteration
Loop
Location of files

The AppManager installation program puts the prepend and append files in the ...\NetIQ\AppManager\bin directory in your AppManager installation. The files are:

PrependFile.ebs
PrependFile.vbs
AppendFile.vbs
PrependFile.pl

Debugging Summit BasicScript scripts

If you have set the debugger path correctly, simply press F5 when you want to debug a script written in BasicScript. The contents of the BasicScript prepend file (PrependFile.ebs) will be added to the beginning of your code and the composite of the two will be opened in the Knowledge Script Editor debugger.

Caution You cannot save changes you make in the code during debugging. You must check out the script, make the same changes in the Developer Console, and check the script back in.

Debugging VBScript scripts

If you have set the debugger path correctly, simply press F5 when you want to debug a script written in VBScript. The contents of the VBScript prepend file (PrependFile.vbs) will be added to the beginning of your code, the contents of the VBScript append file (AppendFile.vbs) will be added to the end of your code, and the composite file will be opened in the Microsoft Script Debugger.

Caution Changes you make in the code during debugging will not be reflected the original script or the script in the repository. You must check out the script, make the same changes in the Developer Console, and check the script back in.
Debugging Perl scripts

There is no automated method for debugging Perl scripts as there is with VBScript and BasicScript.

To debug a Perl script, do this:

1. Create a new file in a text editor. Name it `mergedebug.pl`, for example.

2. Copy the contents of `PrependFile.pl` and paste them into `mergedebug.pl`.

3. Open the Perl script in the Developer's Console and select the Perl (Read-only) view.

4. Copy the entirety of the contents in the Perl (Read-only) view and paste them into `mergedebug.pl` at the end.

5. Comment out the line `use NetIQ::Nqext;` in `mergedebug.pl` and save the file.

Then you can debug your script in any Perl debugger. For example, you can run

```
perl -d mergedebug.pl
```

at the command line to use the Perl interpreter in debug mode.

When you have found the errors, check out the faulty script, make the necessary code modifications using the Developer's Console, and check the script back in.
action  A response to an event. For example, an e-mail message may be sent in response to a particular computer or service going down. In AppManager, actions are typically handled by action Knowledge Scripts.

action schedule  A schedule for specifying when an action Knowledge Script can run.

AppManager agent  A Windows NT service (NetIQmc and NetIQccm) that runs on a managed computer and receives requests from the management server to run or stop a Knowledge Script job. The agent communicates back to the management server, on an exception-basis, any relevant output from a Knowledge Script. See also managed client.

AppManager management server  A Windows NT service (NetIQms) that allows AppManager agents on managed clients (servers and workstations) to communicate with the AppManager repository database.

AppManager Operator Console  The main user interface that allows you to view, configure, and control the execution of Knowledge Scripts on the systems and applications you manage.

AppManager Operator Web Console  A Web interface that allows you to view and manage computer resources from virtually any location using a Web browser. Includes the Report View which allows you to view reports and the Chart Console which allows you to generate and view charts of graph data.

AppManager report agent  An optional component installed with the AppManager agent which enables the AppManager agent to run Report Scripts and generate AppManager reports. See also AppManager agent.
Developing Custom Knowledge Scripts

**AppManager reports**  HTML files that can be read and printed using the Report Viewer available from the Operator Web Console.

**AppManager repository**  An SQL Server database that stores all AppManager data and relevant information about your managed environment. The combination of a repository and a management server constitutes a management site. You can only have one repository for each management server.

**AppManager Report Viewer**  A view available from the Operator Web Console that displays AppManager reports.

**corrective action**  An automated response to an event that corrects the problem found. For example, a corrective action might be to automatically restart a service when the service is detected down. In AppManager, most corrective actions are handled by Action Knowledge Scripts.

**data points**  Numeric information collected by a Knowledge Script during a monitoring period and stored in the AppManager repository.

**data stream**  Series of data points collected by a Knowledge Script over time.

**developer**  An individual who is modifying or creating Knowledge Scripts.

**event**  An alert or notification that some condition or activity you are using AppManager to monitor or keep watch for has occurred on a managed system.

**generated script**  The final script that is generated by AppManager to run as a job.

**job**  An instance of a Knowledge Script running on the AppManager agent that is resident on a managed client (a server or workstation you are monitoring).

**Knowledge Script**  A script (written in VBScript, Summit BasicScript, or Perl) that is encapsulated in an XML file along with other settings, such as parameters for the script, a schedule, and so forth. This script is run on the servers and workstations in your environment to check the health and availability of those systems, collect data for trend analysis, and initiate corrective or responsive actions.
Knowledge Script Group  A pre-configured set of Knowledge Script Group members; each member is an instance of a Knowledge Script. A Knowledge Script Group can be used to create a monitoring policy or to start ad hoc jobs.

Knowledge Script Group member  An instance of a Knowledge Script which is used to create a monitoring job or a policy-based job.

managed client  A server or workstation computer set up to be monitored by AppManager. See also AppManager agent.

managed objects  COM or OLE objects that are installed on a server or workstation when the AppManager agent is installed on that system.

management service  A Windows NT service (NetIQms) that runs on a single Windows NT server. This service manages event-driven communication between the AppManager console programs (Operator Console, Operator Web Console, Security Manager, and Distributed Event Console) and the servers and workstations you are monitoring. Once installed, the computer on which the service is running becomes a management server.

monitoring job  An instance of a Knowledge Script running on a server or workstation and monitoring particular resources.

monitoring policy  Automatically monitors resources on a managed computer as they change using policy-based jobs; a monitoring policy is implemented through one or more Knowledge Script Groups. See also Knowledge Script Group.

parameter  A variable used when calling a method, function, or subroutine. Not to be confused with a Script Parameter.

process  An object created when a program is run.

Properties dialog box  The dialog box that opens in the AppManager Operator Console when a user drags a Knowledge Script to a target object in the TreeView pane.

report scripts  Scripts that generate reports based on graph data in the AppManager repository.

repository host  The computer where AppManager data is stored.
**Script Parameters** Variables or constants in Knowledge Script code that can have their values changed by a user. The developer defines these parameters in the *Script Properties dialog box* and a user alters them using the *Properties dialog box*.

