



Note: When copying code examples from this document, you may experience formatting problems. Please copy code examples from the online version at:

<http://download.dartware.com/docs/DevGuide/>

Developer Guide

Version 5.4

Python Version 2.6

June 2010

Table of Contents

Introduction	5
Software License Agreement	6
Creating Your Own Probes	8
Anatomy of a Probe	10
The <header> Section	13
The <description> Section	18
The <parameters> Section	19
The <autorecord> section	22
Automatically-Recorded Data Values	23
Probe Status Windows	27
IMML - InterMapper Markup Language	28
Probe Comments	30
Built-in Custom Probe Variables	31
SNMP Probes	40
The <snmp-device-variables> Section	43
The <snmp-device-thresholds> Section	48
The <snmp-device-variables-ondemand> Section	50
The <snmp-device-display> Section	57
The <snmp-device-alarmpoints> Section	58
Alarm Point File Format	61
Probe Calculations	68
Probe Properties	79
Specifying SNMP OIDs in Custom Probes	80
Enumerated Values In SNMP Probes	85
About Custom SNMP Trap Probes	86
The <snmp-device-variables> Section For Traps	87
The <snmp-device-display> Section	91
Trap Viewing and Logging	92
Example - Trap Viewer Probe	93
The Dartware MIB	98
TCP Probes	101
The <script> Section	104
The <script-output> Section	112
TCP Probe Command Reference	113

Measuring TCP Response Times.....	126
Example TCP Probe File.....	128
Command Line Probes.....	130
The <command-line> Section.....	134
The <command-exit> Section.....	136
The <tool> Section.....	137
The <command-display> Section.....	142
Command Line Probe Example.....	143
InterMapper Python Plugins.....	145
Nagios Plugins.....	147
Nagios Plugin Example.....	150
NOAA Weather Probe Example.....	152
Installing and Reloading Probes.....	156
Modifying Built-in Probes.....	157
Sharing Probes.....	158
Troubleshooting Probes.....	159
Errors with Custom Probes.....	160
Debugging with the SNMPWalk Command.....	162
Using the SNMPWALK -o Option.....	165
InterMapper HTTP API.....	169
Importing & Exporting Files.....	170
Importing & Exporting Tables.....	173
Acknowledging with HTTP.....	175
HTTP API Scripting Examples.....	176
Retrieving Collected Data.....	181
InterMapper Database Schemas.....	181
Customizing Web Pages.....	182
Target Files.....	183
Template Files.....	185
Directives.....	186
Quoted Links.....	189
Macro Reference.....	190
Folder Structure.....	196
MIME Types.....	198
Tip for Calling Charts.....	198

Command-line Options for InterMapper.....	199
Index	201

Chapter 1

Introduction

Documents InterMapper and InterMapper RemoteAccess 5.4
Document built: 5/4/2011 9:31 AM

InterMapper is a network monitoring and alerting program. It continually tests routers, servers, hubs, and other computer devices that are attached to your network. If InterMapper detects a failure, it sends notifications to one or more individuals via sounds, e-mail, pagers, or by running a program to correct the problem.

Use this manual to learn about the tools you can use to customize how InterMapper monitors the network, and to display data about those results.

[Customizing InterMapper's Probes \(Pg 8\)](#)

Explains how to create your own custom probes, and how to configure them to add power and flexibility to your network monitoring.

[InterMapper HTTP API \(Pg 169\)](#)

Explains InterMapper's HTTP API and how to use it to import and export files and tables, and to use your own scripts to create acknowledgements.

[Customizing Web Pages \(Pg 182\)](#)

Explains how you change the look and function of the web pages available from InterMapper's built-in Web server.

[Retrieving Collected Data \(Pg 181\)](#)

A look at the InterMapper Database, with links to useful resources.

Please give us comments at the address listed below. Thanks!

[Dartware, LLC](#)

[InterMapper Feedback](#)



Software License Agreement

This is a legal agreement between you and Dartware, LLC covering your use of InterMapper®, InterMapper RemoteAccess™, and other Dartware products and the associated documentation (the "Software"). Be sure to read the following agreement before using the Software. BY USING THE SOFTWARE (REGARDLESS IF YOU HAVE REGISTERED THE SOFTWARE OR NOT), YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE AND DESTROY ALL COPIES IN YOUR POSSESSION.

The Software is owned by Dartware, LLC and is protected by United States copyright laws and international treaty provisions. It is licensed to you. Therefore, you must treat the Software like any other copyrighted material (e.g., a book or musical recording).

License

This license allows you the right to use one copy of the Software on a single computer. You may make one (1) copy of the software as a backup. You may not network the Software or otherwise use it or make it available for use on more than one computer at the same time. You may not rent or lease the Software, nor may you modify, adapt, translate, decompile, or disassemble the Software. If you violate any part of this agreement, your right to use this Software terminates automatically and you must then destroy all copies of the Software in your possession.

Disclaimer of Warranty

The Software and its related documentation are provided "AS IS" and without warranty of any kind. The person using the software bears all risk as to the quality and performance of the software. If you paid for the product, and within 30 days find that it doesn't do what you want, then you can notify Dartware, LLC and your money will be refunded and your license canceled. Dartware, LLC hereby disclaims all warranties relating to this software, whether express or implied, including without limitation any implied warranties of merchantability or fitness for a particular purpose. Dartware, LLC will not be liable for any special, incidental, consequential, indirect or similar damages due to loss of data or any other reason, even if Dartware, LLC or their agent has been advised of the possibility of such damages. In no event shall Dartware, LLC be liable for any damages, regardless of the form of the claim.

US Government

Government End Users: If you are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i) if the Software is supplied to the Department of Defense (DoD), the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and (ii) if the

Software are supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR. The manufacturer is Dartware, LLC, 66-7 Benning Street, West Lebanon, NH 03784 USA.

This Agreement shall be governed by the laws of the State of New Hampshire. If for any reason a court of competent jurisdiction finds any provision of the Agreement, or portion thereof, to be unenforceable, that provision of the Agreement shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this Agreement shall continue in full force and effect.

Chapter 2

Creating Your Own Probes

For many Internet services, simply "pinging" a device is not a sufficient test of whether it is operating correctly. InterMapper has a number of built-in probes that can test different aspects of a device's operation, whether it's a web server, router, database, LDAP server etc.

However, InterMapper's built-in probes may not test the kinds of devices you want to monitor, or may not test them ways that are most useful to you. In such a case, you can create your own probes. InterMapper's probes are defined by *probe files*, simple text files that can be duplicated and modified using any standard text editing utility. When you create your own probe, it becomes a first-class citizen and appears in the Probe Configuration window along with the built-in probes.

What is a probe?

A probe is a text file that specifies the way in which InterMapper tests a device. It is essentially a plug-in. All of InterMapper's probes follow the same logic:

- The probe sends one or more queries, as SNMP requests, UDP or AppleTalk datagrams or over a TCP connection to the device being tested.
- The device responds (or fails to respond).
- If there is no response, InterMapper sets the device's status to DOWN.
- InterMapper examines the response(s) from the device, and sets the device's status accordingly.

Parts of a Probe

All of the probe types listed below follow a similar structure, as outlined in [Anatomy of A Probe \(Pg 10\)](#), which explains the common sections of a probe, and the sections that are specific to a particular probe type.

Types of Probes

InterMapper has several kinds of probes. You can use InterMapper's built-in probes as-is, you can copy and modify them, or you can develop your own probes from scratch. These are the different probe types:

[SNMP Probes \(Pg 40\)](#)

InterMapper tests the device's status by sending SNMP queries and comparing the results to user-specified thresholds.

[SNMP Trap Probes \(Pg 86\)](#)

InterMapper can receive and process SNMP traps, and can set the status of a probe based on the contents of the trap's variables. You can create custom probes that alert you to problems in a certain device based on the contents of specific trap variables.

[TCP Probes \(Pg 101\)](#)

InterMapper establishes a TCP connection to a device. It then sends certain requests and evaluates the responses to determine the device's status. TCP probes uses the [TCP Probe Scripting language \(Pg 104\)](#) to create a sequence of commands, branching to different parts of the script under specified conditions.

[Command-line Probes \(Pg 130\)](#)

InterMapper can invoke a program or script (as if "from the command line"), and use the results to determine the device's status.

Big Brother Probes

Big Brother™ is an open-source network monitoring program. InterMapper listens for reports from Big Brother clients and sets the device's status accordingly.