**Script Properties dialog box** The dialog box that opens in the AppManager Developer's Console when a developer chooses *Properties* from the View menu.

**server group** A logical grouping of servers and workstations you manage. A server group is represented by a folder in the *TreeView* pane and can contain individual machines or other server groups.

**target computer** Refers either to the computer that is itself the target object for a script, or to the computer that contains the target object (when the target object itself is a hardware device like a CPU, or a software application or service).

**thread** An object that executes instructions.

**threshold** A level, point, condition, or value that, when exceeded, generates an event or notification that the boundary specified has been passed.

**user** An individual who is using the AppManager Operator Console to run Knowledge Scripts.
Dialog Boxes

Script Editor Options dialog box

You use the **Script Editor Options** dialog box (choose **View > Options**) to set text display options for the text displayed in the **Edit** and **Read-only** views.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab Width</td>
<td>Specify the width for tabs in the Edit and Read-only views. The default tab width is four points.</td>
</tr>
<tr>
<td>Auto Indent</td>
<td>If this option is selected, any new line that you insert by pressing Enter is automatically moved to the same indent as the previous line. If this option is not selected, any new lines you enter will begin at the left margin. Auto indent is selected by default.</td>
</tr>
</tbody>
</table>
Developing Custom Knowledge Scripts

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
</table>
| Syntax Coloring | Select this option to turn on syntax coloring in the Edit view, and VB Script and BasicScript (Read-only) views. This option does not apply to the XML (Read-only) view.  
If you deselect this option, all text appears in black, and the Keyword Case option becomes inactive.  
Syntax coloring is selected by default. |
| Keyword Case | Select this option to capitalize the first letter of keywords displayed in the Edit or Read-only views (for example, with this option selected the keyword “for” appears as “For”).  
Keyword case is selected by default. |
| Font        | Click this button to display a dialog where you can specify the displayed font, style (such as bold or italic), and size of text displayed in the Edit and Read-only views.  
The default font is 10 pt. Courier New |
| Colors      | You can use the options in the Colors group box to apply colors and font styles to syntax elements (such as comments, operators, and strings) in the Edit view, and VB Script and Summit Basic (Read-only) views.  
Syntax coloring is applied by default.  
To view the current color of an element, select the element in the list. The color of the element is displayed in the drop-down list. If bold or italic style is applied, the appropriate option is selected.  
To change the color or style of a syntax element, select the element in the list, then select a color from the drop-down list, and the style options that you want to apply to that element.  
Operators are the only syntax element that appear in boldface type by default. |

Bold  
Italic
Header tab, Script Properties dialog box

Use the Header tab in the Script Properties dialog box (choose View > Properties > Header) to enter or modify the Header information for your Knowledge Script.
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge Script description</td>
<td>Short description of what this Knowledge Script does. This text is displayed</td>
</tr>
<tr>
<td></td>
<td>in the Knowledge Script pane when you click KS &gt; Show Description.</td>
</tr>
<tr>
<td>Knowledge Script type</td>
<td>Type of operation this Knowledge Script performs:</td>
</tr>
<tr>
<td></td>
<td>- <strong>Normal</strong> indicates the Knowledge Script performs a normal monitoring</td>
</tr>
<tr>
<td></td>
<td>or reporting task.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Action</strong> indicates the Knowledge Script performs an action.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Discovery</strong> indicates the Knowledge Script discovers resources.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Install</strong> indicates the Knowledge Script performs remote installation.</td>
</tr>
<tr>
<td>Require passwords</td>
<td>Indicates whether the Knowledge Script requires secure information, such</td>
</tr>
<tr>
<td></td>
<td>as a password or login information. Secure information is stored separately</td>
</tr>
<tr>
<td></td>
<td>from the Knowledge Script in the AppManager repository if you check this</td>
</tr>
<tr>
<td></td>
<td>option.</td>
</tr>
<tr>
<td>Option Explicit</td>
<td>Adds the Option Explicit statement to the beginning of VBScripts to force</td>
</tr>
<tr>
<td></td>
<td>variable definition.</td>
</tr>
<tr>
<td>Administrator’s use only</td>
<td>Indicates whether the Knowledge Script requires the user to be part of the</td>
</tr>
<tr>
<td></td>
<td>AppManager administrator group.</td>
</tr>
<tr>
<td>Target Operating System: Unix</td>
<td>Selecting this option deletes BasicScript and VB Script implementations of</td>
</tr>
<tr>
<td>only</td>
<td>the Knowledge Script. You can then write the programming logic in Perl. You</td>
</tr>
<tr>
<td></td>
<td>will also have to redefine the properties for the script. The script can</td>
</tr>
<tr>
<td></td>
<td>then be used by agents on UNIX computers.</td>
</tr>
<tr>
<td></td>
<td>When you select Unix only, you should be sure to choose the target platforms</td>
</tr>
<tr>
<td></td>
<td>in the bottom panel.</td>
</tr>
<tr>
<td>Target Operating System: Windows</td>
<td>Selecting this option deletes Perl implementations of the Knowledge Script.</td>
</tr>
<tr>
<td>only</td>
<td>You can then write the programming logic in VBScript. You will also have</td>
</tr>
<tr>
<td></td>
<td>to redefine the properties for the script. The script can then be used by</td>
</tr>
<tr>
<td></td>
<td>agents on Windows computers.</td>
</tr>
</tbody>
</table>
Object Types tab, Script Properties dialog box

Use the Object Types tab of the Script Properties dialog box (choose View > Properties > Object Types) to enter or modify the resource object type information for your Knowledge Script. For new Knowledge Scripts, the object type list is empty until you click Add and add the object types you want to use.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
</table>
| Supported scripting languages | Indicates the scripting languages supported in the Knowledge Script. For scripts used on Windows computers:  
  - Select Summit BasicScript to write the main script logic using the BasicScript scripting language.  
  - Select Visual Basic Script to write the main script logic using the VBScript scripting language.  
  For scripts used on UNIX computers:  
  - Select Perl Script to write the main script logic using the Perl scripting language. |
| AppManager Version        | Enter the AppManager version for the script. Only decimal numbers and periods are allowed. |
| Target Platforms          | Enabled only when you select Unix as the OS. Check all Unix platforms that the script will run on. |
### Item | Description
--- | ---
Object is dropped on folder list | Indicates whether the object can be dropped on a folder list.

Object uses full path | Indicates whether the object requires a full path to identify a target. For example, if a Knowledge Script is dropped on the "master" database for the SQL Server TULSA, the full object name path might look like this:

```
SQL Server:TULSA:Databases:master:10
```

The last part of the path, 10, represents the object id in the AppManager repository. You can then use the object id to construct the resource name. For example:

```
resname = "SQLT_DatabaseObj = #" & 10
```

In this line, the # sign indicates that the resource is being identified by object id, not by object name. You can then pass this information to MSActions to more efficiently raise an event. For example:

```
MSActions severity, eventmsg, AKPID, resname, detailmsg
```
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object path delimiter</td>
<td>Specifies the character used as a delimiter between object paths (if using the full path for an object). For example, if you specify a <code>\</code> character as the delimiter: <code>&lt;computer&gt;\&lt;folder&gt;\0</code></td>
</tr>
</tbody>
</table>
| Object instance delimiter   | Specifies the character used as a delimiter between individual objects in a list. For example, if you select the object type NT_CPUNumber and use a comma as the instance delimiter, if you drop the script on a computer with 2 CPUs (0 and 1) the code generated for the script looks like this:  
  `Const NT_CPUNumber = "0,1"` |
| Object type list            | Lists the object types currently defined for the script.                                                                                   |
Add New Object Type dialog box

Use the Add New Object Type dialog box (choose View > Properties > Object Types > Add) to select the object types that are applicable for this Knowledge Script.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object group</td>
<td>Lists the categories of object types available.</td>
</tr>
<tr>
<td>Object types</td>
<td>Lists the object types with the selected group.</td>
</tr>
<tr>
<td>Object to add</td>
<td>Displays your current object selection that will be added when you click Add.</td>
</tr>
<tr>
<td>Include object details</td>
<td>Allows you to select which, if any, object details are included in the Knowledge Script. The detail information available is specific to each object type and is optional.</td>
</tr>
</tbody>
</table>
Default Schedule tab, Script Properties dialog box

Use the Default Schedule tab of the Script Properties dialog box (choose View > Properties > Default Schedule) to set or modify the default schedule information for your Knowledge Script.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule type</td>
<td>Defines the type of schedule the script should use by default:</td>
</tr>
<tr>
<td></td>
<td>• Run once</td>
</tr>
<tr>
<td></td>
<td>• Regular interval</td>
</tr>
<tr>
<td></td>
<td>• X number of times</td>
</tr>
<tr>
<td></td>
<td>• Daily schedule</td>
</tr>
<tr>
<td></td>
<td>• Asynchronous</td>
</tr>
<tr>
<td>Iteration</td>
<td>Defines the default interval period, number of iterations, or start and end times.</td>
</tr>
</tbody>
</table>
Advanced Schedule Configuration dialog box

Use the Advanced Schedule Configuration dialog box (choose View > Properties > Default Schedule > Advanced) to configure the schedule choices that are available for this Knowledge Script. If you uncheck an allowed schedule, users will not be able to select the corresponding schedule when setting Knowledge Script properties for this Knowledge Script.

![Advanced Schedule Configuration dialog box](image)

### Advanced Schedule Configuration

- **Run once immediately before following scheduled run:**
  - **Allowed schedule:**
    - All
    - Run once
    - Interval or iteration
    - Daily
    - Weekly
    - Monthly

[OK] [Cancel] [Help]
Parameters tab, Script Properties dialog box

Use the **Parameters** tab of the **Script Properties** dialog box (choose **View > Properties > Parameters**) to enter or modify the parameters and default values for your Knowledge Script.

![Script Properties Dialog Box](image)

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief description of the parameters</td>
<td>Provides the information displayed in the Values tab about what the script does and how to set the parameters.</td>
</tr>
<tr>
<td>Parameter list</td>
<td>Lists the variable name, data type and default for the parameters currently defined for the script.</td>
</tr>
</tbody>
</table>
Add/Modify Parameter dialog box

Use the Add/Modify Parameter dialog box (choose View > Properties > Parameters > Add or Modify) to define information for the parameters to be included in the Values tab.

![Add new Parameter dialog box](Image)

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable name to use</td>
<td>Defines the variable name you use for a parameter value in the script logic.</td>
</tr>
<tr>
<td>Description</td>
<td>Defines the text displayed on the Values tab in the Knowledge Script Properties dialog box for the variable.</td>
</tr>
<tr>
<td>Data type</td>
<td>Defines the data type for the value associated with this variable:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Integer</td>
</tr>
<tr>
<td></td>
<td>• Double</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Delim</td>
<td>Defines the delimiter for parameters that accept multiple values. If there will not be multiple values, and whitespace is a valid part of the parameter, set this value to comma &quot;,&quot;</td>
</tr>
<tr>
<td>User Interface control type</td>
<td>Defines the user interface control to display for the parameter.</td>
</tr>
<tr>
<td>Min</td>
<td>Defines the minimum valid value for a parameter. This field applies for Integer and Double variables. It is not application for string type variables.</td>
</tr>
<tr>
<td>Max</td>
<td>Defines the maximum valid value for a parameter. This field applies for Integer and Double variables. It is not application for string type variables.</td>
</tr>
<tr>
<td>Unit</td>
<td>Defines the unit associated with the value for a parameter. For example, you can specify that values represent a percentage, MB, or severity level. This field applies for Integer and Double variables. It is not application for string type variables.</td>
</tr>
<tr>
<td>String size</td>
<td>Defines the length of a valid string.</td>
</tr>
<tr>
<td>String range</td>
<td>Defines the range of valid values for a string.</td>
</tr>
<tr>
<td>Default value</td>
<td>Defines the default value for a variable displayed in the Knowledge Script Properties dialog.</td>
</tr>
<tr>
<td>No quotation required</td>
<td>For scripts written in Perl, select this option if the script uses an associative array.</td>
</tr>
</tbody>
</table>
Action Tab, Script Properties dialog box