It's fairly straightforward to modify the existing files to produce new probes. If you make a new probe file that might be useful, please consider sending it to us. See [Sharing Probes \(Pg 158\)](#) for more information.

Anatomy of a Probe

Probe files have several sections, several of which are common to all probe types. These are described below.

Each section contains a number of lines bracketed by

```
<section-name> ... </section-name>
```

You might want to open a separate window with the [example probe file \(Pg 128\)](#) while reading the subsequent sections.

Sections common to all probes

[The <header> Section \(Pg 13\)](#)

Learn about the <header> section of the probe definition, including how the probe is identified, how its name appears in the **Probe Type** menu, and the version numbering system.

[The <description> Section \(Pg 18\)](#)

The <description> section specifies the help text that appears in the Probe Configuration window and typically explains the function of the probe and the use of its parameters. Format the description using IMML, [InterMapper's Markup language \(Pg 28\)](#).

[The <parameters> Section \(Pg 19\)](#)

The <parameters> section defines the probe's parameters and how they are presented in the Probe Configuration window.

Display Sections

Each probe type has its own output section, which controls what appears in the device's Status window. In all probes, you format the appearance of Status window using IMML, [InterMapper's Markup language \(Pg 28\)](#).

Type-Specific Probe Sections

Each probe type has sections that are specific to that probe type.

Sections specific to SNMP Probes

Each custom SNMP probe has:

- [An <snmp-device-variables> section \(Pg 10\)](#) - Specifies which MIB variables to collect from the device.
- [An <snmp-device-thresholds> section \(Pg 48\)](#) - Specifies how those variables are to be tested against thresholds to determine the device's status.
- [An <snmp-device-display> section \(Pg 10\)](#) - Specifies what information about the device and its links should be displayed in the Status window.
- [The <snmp-device-properties> section \(Pg 10\)](#) - Specifies certain aspects of the SNMP queries sent to the device.
- [The <snmp-device-alarmpoints> Section \(Pg 58\)](#) - Allows you to define conditions under which the device changes a particular device state.

Sections specific to SNMP Trap Probes

SNMP Trap Probes do not probe devices - they simply wait for traps to arrive. They have some sections that are common to SNMP Probes, but work somewhat differently.

- [An <snmp-device-variables> section \(Pg 10\)](#) - Specifies which MIB variables to collect from the device. These are set automatically when a trap is received.
- [An <snmp-device-thresholds> section \(Pg 48\)](#) - Specifies how those variables are to be tested against thresholds to determine the device's status.

Sections Specific to TCP Probes

Each custom TCP probe can have:

- [A <script> section \(Pg 104\)](#) - This contains the probe's script. Scripts are written in InterMapper's [TCP Probe Scripting Language \(Pg 104\)](#) , which uses a sequence of commands and program flow that is similar to Basic. A rich set of [TCP commands \(Pg 113\)](#) is available.
- [A <script-output> section \(Pg 112\)](#) - This contains the probe's output, which is displayed in the Status window. Format the Status window using IMML, [InterMapper's Markup language \(Pg 28\)](#).

Sections Specific to Command-line Probes

Each command-line probe also has:

- [The <command-line> section \(Pg 134\)](#) - builds the command-line, specifying the command path and any parameters to be sent with the command.
- [The <command-exit> section \(Pg 136\)](#) - allows you to control the state of the device, depending on what is returned from the script.
- [The <tool> section \(Pg 137\)](#) - contains the code for a companion script that is executed by the probe.
- [The <command-display> section \(Pg 142\)](#) - allows you to control what appears in the device's Status window.

Other Information about Probes

[Comments \(Pg 30\)](#)

All probes use the same format for comments. The comment format is similar to HTML comments.

[Probe File Locations \(Pg 17\)](#) and [Probe File Names \(Pg 17\)](#)

To use probe files, you must import them, and should follow recommended naming conventions.

[Installing and Reloading Probes \(Pg 156\)](#)

Before a modification to a probe becomes effective, you need to click the **Reload probes...** button in the Set Probe window (circular arrow icon below the left pane of the window).

The <header> Section

The <header> section of a probe file contains a formal description of the probe. It is defined as follows, with each header property having a name and a corresponding value.

```
<header>
..[part name] = "[value]"
</header>
```

Note: Information by which InterMapper uniquely identifies the probe is contained in the header. While it is not required, Dartware strongly recommends that you follow [probe file naming conventions](#) that correspond to the unique identifier in the probe header.

Header Parts

type Describes the type of the probe file. InterMapper supports the following probe types:

- **builtin**
- **tcp-script**
- **custom-snmp**
- **custom-snmp-trap**
- **command-line**
- **cmd-line**

```
type = "cmd-line"
```

For custom SNMP probes, use the **custom-snmp** type.

For custom SNMP Trap probes, use the **custom-snmp-trap** type.

For custom TCP probes, use the **tcp-script** type.

For command-line probes, use the **command-line** or **cmd-line** type.

package The first part of the probe's full identifier. Typically, the package is made up of the domain name of the organization that created the probe, with the labels reversed.

For example, all probes created by Dartware, LLC the package statement is as follows:

```
package = "com.dartware"
```

The package part guarantees that different organizations can create probes without concern that their probe identifiers will conflict.

Note: The combination of [package].[probe_name] together form the probe's full identifier. (In the example below, the full identifier is "com.dartware.tcp.custom") By convention, the name of the file that holds the probe definition is the same as the probe's full identifier. This is not required, but it's a good idea. See [Probe File Locations \(Pg 17\)](#) and [Probe File Names \(Pg 17\)](#) below for more information.

probe_name The second part of the probe's full identifier. The probe_name may be whatever string the creating organization chooses.

human_name The string that appears in the left pane of the Probe Picker window. This string helps guides the user to select the probe for a particular device.

version Provides a means to determine which probe file is the current one. The format of the version is "#.#".

address_type A comma-separated list of one or more address types. InterMapper implements "IP" and "AT".

port_number The IP port used by this probe.

display_name Specify the display_name to use with this probe, using forward slashes to specify the heirarchy. To do this, add the following line to the <header> section of the probe:

```
display_name = "[top level]/[next level]/[next level]"
```

Example:

```
display_name = "Custom/Command-line"
```

url_hint Assign a double-click action within the probe (making it the pre-defined double-click action). To do this, add the following line to your <header> section of the probe:

```
url_hint = "url-to-invoke"
```

The following example would invoke the web browser to the device's IP address and port

```
url_hint = "http://${address}:${port}"
```

poll_interval Set the device's default poll interval to the indicated number of seconds. This overrides the map's default setting, and might be used to avoid too-frequent polling for (physical) devices that should not be polled too often.

Setting the poll interval *for the device* will override this poll_interval setting.

```
poll_interval = "300"
```

Sample Header Section

This is a sample header from the Custom TCP script.

```
<header>
  "type"           = "tcp-script"
  "package"        = "com.dartware"
  "probe name"     = "snmp.example"
  "human name"     = "Example SNMP probe"
  "display name"   = "Miscellaneous/Example SNMP Probe"
  "version"        = "1.0"
  "address type"   = "IP,AT"
  "port number"    = "161"
  "flags" = ""
</header>
```

Header Section of Custom SNMP Probes

The <header> section of the probe file is similar to the [standard <header> section](#), (Pg 13) with the following differences:

- The "type" for a Custom SNMP probe is "custom-snmp".
- There is a `FLAGS=xxx,xxx` command that takes the following optional items as parameters:
 - `NOLINKS` - InterMapper will not poll links (interfaces) with SNMP
 - `SNMPV2C` - InterMapper will use SNMPv2c to poll the device
 - `NOICMPFALLBACK` - InterMapper will not send an ICMP ping to a device if no SNMP responses return.
 - `MINIMAL` - the probe queries only its own (specified) variables.
 - `NTCREDENTIALS` - tells InterMapper to elevate its credentials using the username and password found in the NT Services server settings panel long enough to run the command line in the probe. (Windows only.)
 - `ALLOW-LOOPS` - In some network equipment, the indices for the ifTable and related tables do not proceed in the usual strictly increasing fashion, jumping around instead. Adding this flag to the header will instruct InterMapper to allow this situation. N.B. If the SNMP agent in your network device doesn't stop returning values when every item in the table has been read, set this flag to instruct InterMapper to loop over the table continuously until 5000 reads have occurred, at which point it will stop.
 - `IFINDEX-BUG` - Some network equipment will respond incorrectly to SNMP queries for the ifTable and related tables when InterMapper queries only certain entries in a sparse ifTable, rather than trying to query each possible index in turn. Add this flag to the header to instruct InterMapper to work around this situation rather than attempting to be efficient.
 - `LINKCRITICAL` - If a link of a device goes down and the flag is set, the device status changes to critical, and not to alarm, which is the default.
 - `NAGIOS3` - Use "flags" = "NAGIOS3" in the <header> of a command-line probe to indicate that the return value should be treated as a

Nagios Plugin. See the [Nagios Plugins](#) section.