Use the Action tab in the Script Properties dialog box (choose View > Properties > Action) to add or modify actions associated with events raised by the Knowledge Script.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action List</td>
<td>List of actions initiated by events raised by the script.</td>
</tr>
<tr>
<td>New</td>
<td>Click to add a new action.</td>
</tr>
<tr>
<td>Modify</td>
<td>Select an action in the Action List, and click to modify the properties.</td>
</tr>
<tr>
<td>Delete</td>
<td>Select an action in the Action List, and click to delete.</td>
</tr>
</tbody>
</table>
Add New/Modify Action dialog box

Use the New/Modify button in the Action tab in the Script Properties dialog box (choose View > Properties > Action > New or Modify) to define the properties of a new or existing action.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Select an action from the list.</td>
</tr>
<tr>
<td>Location</td>
<td>Select the computer from which the action is initiated:</td>
</tr>
<tr>
<td></td>
<td>• Management server</td>
</tr>
<tr>
<td></td>
<td>• Managed client</td>
</tr>
<tr>
<td></td>
<td>• Proxy (another computer running the AppManager agent)</td>
</tr>
</tbody>
</table>
## Type
Select the type of event that initiates the action:
- **New Event.** The action is initiated when a new event is raised.
- **Repeated Event.** The action is initiated when a duplicate event is raised a specified number of times. Specify that number in the field for this selection.
- **Event Down.** The action is initiated when the event condition no longer exists (for example, when the transfer of bits per seconds is back above the minimum threshold).

## Schedule
Select a schedule for the action. The schedules listed here correspond to the action schedule types defined in the AppManager repository preferences.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Select the type of event that initiates the action:</td>
</tr>
<tr>
<td></td>
<td>• New Event. The action is initiated when a new event is raised.</td>
</tr>
<tr>
<td></td>
<td>• Repeated Event. The action is initiated when a duplicate event is raised a specified number of times. Specify that number in the field for this selection.</td>
</tr>
<tr>
<td></td>
<td>• Event Down. The action is initiated when the event condition no longer exists (for example, when the transfer of bits per seconds is back above the minimum threshold).</td>
</tr>
<tr>
<td>Schedule</td>
<td>Select a schedule for the action. The schedules listed here correspond to the action schedule types defined in the AppManager repository preferences.</td>
</tr>
</tbody>
</table>
Migrate Summit Scripts dialog box

Use the Migrate Summit Scripts dialog box (choose View > Tools > Migrate) to set the directory paths for migrating Summit BasicScript Knowledge Scripts to the .qml format.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory containing Summit scripts to migrate</td>
<td>Path to the Summit BasicScript (.ebs) files you want to migrate.</td>
</tr>
<tr>
<td>Directory to save migrated scripts</td>
<td>Path to where you want the migrated script (.qml) saved.</td>
</tr>
</tbody>
</table>

Set Debuggers dialog box

Use the Set Debuggers dialog box (choose View > Tools > Set Debuggers) to specify the path to the debugger you want to use.
If you have used the default installations, the paths to the debuggers will be:

<table>
<thead>
<tr>
<th>Scripting Language</th>
<th>Debugger</th>
<th>Default Path to Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit BasicScript</td>
<td>NetIQ Knowledge Script Editor</td>
<td>C:\Program Files\NetIQ\AppManager\bin\niqnedit.exe</td>
</tr>
<tr>
<td>VBScript</td>
<td>Microsoft Script Host</td>
<td>C:\WINNT\system32\cscript.exe</td>
</tr>
</tbody>
</table>

The Microsoft Windows Script Host requires a command line argument, either //D or //X, to run in debug mode. You must enter one of these arguments in the third field of the Set Debuggers dialog box. If you use the //D parameter, the debugger will only kick in if an error occurs. Using the //X parameter starts the debugger and puts a breakpoint on the first executable line of script code.
Script Check-in dialog box

Use the Developer Console Logon dialog box (choose Tools > Check in Knowledge Script) to log on to the AppManager repository.

![NetIQ AppManager Developer Console Logon dialog box](image)

<table>
<thead>
<tr>
<th>For</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type the user name of the SQL Server login account used to access the AppManager repository.</td>
</tr>
<tr>
<td>Password</td>
<td>Type the password for the SQL Server login account.</td>
</tr>
<tr>
<td>Server</td>
<td>Type the name of the SQL server that manages the AppManager repository. When specifying a computer name, you can enter the Windows NT computer name or the IP address. For example, to specify a named instance on SQL Server 2000, you can enter 10.1.10.43\INST1.</td>
</tr>
<tr>
<td>Repository</td>
<td>Type the name of the AppManager repository you want.</td>
</tr>
</tbody>
</table>

Convert To Knowledge Script dialog box

This tool (choose Tools > Convert Perl script to KS) steps through a Perl script searching for lines of code that need to be converted to use AppManager-compatible constructs.
## Developing Custom Knowledge Scripts

**Item** | **Description**
---|---
Original Script field | Existing code.
Suggestion field | Suggested additions where appropriate code does not exist, or suggested substitutions for existing code. Select a line of suggested code and click **Change** to make the substitution. Click **Ignore** to skip the change and move to the next line in the original script. Double-click a suggested line to open a dialog box that displays the line in its entirety.
Perl Conversion Options

Use the **Perl Conversion Options** dialog box (choose **Tools > Perl Conversion Options**) to take an existing Perl script and automatically generate AppManager-specific callback functions that enable the script to send events and data to the AppManager repository.