Note: The **old_protocol** and **old_script** parts, added for backward compatibility, are deprecated, and are ignored in any older probes that use them.

Probe File Locations

Probes files are saved in the **Probes** folder of the **InterMapper Settings** folder.

Probe File Names

The probe files are named with two parts separated by a period ("."). The parts are:

- **package name** - must be unique for each organization creating probe files.

By convention, the package is composed of the organization's DNS domain name, with the segments reversed. Thus, the built-in probes for InterMapper all have a package of "com.dartware." Other organizations may create and share their own probes, since the file names will not collide.

- **probe name** - identifies the probe.

For example, the built-in Custom TCP probe is defined in the file named:

```
com.dartware.tcp.custom
```

The probe's package name and probe name are defined in the probe definition's header section. Dartware recommends that you name the file with the combination of the package name and probe name, as shown above.

The header section of the probe definition in the example above contains these two lines:

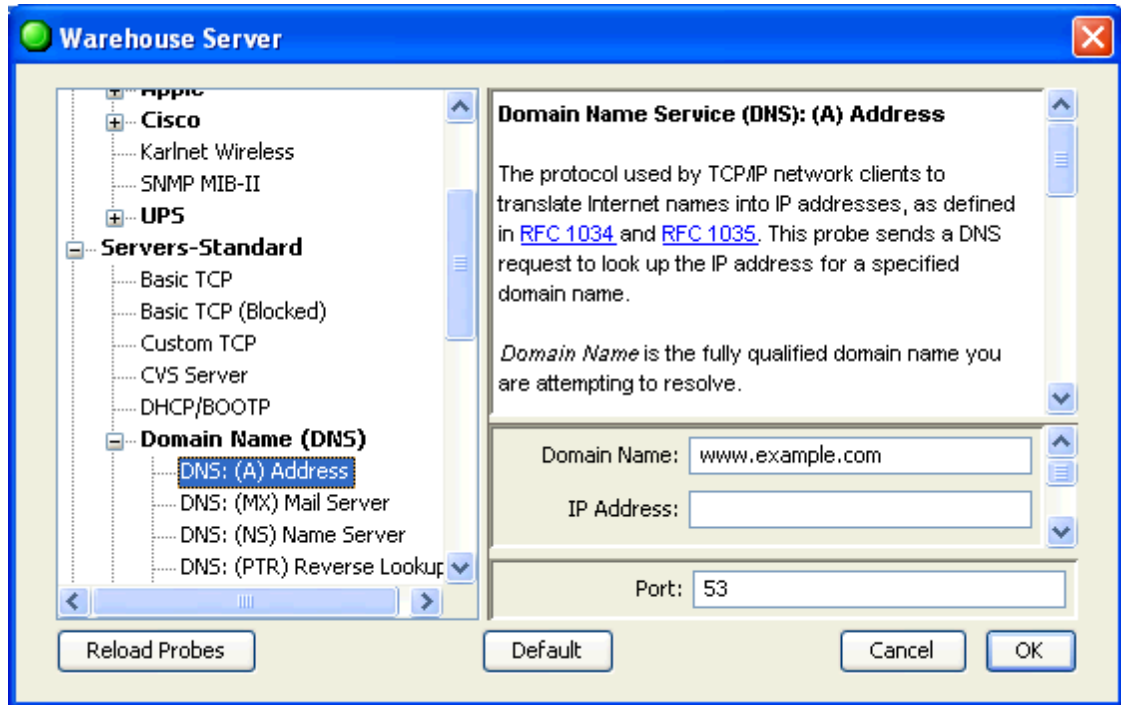
```
package = "com.dartware"  
probe name = "tcp.custom"
```

The <description> Section

The Description section of a probe file contains text that will be displayed as a description of the probe in the Probe Configuration window. All probe types can have a description section. It is defined using the following tags:

```
<description> ... </description>
```

The description can be formatted using IMML, [InterMapper's Markup Language](#). The [Example Probe File \(Pg 128\)](#) shows a sample description section.



The Set Probe window, showing the description field. Note that the blue underlined links are actually links to the relevant RFC specifications.

The <parameters> Section

A probe can have one or more parameters. These parameters are set by the user in the Set Probe window (shown below). They are used for specifying numeric thresholds or strings to be sent to or received from the device.

The *parameter* section of the defines a set of name/value pairs with the format:

```
<parameters>
  [parameter name]= "[parameter value]"
</parameters>
```

Each parameter name/value appears in its own entry field in the Set Probe window.

Probe parameters are accessed and used just like variables. They can be used in calculations, alarm/warning thresholds, and displayed in the status window. To refer to a parameter whose name contains one or more spaces, the name will need to be enclosed in curly braces. (Example: "\${Seconds to wait}").

Input Field Types

Four types of input fields are available:

- **Text** - Input a text string
- **Password** - Input a text string, obscuring the characters
- **Dropdown** - Choose from a dropdown menu.
- **Checkbox** - Set a variable to true or false by selecting or clearing a checkbox.

Text Fields

This field type presents a simple text box for entering a string.

```
"Text" = "Text Value"
```

The line above sets the variable \${Text}

Password Fields

You can create input parameters that conceal the string from casual view (so-called *password parameters*.) The data is displayed as a line of asterisks ("*****") when a user types the password. To specify a password parameter, place a single asterisk ("*") after the name of the field, like this:

```
"Password*" = ""
```

Note that the variable name remains `${Password*}` and you have to refer to it as such in your script. The "*" is removed before displaying the name, so the above password parameter would appear as "Password" in the Set Probe window.

Dropdown Fields

You can create input parameter fields that present a dropdown menu from which the user can choose from a number of choices.

To create a dropdown field, use this syntax:

```
"Test[Equal,NotEqual]" = "NotEqual" //Default value is NotEqual
```

The values between brackets define the choices available to the user. The value on right of the statement is the initial value of the dropdown field.

You can use this parameter in expressions. The full variable is `${Test[Equal,NotEqual]}`, and it returns the current value of the dropdown as selected by the user. To display the value of a dropdown in the Status window, you must use the full variable definition.

```
\4\Dropdown:\0\ ${Test[Equal,NotEqual]}\0\
```

```
<snmp-device-variables>
  alarm: (${Dropdown[Choice1,Choice2,Choice3]} !=
    "Choice2") "It's not Choice2!"
</snmp-device-variables>
```

Check Box Fields

To create a check box, use this syntax:

```
"Checkbox[true,false]" = "true" //Default value is "true"
```

You can use this parameter in expressions. The full variable is `${Checkbox[Equal,NotEqual]}`, and it returns the current value of the check box as selected by the user

Parameter Section Example

This is an example parameter section that demonstrates the use of the four types of input fields. The screenshot below shows how each input field type appears.

```
<parameters>  
  "Text" = "Text Value"  
  "Password*" = ""  
  "Dropdown[Choice1,Choice2,Choice3]" = "Choice2"  
  "Checkbox[true,false]" = "true"  
</parameters>
```

The screenshot shows a window titled "Parameter Test Probe". Inside, there is a text area with the description: "This probe shows the various data types that can be used as parameters to a custom probe." Below this, there are several input fields:

- Text:** A text box containing "Text Value".
- Password:** A text box containing six asterisks "*****".
- Dropdown:** A dropdown menu showing "Choice2" with a downward arrow.
- Checkbox:** A checkbox that is checked, indicated by a green checkmark.
- SNMP Version:** A dropdown menu showing "SNMPv1" with a downward arrow.
- Port:** A text box containing "161".
- Community:** A text box containing six asterisks "*****".

Probe Parameters

The <autorecord> section

Certain data values collected from a device are automatically recorded to Inter-Mapper Database. You can also specify other variables you want to record when data for a device is stored.

For all probes, the following data is recorded:

- response time (in msec) - tag: **BiRt**
- short-term packet loss (%) - tag: **RPkL**
- input byte rates for all visible interfaces - tag: **BytR**
- output byte rates for all visible interfaces - tag: **BytT**

In addition to the values listed above, built-in probes automatically record other values. The [Automatically-Recorded Data Values \(Pg 23\)](#) page lists those values for each built-in probe.

For custom probes, you can specify which variables should be recorded. The syntax for this is as follows:

```
<autorecord>
  $var1, 'tag1', "Legend 1 :units(xxx)"
  $var2, 'tag2', "Legend 2"
  $var3, 'tag3', "Legend 3"
</autorecord>
```

where:

- **\$varX** is a variable defined by the probe
- **tagX** is a short tag that identifies a particular class of dataset. Use these tags to create a report of like variables, such as CPU%, temperature, etc. See the Automatically-Recorded Data Values page to view pre-defined tags. Probe writers are free to create their own short tags as long as they don't start with "_".
- **Legend X :units(xxx)** is a human-readable text string that describes the dataset, and shows the customer the kinds of data being collected for a particular device. Specify the units for the dataset using the optional ":units" attribute. This legend will override a legend placed in a <snmp-device-variable> section.

Example

```
<autorecord>
  $lcpu.busyPer, 'cpupercent', "CPU Percent :units(%)"
  $lcpu.avgBusy1, 'cpupercentavg', "Average CPU Percent :units(%)"
  $lmem.freeMem, 'freemem', "Available memory :units(bytes)"
</autorecord>
```

Automatically-Recorded Data Values

When recording data from built-in probes, Dartware has selected values for each probe which seem sensible to record.

Probe Name	\$variable in probe	Tag (30)	Legend (255)
APC UPS	leftCharge	pctcharge	Percent Charge
	batMin	batttimeleft	Time left on battery (min)
	inVolt	involts	Input Voltage
	batTempC	temperature	Internal temperature (C)
APC UPS - AP961x	[same as APC UPS]		
Barracuda HTTP	in_queue_size	inqueue	Input Queue
	out_queue_size	outqueue	Output Queue
	avg_latency	latency	Average Message Latency
Barracuda HTTPS - same as HTTP	[Same as Barracuda HTTP]		
BestPower UPS	cTimeOnBattery	batttimeleft	Time left on battery (min)
	cInputVoltage	involts	Input Voltage
	cIntTempC	temperature	Internal temperature (C)
Custom TCP	_connect	conntime	Time to establish connection
	_active	connactive	Time spent connected to host
Exide UPS	LeftCharge	pctcharge	Percent Charge
	LeftMin	batttimeleft	Time left on battery (min)
	in1Volt	involts	Input 1 Voltage
HTTP	_connect	conntime	Time to establish connection
	_active	connactive	Time spent connected to host
HTTP (Authenticate)	Same as HTTP		
HTTP (Post)	Same as HTTP		
HTTP (Proxy)	Same as HTTP		
HTTP (Virtual Server)	Same as HTTP		
HTTPS	Same as HTTP		

HTTPS (Post)	Same as HTTP		
Liebert UPS - Open-Comms	LeftCharge	pctcharge	Percent Charge
	LeftMin	batttimeleft	Time left on battery (min)
	in1Volt	involts	Input 1 Voltage
	batteryTempC	temperature	Internal temperature
OS X Server - AFP	currentThroughput	throughput	Current throughput
	currentConnections	connections	Current connections
OS X Server - FTP	realConnectionCount	authconns	Authenticated Connections
	anonymousConnectionCount	anonconns	Anonymous Connections
OS X Server - NAT	activeTCP	tcpconns	TCP Links
	activeUDP	udpconns	UDP Links
	activeICMP	icmpconns	ICMP Links
OS X Server - Print	currentQueues	queues	Current Queues
	currentJobs	numjobs	Spooled Jobs
OS X Server - QTSS	currentConnections	connections	Current connections
	currentThroughput	throughput	Current throughput
OS X Server - Web	currentRequestsBy10	requestrate	Requests/second
	cacheCurrentRequestsBy10	cachehitrate	Cache Requests/second
	currentThroughput	throughput	Current throughput
	cacheCurrentThroughput	throughputcache	Cache throughput
OS X Server Info	cpu	cpupercent	CPU Usage
SNMP - Cisco - Old CPU MIB	lcpu.busyPer	cpupercent	CPU Percent
	lcpu.avgBusy1	cpupercentavg	Average CPU Percent
	lmem.freeMem	freemem	Available memory
SNMP - Cisco - Process and Memory Pool	lcpu.busyPer	cpupercent	CPU Percent

	lcpu.avgBusy1	cpupercentavg	Average CPU Percent
	ciscoMemoryPoolFree1	freemem	Available memory
SNMP - Cisco (v2c)	lcpu.busyPer	cpupercent	CPU Percent
	lcpu.avgBusy1	cpupercentavg	Average CPU Percent
	lmem.freeMem	freemem	Available memory
SNMP - Cisco IP SLA Jitter Probe	cpmCPUTotal1min	cpupercentavg	Average CPU Percent
	AvgJitter	jitteravg	Average Jitter
	AvgLatency	latencyavg	Average Latency
	PercentPacketLoss	pktloss	Packet Loss
SNMP - Comparison	theOID	value	Probe value
SNMP - High Threshold	theOID	value	Probe value
SNMP - Host Resources (built-in)	Average of items in hrProcessorTable or UCD-SNMP CPU table (anchor is UcuR)	cpupercentavg	Average CPU Percent
SNMP - Low Threshold	theOID	value	Probe value
SNMP - Range Threshold	theOID	value	Probe value
SNMP - TCP Check	tcpCurrEstab	numconns	Number of TCP connections
Standard UPS (RFC1628)	LeftCharge	pctcharge	Percent Charge
	LeftMin	batttimeleft	Time left on battery (min)
	in1Volt	involts	Input 1 Voltage
	batTempC	temperature	Internal temperature (C)
TrippLite UPS	LeftCharge	pctcharge	Percent Charge
	LeftMin	batttimeleft	Time left on battery (min)
	in1Volt	involts	Input 1 Voltage
	envTempC	temperature	Ambient temperature (C)
	envHumid	humidity	Humidity (%)
Victron UPS	batt.rem	batttimeleft	Time left on battery (min)

input.volt1

involts

Input 1 Voltage

Probe Status Windows

When you create a custom probe, you can override the default contents of Status windows. How you do this depends on the type of probe, as follows:

- **The <snmp-device-display> section** - SNMP probes
- **The <script-output> section** - TCP probes
- **The <command-display> section** - Command-line probes

All of these sections can be formatted using IMML, [InterMapper's Markup Language](#). See the <snmp-device-display> example below.

Controlling the Status window in SNMP Probes with <snmp-device-display>

Use the optional <snmp-device-display> section to describe the text that appears in a custom SNMP probe's Status window. Probe variables are replaced with their values in the Status window's text.

The default font for the Status window's text is a mono-spaced font, so alignment of text is straightforward. You can change the appearance of the text in the Status window using IMML, [InterMapper's Markup Language](#).

Here is a sample <snmp-device-display> section. Note that the variables are replaced with the values retrieved from the device, and that formatting is controlled by IMML.

```
<snmp-device-display>
  \B5\Custom SNMP Probe\OP\
  \4\ipForwDatagrams:\0\  ${ipForwDatagrams} datagrams/sec
  \4\ipInHdrErrors:\0\    ${ipInHdrErrors} errors/minute
  \4\tcpCurrEstab:\0\     ${tcpCurrEstab} connections
</snmp-device-display>
```

Controlling the Status window in TCP Probes with <script-output>

Use the optional <script-output> section to describe the text that appears in a TCP-based custom probe's Status window. The data in this section appears in the Status window when you click and hold the device on the map.

Controlling the Status window in Command-Line Probes with <command-display>

Use the optional <command-display> section to describe the text that appears in a command-line-based custom probe's Status window. The data in this section appears in the Status window when you click and hold the device on the map.

The format of this section is the same as the <snmp-device-display> section described above.

IMML - InterMapper Markup Language

You can apply text styles to a probe's description text or to the content in a Status window using IMML, InterMapper's markup language. IMML consists of formatting commands bracketed by backslash ("\") characters. There may be many markup commands between a pair of \...\
characters. The [Example Probe File \(Pg 128\)](#) shows a sample description section.

Historical note: Prior to InterMapper 4.0, the markup characters were « and » (≤ and ≥). InterMapper still accepts these characters, although we encourage everyone to use the \...\
in new probe files as they're easier to type and will pass unchanged through all mail systems.