<table>
<thead>
<tr>
<th>This option</th>
<th>Makes this conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert ‘die’ to</td>
<td>Perl scripts use <code>die</code> to exit on an unrecoverable error. <code>die</code> quits the program and prints the line number to STDERR.</td>
</tr>
<tr>
<td></td>
<td>The <strong>No Change</strong> option uses the default <code>die</code> behavior, and nothing is returned to the AppManager repository.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Abort</strong> option converts <code>die</code> to <strong>AbortScript</strong>, which terminates the script and stops it in an error state.</td>
</tr>
<tr>
<td></td>
<td>The <strong>Event</strong> option converts <code>die</code> to <strong>CreateEvent</strong>, which raises an event before the script terminates.</td>
</tr>
</tbody>
</table>
## This option | Makes this conversion
---|---
Convert 'exit' to | The Perl command `exit` quits the program with a return value. Since `exit` is not allowed in a Knowledge Script because it quits the agent program, the default behavior is to convert `exit` to `die`. The `Abort` option converts `exit` to `die` when the return value is zero, but makes an additional call to `AbortScript` when the return value is not zero (programs typically return a non-zero value to indicate an error). The `Event` option converts `exit` to `die` when the return value is zero, but makes an additional call to `CreateEvent` when the return value is non-zero (programs typically return a non-zero value to indicate an error). The `Data` option converts `exit` to `CreateData`, with data values equal to `exit`’s return value, regardless of whether the value is zero, and then uses `die` to quit the script.

Allow job retries after an 'AbortScript' | The default behavior for `AbortScript` is to error out and stop the job. This option indicates that AppManager should periodically try to restart the job.

Convert STDERR | By default, `STDERR` from the script is not returned to the Operator Console. This option allows `STDERR` to be returned as an event by redirecting `print` to a temporary file whose contents are returned to the AppManager repository as an event at the end of execution. Use any positive value for this option. The value for this option takes precedence over the value for the All Events option. If this option is not selected, `STDERR` is not returned as an event.
By default, STDOUT from the script is not returned to the Operator Console. This option allows STDOUT to be returned as an event by redirecting print to a temporary file whose contents are returned to the AppManager repository as an event at the end of execution.

Use any positive value for this option. The value for this option takes precedence over the value for the All Events option.

If this option is not selected, STDOUT is not returned as an event.

If the Perl script writes data to a log file, this option returns the contents of that file as an event.

Use any name for the file.

This file contains the original script plus the line numbers where problems might occur in the conversion. Alongside each line number are possible versions of the line:
- The original line
- The converted line
- Alternative conversions of the line

The conversion tool reads each of the possible line versions and gives the user the option to select one of them or to make the changes manually.
Perl Syntax Check

Use the Perl Syntax Check (choose Tools > Perl Syntax Check) dialog box to check the syntax of a Perl script.

<table>
<thead>
<tr>
<th>Line #</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>Error</td>
<td>No such class EQ_EVENT at line 47, near &quot;your_EQ_EVENT&quot;</td>
</tr>
<tr>
<td>47</td>
<td>Error</td>
<td>syntax error at line 47, near &quot;your_EQ_EVENT&quot;</td>
</tr>
<tr>
<td>54</td>
<td>Error</td>
<td>No such class AH_PIN at line 54, near &quot;your_AH_PIN&quot;</td>
</tr>
<tr>
<td>54</td>
<td>Error</td>
<td>syntax error at line 54, near &quot;your_AH_PIN&quot;</td>
</tr>
<tr>
<td>66</td>
<td>Error</td>
<td>BEGIN not safe after error-computation aborted at line 66</td>
</tr>
</tbody>
</table>

**To**   **Do this**

Dismiss the dialog box  Click **Close**.

Highlight the line of code containing the error  Select the error message from the list, and click **Go to**. The corresponding line of code is highlighted in the Developer’s Console. The script code is displayed in the Perl Script (Read-only) view.

Edit the code to fix the error  Select the error message from the list, and click **Edit**. The dialog box is dismissed, the corresponding line of code is highlighted in the Developer’s Console, and the script is displayed in the Edit view. You can then make the necessary changes to your code.

To continue with the syntax check, click **Tools > Perl Syntax Check**.
Appendix B

Perl Development

Due to the AppManager agent architecture on UNIX (a multi-threaded application which hosts multiple Perl engines on several threads), one should not use certain Perl language constructs. Nor should one use certain system functions within C/C++ based managed objects. Here is a list of these restrictions. Workarounds are provided in some cases.

In the AppManager UNIX agent architecture, each Perl job is executed by a Posix thread within the UNIX agent. Perl Knowledge Scripts and managed objects should not perform any operations that are not multi-thread safe. In the V1 UNIX agent, each Perl job is executed by one separate Perl engine and there is no limitation on concurrent Perl engines. In the V1.1 UNIX agent, each Perl job is still executed by one separate Perl engine—however, for this agent there is a limitation of concurrent Perl engines that is configurable via the `nqmcfg.xml` file or the `nqagt` command line option. The V2 UNIX agent uses a pool of Perl engines where one Perl engine may execute more than one Perl jobs.

The Perl engine bundled with all of the AppManager UNIX agents is version 5.6.1.

Compiling your Perl modules

You must compile your Perl module or C/C++ based managed object with the exact same Perl engine that is bundled with the UNIX agent (located in `/code/perl/5.6.1`). You will need to compile separate objects for Solaris, Linux, HPUX, and AIX.

**Warning** If you do not use the same perl engine (e.g., you compiled
Developing Custom Knowledge Scripts

your Perl module with single threaded Perl engine and then use it under UNIX agent which has multi-threaded Perl engine), at runtime you may observe “reallocation error” or “unsolved symbol ...” in the \textit{nqmlog} file.

The correct multi-threaded Perl engines (5.6.1) are available as:

\begin{itemize}
\item perl-solaris.tar.Z
\item perl-linux.tar.Z
\item perl-hpux.tar.Z
\item Lperl-aix.tar.Z
\end{itemize}

The Perl engine must be installed under:

\texttt{.../opt/netiq/UnixAgent/lib/.....}

You should compile your code with the exact same compiler/linker options that were used to compile the Perl engine. The easiest way to achieve this is to use the Perl way to compile your Perl module, i.e., use the correct Perl to generate a \texttt{Makefile} based on \texttt{Makefile.PL}. This means the compilation of your perl module should use the same compiler and compilation/linking options as the Perl engine. Refer to:

\begin{itemize}
\item http://www.perldoc.com/perl5.8.0/pod/perlxstut.html
\item http://www.perldoc.com/perl5.8.0/lib/ExtUtils/MakeMaker.html
\end{itemize}

\section*{Perl best practices}

\begin{enumerate}
\item Do not call \texttt{fork()}, \texttt{exec()}, \texttt{system()}.