How Markup Tags Are Applied

- A markup commands applies to all text that follows it.
- Subsequent markup tags may add to or counteract a previous set of markup tags.

Markup Tag Summary

Tag	Action
-----	--------

- | | |
|----------|---|
| M | Set the font to a mono-spaced font.. |
| G | Set the font to Geneva or other proportional-spaced font). |
| + | Increase the font size by one. Multiples ("++") increase the font size by the corresponding amount. |
| - | Decrease the font size by one. Multiples are allowed. |
| B | Set following text in bold. |
| I | Set following text in italic. |
| P | Set following text to plain. This undoes all other stylings |
| U | Set following text to underlined. See Creating a link below for making hyperlinks. |
| ! | Turn off a format that is on. |

digit Set text color to one of the following:

- | | |
|----------|---------------|
| 0: Black | 4: Light blue |
| 1: Red | 5: Green |
| 2: Blue | 6: Orange |
| 3: Gray | 7: Yellow |

Examples

The following description text is rendered as shown:

<code>\b\Bold \i\Bold Italic \!b\Italic \p\Plain</code>	Bold Bold Italic Italic Plain
<code>\Ml++\Big red monospace\p\ </code>	Big red monospace

```
\2U\http://www.example.com\p0\
```

<http://www.example.com>

```
\2U=http://www.example.com\Text  
Link\p0\
```

[Text Link](#)

Creating a link

The last two examples above shows the script code required to create a link. In both example, "\2U\" means "set the color to blue, underline the text."

Special Cases:

- If, as in the first of the two link examples above, the only text between the opening and closing tags is a URL (e.g., <http://www.example.com>), Inter-Mapper treats it as a link to that page.
- If, as in the last link example above, the underline tag contains "[URL]", the text following the backslash ("Text Link" in the example) appears as blue and underlined.
- In both cases, clicking the text opens that page in a browser.

Probe Comments

Comments in InterMapper probe files are quite similar to those in HTML. The comments may be interspersed anywhere in a probe file.

Note that HTML comments have a complicated syntax that can be simplified by following this rule:

Begin a comment with "`<!--`", end it with "`-->`", and do not use "`--`" within the comment.

Use this rule with InterMapper as well.

Example:

```
<!--  
  This is a probe comment.  
  It spans several lines.  
  It contains no double-hyphens.  
-->
```

One-line Comments

You can also use the comment indicator "`--`" at the beginning of a line. The remainder of the line is ignored.

Example:

```
-- This line is a comment
```

Built-in Custom Probe Variables

Here is a list of built-in variables available in custom-probes and notifiers. Note that certain variables are available only in certain contexts. The variables are listed by context.

- [Command-line Probe Variables](#) (Pg 31)
- [SNMP Probe Variables](#) (Pg 33)
- [TCP Probe Variables](#) (Pg 35)
- [Command Line Notifier Variables](#) (Pg 36)
- [The Chartable macro](#) (Pg 36)
- [The Eval Macro](#) (Pg 37)
- [The Scalable10 and Scalable2 Macros](#) (Pg 37)

Command Line Probe Variables

The following variables are available in the specified sections of Command-line probes. (probe-type=cmd-line)

The <command-line> and <command-exit> sections

The following variables are available in <command-line> and <command-exit> sections of command line probes

Variable Name	Variable Description
\${address}	The network address of the device.
\${devicename}	The device name of the device. Note: In some cases, the device name may resolve to the device's IP address.
\${port}	The network port number that is being monitored.
\${exit_code}	The exit code of the command-line probe. Note: The \${exit_code} variable is used in <command-exit> only.
\${cscript}	Evaluates to the full path to the Windows cscript.exe utility; it also automatically adds /nologo as a command-line option.
\${python}	Note: This variable is only available in Windows. Evaluates to the full path to the python interpreter installed as part of InterMapper DataCenter; it also automatically adds any necessary command-line options for normal operation.

The <command-display> section

The following variables are available in the <command-display> section of command line probes. (probe-type=cmd-line)

Variable Name	Variable Description
<code>\${devicename}</code>	The device's name taken from first line of the label.
<code>\${deviceaddress}</code>	The network address of the device.
<code>\${eval:} (Pg 37)</code>	The eval macro.
<code>\${chartable[:fmt]:expr} (Pg 36)</code>	Evaluate <code>expr</code> and format the result as a chartable value
<code>\${scalable2:fmt:expr}</code>	Scales large numbers into smaller units for better readability. The values are chartable.
<code>\${scalable10:fmt:expr} (Pg 37)</code>	
<code>\${^stdout} (Pg 38)</code>	Any output written to the standard output of a command-line script. See below for additional information about the effect of <code>\${^stdout}</code> on the <code>reason</code> string.
<code>\${nagios_output}</code>	Parses a Nagios plugin's output for display.

SNMP Probe variables

These variables are available in SNMP Probes. (probe-type = customsnmp)

The <snmp-devicedisplay> section

Variable Name	Variable Description
<code>\${devicename}</code>	The device's name taken from first line of the label.
<code>\${deviceaddress}</code>	The network address of the device.
<code>\${imserveraddress}</code>	The network address of the InterMapper Server.
<code>\${alarmpointlist}</code>	alarm point list.
<code>\${eval:expr} (Pg 37)</code>	The eval macro.
<code>\${chartable[:fmt]:expr} (Pg 36)</code>	The chartable macro.
<code>\${scalable2:fmt:expr}</code>	Scales large numbers into smaller units for better readability. The values are chartable.
<code>\${scalable10:fmt:expr} (Pg 37)</code>	
<code>\${[variablename]:legend}</code>	The variable's legend as specified in the <snmp-device-variables> section For more information, see SNMP Probe Variables (Pg 43) .

*The <snmp-deviceproperties> section***Variable Name Variable Description**

\${ifIndex}	The interface index.
\${ifType}	The interface type.
\${ifDescr}	The interface description.
\${ifAlias}	The interface's alias

The OID column of the <snmp-device-variables> section

Variable Name	Variable Description
\${SpecificTrap}	Trap Field: specific-trap (SNMP v1; generic-trap is enterpriseSpecific)
\${GenericTrap}	Trap Field: generic-trap (SNMP v1)
\${TimeStamp}	Trap Field: trap timestamp (SNMP v1, v2c)
\${Enterprise}	Trap Field: enterprise (SNMP v1)
\${CommunityString}	Trap Field: community (SNMP v1, v2c)
\${TrapOID}	Trap Field: trap OID (SNMPv2c, v3)
\${SnmVersion}	Trap Field: trap version
\${SenderAddress}	Trap Field: trap sender's address
\${AgentAddress}	Trap Field: trap agent's address (if different from sender)
\${VarbindCount}	Trap Field: count of varbind variables
 Note: the next three macros do not use a colon \${V-arbindValue8} returns the value of the eighth Varbind item.	
\${VarbindOID<num>}	Trap Field: nth varbind OID
\${VarbindValue<num>}	Trap Field: nth varbind Value
\${VarbindType<num>}	Trap Field: nth varbind Type

TCP Probe Variables

These variables are available in TCP probes. (probe-type = tcp-script)

The <script> section

Variable Name	Variable Description
<code>\${_remoteaddress}</code>	The network address of the remote end of the connection.
<code>\${_remoteport}</code>	The network port number of the remote end of the connection.
<code>\${_localaddress}</code>	The network address of the local end of the connection.
<code>\${_localport}</code>	The network port number of the local end of the connection.
<code>\${_gmttime}</code>	The current time in RFC 822 format.
<code>\${_version}</code>	The version number of the InterMapper program.
<code>\${_line:len}</code>	The text of the last line received, truncated to the given length.
<code>\${_idletimeout}</code>	The idle timeout for the probe in seconds.
<code>\${_string-to-match}</code>	The string we attempted to match in the last EXPT or MTCH command.
<code>\${_base64:str}</code>	Encode the given argument into base64.
<code>\${_cvspass-word:str}</code>	Encode the given argument using the cvs password algorithm.
<code>\${_md5:str}</code>	The MD5 hash of the given argument, in hexadecimal.
<code>\${_idleline}</code>	The line number of the script where we were before the idle handler was invoked.
<code>\${_sec-connected}</code>	The number of seconds the probe spent connected to the other end. May be 0 if we were disconnected immediately or connection failed.
<code>\${_length:str}</code>	The length of the given argument in bytes.
<code>\${_float:num}</code>	The argument "pretty-printed" as a floating point number via printf %g.
<code>\${_hmac:key:msg}</code>	The HMAC-MD5 of the message, using the given key.
<code>\${_xor}</code>	
<code>\${_urlencode:str}</code>	Encodes the specified string for use in URLs.

The <script-output> section

Variable Name	Variable Description
<code>\${_devicename}</code>	The device's name taken from first line of the label.
<code>\${_deviceaddress}</code>	The network address of the device.
<code>\${_eval:} (Pg 37)</code>	The eval macro.
<code>\${_scalable2:fmt:expr}</code>	Scales large numbers into smaller units for better readability. The values are chartable.
<code>\${_scalable10:fmt:expr} (Pg 37)</code>	

Variables passed to Command-line Notifiers

The following variables are available for passing to a command line notifier. These values allow you to pass messages or URLs as command-line arguments in formats that are platform-friendly.

Variable Name	Variable Description
<code>\${message}</code>	The notifier's message text. (On Windows, each double-quote " is escaped by \".)
<code>\${stripped_message}</code>	The notifier's message text with quotes (' and ") removed and newlines (\r and \n) replaced by space.
<code>\${escaped_message}</code>	The notifier's message text escaped for url syntax (e.g. %20 for space, etc.)
<code>\${urlescape:str}</code>	Escapes a string specified in str for use in a URL. Any macros included in str are expanded prior to escaping.

The `${chartable}` macro

Use the `${chartable}` macro to evaluate **expr** and to format the result as a chartable value.

Usage

```
${chartable[:fmt]:expr}
```

where `fmt` may be of the form:

- any format specifier supported by the `sprintf` function [Probe Calculations \(Pg 68\)](#), enclosed in double quotes (")
- `##.##` where the `#` specifies the number of digits to the left and right of the decimal point, and `val` may be either a number or the name of a variable.

Example

A SNMP device might return a temperature as "723", where the temperature is 72 degrees Fahrenheit. Although this value is useful in charting the value, you might want to format the output in the status window as follows:

```
\4\Temperature (F):\0\ ${chartable: ##.##: ${temp}}
<!-- or -->
\4\Temperature (F):\0\ ${chartable: "%4.1f": ${temp}}
<!-- Both result in "72.3" -->
```

The `eval` macro

Use the `eval` macro to compute a value in the output of a script. It is available in the following contexts:

- The `<command-display>` section of command-line probes
- The `<snmp-devicedisplay>` section of SNMP probes
- The `<script-output>` section of TCP probes

Usage

The syntax for the `eval` macro is as follows:

```
eval:[expr]
```

The expression can use any operators or functions defined in [Probe Calculations \(Pg 68\)](#), allowing you to perform variable assignments, arithmetic calculations, relational and logical comparisons, as well as use built-in functions to perform bit-wise, rounding and mathematical operations. You can also perform operations on strings using `sprintf` formatting and regular expressions.

Examples

Here are some examples that use the Eval macro:

```
Arithmetic: eval:$test := (4-1)*(2+1)/(9/3)
<!-- result = 3, also assigns result to $test -->

Modulo: eval:$test%2
<!-- result = 1, uses the $test variable -->

String assign: eval:$yes:="Yes" eval:$no:="No"
Numeric assign: eval:$test:=5 (
Conditional: eval: $test==5 ? ($response := "Yes") : ($response :=
"No") )
<!-- result="Yes", because $test variable = 5
      result also assigned to $response variable, output on the next
line -->
$response

subid(): eval:subid("1.3.6.1.2.1.4.20.1.1.10.10.2.20", -4, 4)
<!-- result="10.10.2.20" -->

Regular Expression: eval: "test123" =~ "^te([st]*) ([0-9]*)"; "${1}
${2}"
<!-- result = "st 123"-->
```

The `scalable10` and `scalable2` macros

Use the `scalable10` and `scalable2` macros to display numbers in the appropriate scaled units. The values are always between 1.0 and 1000, are chartable, and are scaled by a factor of 1000 or 1024:

- The `${scalable10}` macro scales by a factor 1000 (for msec, Mbps, etc.).
- The `${scalable2}` macro scales by a factor of 1024 (for KBytes, GBytes, Terabytes, etc.)

Both macros display the appropriate scale. They use the same syntax as the `${chartable}` macro.

Usage

`${scalable10[:fmt]:expr}`

`${scalable2[:fmt]:expr}`

Example

```
${scalable10: #.## : 12304 }bytes => 12.30 kbytes
${scalable10:"%3.2d" : 12304 }bytes => 12.30 kbytes
```

```
${scalable2: #.## : 12304 }bytes => 12.02 kbytes
${scalable2:"%3.2d" : 12304 }bytes => 12.02 kbytes
```

The following examples use the `scalable10` macro.

char	scale factor	short for	example val	units	example output
k	* 1000	kilo	12304	bytes	12.30 kbytes
M	* 1e6	Mega	3421814	bytes	3.42 Mbytes
G	* 1e9	Giga	125032100300	bytes	125.03 Gbytes
T	* 1e12	Tera	1.23 x 10 ¹²	bytes	1.23 Tbytes
none	* 1	nothing	123	bytes	123.00 bytes
m	/ 1000	milli	0.02835	sec	28.35 msec
u	/ 1e-6	micro	0.00047658	sec	476.58 usec
n	/ 1e-9	nano	0.0000000032	sec	3.20 nsec
p	/ 1e-12	pico	1.0 x 10 ⁻¹⁰	sec	100.00 psec

The `${^stdout}` variable and the Reason string

In command-line probes, a specially formatted output string is used to define variables and their values. Normally, *anything else written to standard output is used as the reason string for the probe.*

If `${^stdout}` exists in the command-display section of a command-line probe, then anything written to standard output by the probe script is assigned as the value of the `${^stdout}` variable. This allows the script to programmatically define all or part of the contents of the lower part of the status window.

Using `${^stdout}` means that the reason string does not get defined. To compensate, and to allow the definition of a meaningful reason string, a convention was defined. If the specially-formatted output string mentioned above defines a variable named `reason`, then its value is assigned to the reason string used in the status window.

For example, output from the WMI Top Processes probe might look like this:

```
\{ProcessTime0:=100.0,CPU:=1.0,reason:="CPU utilization is below  
60%"}  
\B5\WMI Top Processes\OP\  
  \B4\CPU Utilization:\P0\    $CPU %  
    \B4\wmiprvse(3924)\P0\    $ProcessTime0 %
```

Chapter 3

SNMP Probes

```
type="custom-snmp"
```

Using custom SNMP probes, you can monitor certain MIB variables that aren't tested by InterMapper's built-in probes. These MIB variables might include the CPU utilization of a router, temperature inside the equipment, switch closures, etc.

Like other probes, custom SNMP probes are invoked and return the status and condition string for the device being tested. Here's a summary of the operational flow of a custom SNMP probe:

1. InterMapper polls the device for the values (called *probe variables*) specified in the probe file as well as the device's built-in MIB variables (usually byte and packet rates for interfaces).
2. InterMapper also polls each interface for probe variables as needed.
3. InterMapper then evaluates a series of expressions in the probe file, comparing the probe variables to thresholds.
4. If a comparison is triggered (generally, if the probe variable is above or below the given threshold), then InterMapper sets the device status as specified in the probe, if it is "worse" than the device's current status.
5. When the user clicks and holds on a device, InterMapper processes the relevant *display* section to produce the text for the Status window.

Common Sections of an SNMP probe

Custom SNMP probes follow the same general format as other probe files.

- The [<header>](#), (Pg 13) section of a command-line probe specifies the probe type, name, and a number of other properties fundamental to the operation of the probe.
- The [<description>](#) (Pg 18) section specifies the help text that appears in the Probe Configuration window. Format the description using IMML, [InterMapper's Markup language](#).
- The [<parameters>](#) (Pg 19) section defines the fields presented to the user in the Probe Configuration window.

Sections Specific to SNMP Probes

```
<header>  
  type = "custom-snmp"  
</header>
```

Note: See [Custom SNMP Trap Probes](#) for information on creating probes that handle SNMP traps.