Any \texttt{fork}, \texttt{exec}, or \texttt{system} operation from a thread within a multi-threaded application can cause application deadlock.

\textbf{Workaround:} Use the \texttt{ExecCmd} callback function. \texttt{ExecCmd} is programmed to serialize concurrent \texttt{fork}, \texttt{exec}, and \texttt{system} calls and therefore avoid deadlock. Also see the note about I/O redirection in issue #11.

\item Do not use back quotes to call a command (`\texttt{CMD}`).

The same reason as #1.
\end{enumerate}
Workaround: Use the `ExecCmd` callback function, which is programmed to serialize concurrent calls to avoid deadlock. Also see #11.

3 Do not call `chdir()` and `chroot()`

There is only one current directory per application. Changing the current directory of an application from one thread may cause problems for other threads within the same application. Also see issue #17.

Workaround: Either of the following two will do.

1. Instead of `ExecCmd("cmd")`, use `ExecCmd("cd $dir ; cmd")`. That is, replace
   
   ```
   chdir $dir;
   .....;
   ExecCmd("cmd");
   ```
   with
   
   ```
   ExecCmd("cd $dir ; cmd")
   ```
   
   The entire command execution within `ExecCmd` is to change to the directory specified by `$dir` and then execute `cmd`. It is also OK to start background processes by using this method. For example, `ExecCmd("cd $dir ; cmd &")`.

2. Use the `ExecCmd` callback function to invoke an external program (shell script, for example) that performs the `cd` operation.

4 Do not set up any signal handling routines, including `alarm()`, in Perl.

The UNIX agent is based on Java, which already catches quite a few signals. In addition, the UNIX agent itself also catches a few signals. Any operation to modify any signal handling routines can cause UNIX agent deadlock.
5 Redirect stdout or stderr to the Nqext::CreateEvent callback function, or to /dev/null, or to an individual file.

Any stdout or stderr will be lost because the UNIX agent runs as a daemon process. One should also replace print or printf with other functions.

6 Avoid calling sleep().

Invoking sleep() ties the current job to a Perl engine that could otherwise execute other jobs (in the case of pool of Perl engine).

Workaround: Avoid sleep() if possible. If you must use it, specify a short period for sleep() (e.g., less than 5 seconds). If one expects to sleep for a long time, remember the state of the job/Knowledge Script and re-check the status in the next job iteration.

You can also wrap the sleep logic into an external script and have Exec callback execute the script asynchronously.

7 The END block in Knowledge Scripts or in Perl modules for V2 (or greater) UNIX will not be executed

Having BEGIN or INIT in either Knowledge Scripts or Perl modules is OK. Also note that you can have initialization code defined in the boot section within XS based code.

Workaround: None.

8 Take advantage of Perl features, as much as possible. for example:

● hash variables
  To avoid memory leaks, before the end of each KS iteration, one must deallocate each hash variable, i.e.,

  %hash_var = ();
  or
  undef %hash_var;

● Regular expression, pattern matching (instead of invoking grep command via ExecCmd)
● Perl built-in constructs for file operations, input/output operations, directory reading operations, system interactions, networking, IPC, information from system files, etc.
● Various Perl modules

9 Use the `ExecCmd()` callback function sparingly.

Minimize calls to this callback because it is serialized. All current running Perl jobs within the UNIX agent have to be suspended during the process of `ExecCmd`. Instead of calling `ExecCmd`, you should consider using Perl language constructs to perform operations wherever possible.

10 Do not open a pipe with the Perl construct `open()`

The Perl construct `open(F, "CMD |")` opens a pipe to command CMD and read the stdout from CMD. For the same reason that one should not use fork(), exec(), or system(), one should not create processes via the construct `open()`.

**Workaround:** Replace the following code

```
open(F, CMD |);
...
close(F);
```

with

```
$f = Nqext::ExecCmd(CMD, 1);
open(F, $f);
close F;
unlink F;
```

Redirect `stdout` and `stderr` if you start a background process with the `ExecCmd()` callback function.

If `stdout` and `stderr` are not redirected, the `ExecCmd()` callback function will hang forever.

**Workaround:** You should always redirect `stdout` and `stderr` of
any background process to /dev/null (if stdout and stderr are not needed) or to files (if they are needed). For example, do either of the following:

# start script.sh in background
ExecCmd(script.sh > /dev/null 2> /dev/null &);

or

ExecCmd(wrap_script.sh);  # start wrap_script.sh

where wrap_script.sh contains

#!/bin/sh
...
script.sh > /dev/null 2> /dev/null &
# wrap_script.sh continues even if script.sh
# has not terminated.

You can also redirect to a temporary file.

Note If you redirect stdout and stderr to /dev/null, then
ExecCmd will not be able to return stdout or stderr from the command.

11 Unless you are within an eval, do not escape any Perl scripts run by
the AppManager UNIX agent. This includes both Knowledge
Scripts and the managed objects in Perl modules. That is, do not use
the die, exit, or croak commands. They will (sometimes, but not always) exit the entire UNIX agent.

Workaround: Use AbortScript.

12 You can overwrite any section of Makefile generated from
Makefile.PL

Most of time the Makefile generated from the Perl Makefile.PL
has everything you need. But sometimes it does not, especially for
AIX.

Workaround: Use the Perl module ExtUtils::MakeMaker to
overwrite any section within the Makefile. For example, you can
overwrite the postamble section of the Makefile with the following from within Makefile.PL:

```perl
sub MY::postamble {
  '
  $(MYEXTLIB): sdbm/Makefile
  cd sdbm && $(MAKE) all
  ';
}
```


13 The `ExecCmd` callback function does not provide exit code from the command just executed.

**Workaround:** Replace `ExecCmd(cmd);` with

```perl
ExecCmd(cmd ;
  echo $?);
```

14 If you are using C/C++ to develop your managed objects, be aware that any operation should be thread-safe. Use an appropriate mutex mechanism (e.g., `pthread_mutex_lock`, `pthread_mutex_unlock`) to protect the critical section.