Each custom SNMP probe also has:

- [An <snmp-device-variables> section \(Pg 41\)](#) - Specifies which MIB variables to collect from the device
- [An <snmp-device-thresholds> section \(Pg 48\)](#) - Specifies how those variables are to be tested against thresholds to determine the device's status
- [An <snmp-device-display> section \(Pg 41\)](#) - Specifies what information about the device and its links should be displayed in the Status window
- [The <snmp-device-properties> section \(Pg 42\)](#) - Specifies certain aspects of the SNMP queries sent to the device

SNMP Probe Variables - The <snmp-device-variables> Section

Use the <snmp-device-variables> section to specify the values you want to retrieve using a particular SNMP OID. These values, called *probe variables*, can then be compared to thresholds to create alarms, warnings, etc.

Each line of the <snmp-device-variables> section defines a particular variable to be retrieved. The definition is composed of four comma-separated attributes:

```
[Variable-name], [OID], [Type], [Chart-legend]
```

For a complete description of these variables, see [Probe Variables \(Pg 43\)](#).

Sample <snmp-device-variables> Section

```
<snmp-device-variables>
  ---Variable-name  OID --- TYPE --- CHART LEGEND -----
  ipForwDatagrams, 1.3.6.1.2.1.4.6.0, PER-SECOND, "Forwarded data-
grams"
  ipInHdrErrors,   1.3.6.1.2.1.4.4.0, PER-MINUTE, "IP received header
err"
  tcpCurrEstab,    1.3.6.1.2.1.6.9.0, DEFAULT,    "Number of TCP
conn's"
  sysDescr,        1.3.6.1.2.1.1.1.0, DEFAULT
</snmp-device-variables>
```

Note: The OIDs above have a trailing ".0" to specify their full OID.

Status Window Text - The <snmp-device-display> Section

Use the <snmp-device-display> to control the way information gathered during polling appears in the Status window. Create a <snmp-device-display> section with the items to be displayed. For more information, see [Customized Status Windows \(Pg 27\)](#).

SNMP Query Settings - The `<snmp-device-properties>` Section

The `<snmp-device-properties>` section specifies certain aspects of the SNMP queries sent to the device. Like other sections, it is closed with a `</snmp-device-properties>` tag. The items that may be set include:

- **nomib2="true"** - InterMapper does not query the sysUptime MIB-2 variable
- **pdutype="get-request"** - InterMapper uses SNMP Get-Request, instead of Get-Next-Request queries
- **apcups="false"** - If apcups is false, InterMapper does not query the APC-UPS MIB even for devices that auto-detect as one.
- **maxvars="10"** - maxvars controls the maximum number of variables to put in each SNMP request. If a custom probe requires more variables than maxvars, InterMapper sends multiple queries containing up to maxvars variables.

```
<snmp-device-properties>
  nomib2="true"
  pdutype="get-request"
  apcups="false"
  maxvars="10"
</snmp-device-properties>
```

The <snmp-device-variables> Section

InterMapper can retrieve MIB variables from a device and then test them against thresholds. The <snmp-device-variables> section defines the OIDs of MIB variables that are to be retrieved. These values are called *probe variables* and can then be compared to thresholds to create alarms, warnings, etc.

Each line of the <snmp-device-variables> section defines a particular variable to be retrieved. The definition is composed of four comma-separated attributes:

```
[VariableName], [OID], [Type], [Chart Legend (optional)]
```

The definitions of these attributes are:

- **VariableName** is the name used to represent the particular MIB value in this probe. A VariableName consists of letters, digits and an underscore, and must begin with a letter. VariableNames are not case-sensitive. These variable names will be represented in the probe as `$VariableName`. It is not necessary that a particular VariableName match the corresponding name in the MIB, although that is definitely convenient.

Note: Versions of InterMapper prior to 4.2 required that variable names be enclosed in curly braces, e.g., `${VariableName}`. This not necessary in version 4.2 and later, and both forms are now identical. If, however, you want to use a variable name that contains spaces, you *must* enclose the variable in curly brackets.

- **OID** is the Object ID for the particular probe variable. The OID can be expressed as a string of dotted numbers or as an OID name, if the corresponding MIB has been imported into InterMapper. An OID can also be an expression, if the type is "CALCULATION" (see note below).

Notes:

- Calculation variables have a slightly different form, as described below.
- See [Probe Calculations \(Pg 68\)](#) for a description of the functions and operators that are available in expressions. A scalar's OID must end in ".0" according to the SNMP specifications. See [SNMP OIDs \(Pg 80\)](#) for a description of allowable formats for OIDs
- See [On-Demand SNMP Tables \(Pg 50\)](#) for a description of how your probe can display tabular information from a MIB.
- When InterMapper retrieves a value, by default it issues an SNMP Get-Next-Request for the previous OID, unless the pdutype is set to "get-request" (see [Probe Properties \(Pg 79\)](#))
- **Type**- may be one of the following:
 - **Default** - InterMapper deduces an applicable type from the SNMP type of the variable and displays it according to the "Format for DEFAULT types" table below.

- **Integer** - values are coerced to a numeric value. If you have a string value returned "78Fred", the value as INTEGER will be 78.
 - **Integer64** - values are coerced to a numeric value (up to 64-bits). If you have a string value returned "78Fred", the value as INTEGER will be 78.
 - **Hexadecimal*** - If the value is a number, it is displayed a hex number, preceded by "0x" (0xFFFFFFFF). If it is a string, it is represented as a series of hex characters (44 61 72 77 69 6E 20 52 69 63 68 61 72).
 - **Hexnumber** - converts a string of hexadecimal digits into a number. For example, a string value of "FE" would be converted to the number 254.
 - **Double** - converts a string to the corresponding floating point number. Thus a string of "5.678" will be represented internally as a floating point number between 5 and 6, and displayed
 - **Total-value** - displays the actual value of a counter or gauge, not a computed rate value. This will always be an unsigned number.
 - **Total64-value** - displays the actual value of a counter or gauge, not a computed rate value. This will always be an unsigned number (up to 64-bits).
 - **Per-second** and **Per-minute** - force InterMapper to compute a rate for the particular variable by computing the difference between two successive samples and dividing by the elapsed time.
 - **String*** - sets a variable to the text string that corresponds to this OID's enumerated type, as defined in the MIB. (see below)
 - **Calculation** - sets the variable to the result of the calculation shown in the OID field.
 - **TrapVariable** - sets a variable based on the value received from an SNMP trap. A complete discussion of Trap Variables is available in [About Custom SNMP Trap Probes \(Pg 86\)](#)
 - **IPADDRESS** - InterMapper displays a 4-byte octet string as an IPv4 address and a 16-byte octet string as an IPv6 address.
- * STRING and HEXADECIMAL are both strings; they can't be charted.

Format for DEFAULT types:

If the variable is of type... InterMapper will display it as:

Counter32, Counter64 => Per-Second

Unsigned32 (Gauge) => Total-Value

Integer => Integer

OctetString => String (if first digit printable)
or Hexadecimal (if not)

Object ID => String

IPAddress => String

TimeTicks => String

- **Chart-legend** is an optional field that provides a text label to be placed on any strip charts that incorporate this variable. Chart legends may contain embedded variable names in the form `$VariableName`.

Here is a sample `<snmp-device-variables>` section.

```
<snmp-device-variables>
--Variable-name  OID --- TYPE ---- CHART LEGEND -----
ipForwDatagrams, 1.3.6.1.2.1.4.6.0, PER-SECOND, "Forwarded data-
grams"
ipInHdrErrors,   1.3.6.1.2.1.4.4.0, PER-MINUTE, "IP received
header err"
tcpCurrEstab,    1.3.6.1.2.1.6.9.0, DEFAULT, "Number of TCP
conn's"
sysDescr,        1.3.6.1.2.1.1.1.0, DEFAULT
-- Non-pollled values:-- Calculation variables are computed each
poll time
SineValue, (10*sin(0.01*time())), CALCULATION, "10 * sin(0.01 *
time())"
</snmp-device-variables>
```

Note:The OIDs above have a trailing ".0" to specify their full OID.

Calculation Variables

A *Calculation* type variable receives the result of an arithmetic expression. After all variables have been polled, InterMapper calculates the expression, and sets the value of its variable to the result. In the example above:

```
SineValue, (10*sin(0.01*time())), CALCULATION, "10 * sin(0.01 *
time())"
```

The variable "SineValue" will be set to the value of the expression $10 * \sin(0.01 * \text{time}())$. This gives a sine wave that makes an attractive chartable value. Use "\$SineValue" to refer to the variable elsewhere in the probe.

Built-in Variables

InterMapper provides a number of built-in variables. They are detailed in the [Built-in Variable Reference \(Pg 31\)](#) topic. Three macros are described below.