15 Never hard-code an output file name. If you do, and then two such jobs (using the same output file name) run concurrently, you will have problem. You could use callback `GetTempFileName` to get a unique file name, or make the file name be a function of something unique, such as `jobid`. 
16 {A super set of #3} Unless you know what you are doing, do not call functions that can affect process-scope state, such as

- `setpgrp()` -- sets the process group for a specific PID
- `setpriority()` -- sets the current priority for a process
- `umask()` -- sets the umask for the process
- `chdir()` -- changes the current working directory for the process.
  
  See #3 for workaround.
- `chroot()` -- changes the root directory for the process.

17 {super set of #6} Avoid issuing any long blocking calls, such as reading from a socket, etc. This again would tie the current job to a Perl engine that could otherwise execute other jobs in the case of pool of Perl engine.

**Workaround:** If the KS can perform other useful computations while the long API is in execution,

- Use an asynchronous version of the API, if available, instead of using the sync version.
- Create a separate process (ExecCmd) to perform the blocking operation

18 On AIX, use the `slibclean` command to clean up any modules (including any dynamically loaded modules, `.so`) cached in the kernel before starting the UNIX agent with an updated Perl module. In C, `unload()` is the API for this purpose. See http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/cmds/aixcmds5/slibclean.htm

    and

    http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprgd/kernextc/kernex_binding.htm#A23C0f1a0

    for details.

19 Avoid calling C/C++ functions that are thread-unsafe. Depending on the platform, thread-safe functions usually have the name
appended with _r. Check the manual pages for details. The following are a few important ones for Solaris:

<table>
<thead>
<tr>
<th>Thread-unsafe functions</th>
<th>Thread-safe functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>localtime ()</td>
<td>localtime_r()</td>
</tr>
<tr>
<td>localtime()</td>
<td>localtime_r()</td>
</tr>
<tr>
<td>get{gr,host,net,proto,serv, pw}*()</td>
<td>get{gr,host,net,proto,serv,pw}*_r()</td>
</tr>
<tr>
<td>readdir()</td>
<td>readdir_r()</td>
</tr>
<tr>
<td>rand()</td>
<td>rand_r()</td>
</tr>
<tr>
<td>srand()</td>
<td>N/A</td>
</tr>
</tbody>
</table>

20 Do not call perl built-in functions that are not thread-safe. In Perl 5.6.1 on Solaris, the following are not thread safe (list is not all-inclusive):

localtime(), gmtime(), get{gr,host,net,proto,serv,pw}*(), readdir().

Note Perl functions rand and srand invoke rand48(3C) and srand48(3C), which are thread-safe.
Index

A
AbortScript callback function 232
AbortScript() callback function 292
action 323
action schedule 323
action scripts 53, 133
ending actions 137
events without actions 136
invoking actions 136
modifying 133
Perl 185
Summit BasicScript 161
VBScript 133
setting up to perform actions 134
size limit 54
XML messages 137
Action_Messenger.qml script 162
Action_MessengerEx.qml script 183
Action_UXCommand.qml script 192
Action_UXCommandEx.qml script 199
Action_WriteToFile.qml script 140
Action_writeToFileEx.qml script 158
AKP_NULL 52
AKPID 48
AppManager
agent 17
architecture 40
management server 41
management server components 42
repository 42
version number 23
AppManager agent 43

B
BasicScript 23
debugging 320
setting debuggers 316

C
callback functions 25
callbacks
Perl 291
AbortScript() 292
CounterValue() 294
CreateData() 295
CreateEvent() 298
ExecCmd() 301
ExportData() 303
ExportHugeData_pl() 305
GetJobID() 306
GetMachName() 307
GetScriptInterval() 308
GetTempFileName() 309
ImportData() 310
ImportHugeData_pl() 312
IterationCount() 313
Summit BasicScript and VBScript 229
AbortScript 232
CreateData 234
callbacks
Summit BasicScript and VBScript 229
CreateEvent 237
DataHeader 240
DataLog 242
DynaCollectData 244
DynaDataLog 246
GetAgentInfo 248
GetContextEx 249
GetJobID 252
GetKPIInterval 253
GetMachName 254
GetProgID 255
GetSecurityContext 256
GetTempFileName 257
GetVersion 258
Item 260
ItemCount 262
IterationCount 264
LongDataHeader 265
LongDataLog 267
LongDynaDataLog 268
MCAbort 270
MCEnterCS 271
MCExitCS 272
MCGetMOID 273
MCVersion 275
MCWaitForObject 276
MCWaitForObjectEx 278
MSActions 280
MSLongActions 284
NQSleep 285
QTrace 286
WaitForObject 288
choosing a scripting language 69

COM objects 25
converting older Knowledge Scripts to qml format 31
corrective actions 324
counters, Performance Monitor 87
CounterValue( ) callback function 294
CreateData callback function 234
CreateData( ) callback function 295
CreateEvent callback function 237
CreateEvent( ) callback function 298
creating new scripts 53

D
data points 324
data stream 324
DataHeader callback function 240
DataLog callback function 242
debuggers, setting 316
debugging 315
  BasicScript 320
  Perl 321
  prepend files 318
  Summit BasicScript 320
  VBScript 320
  where to debug scripts 316
default properties, Knowledge Scripts 56
default schedule 21
default scripting language 35
developer 324
developer license 30
Developer’s Console 30
dialog boxes 327
  Add New Object Type 334
  Add New/Modify Action 341
  Add/Modify Parameter 338
Developer’s Console 30
dialog boxes 327
   Advanced Schedule
      Configuration 336
   Convert To Knowledge Script
      345
   Migrate Summit Scripts 343
   Perl Conversion Options 347
   Perl Syntax Check 350
   Script Check-in 345
   Script Editor Options 327
   Script Properties dialog box
      Action Tab 340
      Default Schedule tab 335
      Header tab 329
      Object Types tab 331
      Parameters tab 337
   Set Debuggers 343
Edit view 36
   editing scripts 31
   opening 31
   opening files 32
   Properties dialog box 60
      Action tab 60
      Default Schedule tab 59
      Header tab 57
      Object Types tab 58
      Parameters tab 61
   Script Properties dialog box 24
   VBScript (Read-only) view 36
   views 35
Developer’s tools 30
dialog boxes
   Script Properties dialog box
      Object Types tab 331
discovery scripts 51, 53
DO_DATA 52
DO_EVENT 52
documentation
   additional 12
   conventions 11
   suggestions 15
DynaCollectData callback function 244
DynaDataLog callback function 246
E
ebs extension 24
   error icon, blinking 23
   event 324
   ExecCmd() callback function 301
   executable script 23, 40
   ExportData() callback function 303
   ExportHugeData_pl() callback function 305
G
generated script 29, 324
GetAgentInfo callback function 248
GetContextEx callback function 249
GetJobID callback function 252
GetJobID() callback function 306
GetKPInterval callback function 253
GetMachName callback function 254
GetMachName() callback function 307
GetProgID callback function 255
GetScriptInterval() callback function 308
GetSecurityContext callback function 256
GetTempFileName callback function 257
GetTempFileName() callback function 309
Developing Custom Knowledge Scripts

GetVersion callback function 258

I
Icon Manager 31
ImportData() callback function 310
ImportHugeData_pl( ) callback function 312
input validation 21
install scripts 53
Item callback function 260
ItemCount callback function 262
IterationCount callback function 264
IterationCount() callback function 313

J
job 324

K
Knowledge Script 324
code 23
definition 324
version number 23
Knowledge Script Editor 31
Knowledge Script Group 325
Knowledge Script Group member 325
Knowledge Script jobs 17
Knowledge Script name 49
Knowledge Script Properties
   Schedule tab 21
   Values tab 21
Knowledge Script Properties dialog box 20
Knowledge Scripts
   architecture 17
   checking in 33, 34
   checking out 32
code component 25
components 23
configuring a job 17
converting to qml format 31
copying 33
creating new script 53
debugging 315
elements 49
final, generated script 28
how AppManager processes scripts 23
job 23
naming 49
non-code XML elements 24
opening files 32
renaming 33
running 43
sample 26
saving 34
setting default properties 56
KS_INIT() 37

L
license, developer 30
location, sample scripts 38
LongDataHeader callback function 265
LongDataLog callback function 267
LongDynaDataLog callback function 268

M
managed client 17, 325
managed computer 41, 43
managed computer components 43
managed object methods 25
managed objects 43, 325
management server 41, 42
management service 43, 325
MCAbort callback function 270
MCEnterCS callback function 271
MCExitCS callback function 272
MCGetMOID callback function 273
MCVersion callback function 275
MCWaitForObject (Summit BasicScript only) 276
MCWaitForObjectEx (Summit BasicScript only) 278
modifying action scripts 133
Perl 185
Summit BasicScript 161
VBScript 133
modifying monitoring scripts 71
Perl 117
Summit BasicScript 91
VBScript 71
monitoring job 325
monitoring policy 325
monitoring scripts
modifying 71
Perl 117
Summit BasicScript 91
VBScript 71
MSActions callback function 280
MSLongActions callback function 284
N
naming Knowledge Scripts 49
non-code XML 24
normal scripts 53
NQSleep callback function 285
NT_CpuLoaded.qml script 91
NT_CpuLoadedEx.qm script 111
O
object type checking 20
object type variable 23
object type, assigning 50
Object Types tab, Script Properties dialog box 331
ObjType value 47
Operator Console 17, 42
configuring a job 17
P
parameter 325
parameter non-code XML elements 47
Performance Monitor counters 87
Perl 24
debbuging 321
Perl modules 25
prefix, Knowledge Script name 49
prepend files 318
process 325
Properties dialog box 325
properties, running script 56
Q
qml extension 24
QTrace callback function 286
R
report scripts 53, 204, 325
about 204
adding variables 224
altering value set of an existing script 207
copying 207
discovering the Report agent 205
manipulating data 224
modifying Event Script Parameters 217
Developing Custom Knowledge Scripts

modifying non-code XML elements 219
modifying report settings 217
modifying script properties 221
modifying the code 223
releasing references to created objects 227
saving 227
selecting aggregation interval 217
selecting data streams 208
selecting days of the week 216
selecting the way data is presented 213
selecting time range 214
setting a new time range 227
repository host 325
repository, AppManager 42
resource object types 50
running script, properties 56

S
sample script listings 71
Action_Messenger.qml 162
Action_MessengerEx.qml 183
Action_UXCommand.qml 192
Action_UXCommandEx.qml 199
Action_WriteToFile.qml 140
Action_writeToFileEx.qml 158
NT_CpuLoaded.qml 91
NT_CpuLoadedEx.qml 111
Samples_FilesOpen.qml 71
Samples_FilesOpenEx.qml 86
Samples_HTTPHealth.qml 117
Samples_HTTPHealthEx.qml 130
Sample scripts 68
checking in 68

location 38
Samples_FilesOpen.qml script 71
Samples_FilesOpenEx.qml script 86
Samples_HTTPHealth.qml script 117
Samples_HTTPHealthEx.qml script 130
Script Parameters 52, 326
deciding on 52
defining 63
range of possible values 21
user-definable 52
Script Properties dialog box 326
Object Types tab 331
scripting language, choosing 69
scripting language, default 35
scripts
action 53
discovery 53
install 53
monitoring 53
normal 53
report 53
server group 326
setting debuggers 316
Summit BasicScript 23
debugging 320
setting debuggers 316

T
target computer 18, 326
technical support 15
testing and debugging 315
thread 326
threshold 326
TreeView pane 19
type checking 51
UNIX, managing computers 24
user 326
user interface, AppManager 42
user-definable Script Parameters 52

VBScript 24
  debugging 320
  setting debuggers 316
version 326

WaitForObject callback function 288

XML (Read-only) view 36