Macros

InterMapper supports several macros that show information about a variable:

`${variablename}` or `$variablename`

In the `<snmp-device-display>` section of a probe file, an occurrence of a variable name (with or without the curly braces `{...}`) is replaced with its value, rounded to the nearest integer. For example, if a calculation variable has the value of 3.14159265, using it in the `<display-output>` section results in the value of "3"; if the variable had the value 4.75 it is displayed as "5". This value is chartable,

that is, clicking it makes a new strip chart, or dragging it adds it to an existing chart. The *ipForwDatagrams* variable defined above could be referred to in either of these forms:

```
${ipForwDatagrams} or $ipForwDatagrams
```

`${chartable: xxx : yyy}`

In the <snmp-device-display> section of a probe file, the `${chartable: ...}` macro creates an underlined value that can be clicked to add it to a strip chart. The macro also controls the field width and number of decimal places. There are two parameters:

- A formatting string that indicates the number and placement of the digits near the decimal point, and the variable to be formatted. The formatting string can be either a mask composed with the '#' symbol or a quoted printf specifier like those accepted by the [sprintf \(Pg 71\)](#) function.
- A variable or an expression (but not a macro). InterMapper evaluates the expression and displays the result according to the formatting string.

Examples:

```
${chartable: #.## : 3 }:      3.14 -->      3.14
${chartable: #.##### : 3 }:  3.1415927 -->  3.1415927
${chartable: "%3d" : 3 }:      3 -->          __ 3 (with 2 leading
spaces)
${chartable: "%9.7f" : 3 }:      3.1415927 ->  3.1415927
${chartable: "%11.7f" : 3 }:     3.1415927 ->  __ 3.1415927 (also with
2 leading spaces)
${chartable: #.##### : 3*100}: 314.1592650 --> 314.1592650
${chartable: "%9.7f" : 3*100}:  314.1592650 ->  314.1592650
${chartable: "%11.7f" : 3*100}: 314.1592650 -> 314.1592650 (no
leading spaces)
```

`${variablename:legend}`

In the <snmp-device-display> section of a probe file, the `${variablename:legend}` macro is replaced with the legend field defined for that variable in the <snmp-device-variables> section. For example:

```
${ipForwDatagrams:legend}
```

results in the string "Forwarded datagrams".

`${eval: expr}`

In the <command-display> section of command-line probes, the <snmp-device-display> section of SNMP probes, or the <script-output> section of TCP probes, use the `${eval}` macro to perform calculations or to assign values to variables.

These macros are detailed in the [Built-in Variable Reference \(Pg 31\)](#) topic.

Enumerated Values

Many MIBs use an integer to represent one of several states. For example, `ifOperStatus` (1.3.6.1.2.1.2.2.1.8.x) is defined in MIB-II as:

```
INTEGER { up(1), down(2), testing(3) }
```

This means that the value 1 represents the "up" condition; 2 represents "down"; and 3 represents "testing".

If you define a variable to retrieve this value as `INTEGER` or `DEFAULT`, the probe will display the value as a number. If you define it as a `STRING`, the probe will use the MIB to find the string representation, and will set the variable to the value "up", "down", or "testing".

If the OID or MIB name isn't defined (because the corresponding MIB hasn't been imported or because of a typo), the probe will display the integer value.

The <snmp-device-thresholds> Section

Use the <snmp-device-thresholds> section to specify the comparisons that should be made between probe variables and other values.

Each line in the threshold section contains a *status*, a *comparison*, and an optional *condition string* for probe variables. If the comparison is *triggered*, (if the expression comparing the probe variable to a constant or other variable is true) then the device is changed to the corresponding status (if that exceeds its current status.)

A threshold can be one of the following (they are case-sensitive):

- okay
- warning
- alarm
- critical
- down

Sample <snmp-device-threshold> Section

```
<snmp-device-thresholds>
  critical: ${ipInHdrErrors} > 15 "ipInHdrErrors critical"
  alarm: ${ipForwDatagrams} > 10 "ipForwarded datagrams too
high"
  alarm: ${tcpCurrEstab} >= 1
  alarm: ${ipInHdrErrors} > 10 "ipInHdrErrors too high"
  warning: ${ipForwDatagrams} > 5
  warning: ${ipForwDatagrams} <= 2
  warning: ${ipInHdrErrors} > 5
</snmp-device-threshold>
```

Creating Comparisons

Comparisons are evaluated in order from top to bottom until a comparison is triggered (result is *true*). At that point, if the associated status is more severe than the device's current status, the device now uses its status and condition. No further comparisons are made once one has triggered.

When a comparison is triggered, it is written to the log file as well as being added to the bottom of the device's Status window. If the condition string is present, it is displayed in addition to the comparison string.

Numeric Comparisons

The following numeric comparison operators are legal:

```
>, >=, <, <=, =, and !=.
```


String Matches

By default, InterMapper performs numeric comparisons.

To compare values as strings:

- Enclose one or both of the operands in double-quotes ("). For example, the comparison

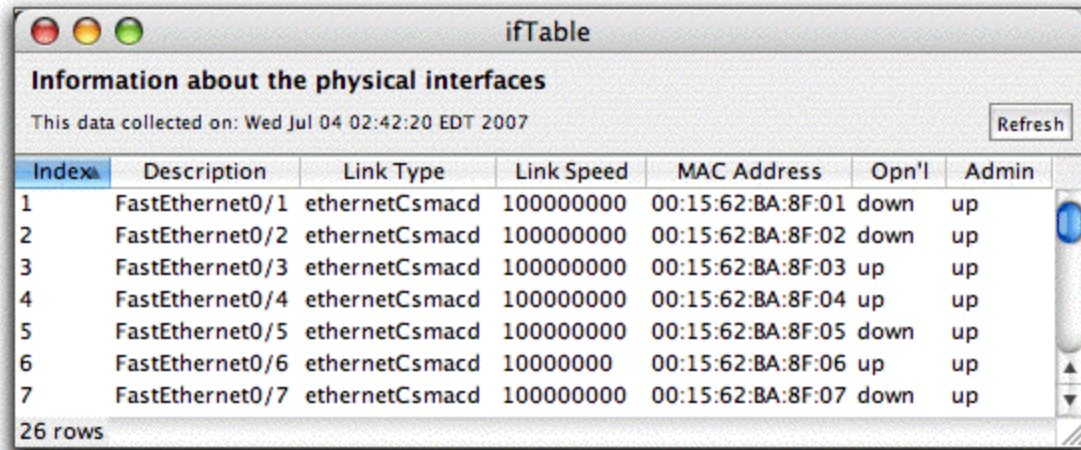
```
warning: ${sysContact} != "Fred Flintstone"
```

performs a string comparison because the name is enclosed in quotes.

- Use the `=~` and `!~` operators to provide partial string matches. They perform "contains" and "doesn't contain" comparisons, respectively.

The <snmp-device-variables-ondemand> Section

Many devices store information in SNMP tables. InterMapper can retrieve this tabular information and display the data "on demand", that is, when requested by user action. When you view a table, InterMapper retrieves the information from the device immediately and displays it in a separate window. The information in an on-demand window is not part of the regular polling cycle, nor is it refreshed until you specifically request it. The image below shows a sample on-demand window.



The screenshot shows a window titled "ifTable" with the subtitle "Information about the physical interfaces". Below the subtitle, it says "This data collected on: Wed Jul 04 02:42:20 EDT 2007" and there is a "Refresh" button. The table has the following columns: Index, Description, Link Type, Link Speed, MAC Address, Opn'l, and Admin. The data is as follows:

Index	Description	Link Type	Link Speed	MAC Address	Opn'l	Admin
1	FastEthernet0/1	ethernetCsmacd	100000000	00:15:62:BA:8F:01	down	up
2	FastEthernet0/2	ethernetCsmacd	100000000	00:15:62:BA:8F:02	down	up
3	FastEthernet0/3	ethernetCsmacd	100000000	00:15:62:BA:8F:03	up	up
4	FastEthernet0/4	ethernetCsmacd	100000000	00:15:62:BA:8F:04	up	up
5	FastEthernet0/5	ethernetCsmacd	100000000	00:15:62:BA:8F:05	down	up
6	FastEthernet0/6	ethernetCsmacd	100000000	00:15:62:BA:8F:06	up	up
7	FastEthernet0/7	ethernetCsmacd	100000000	00:15:62:BA:8F:07	down	up

At the bottom left of the table, it says "26 rows".

On-demand tables are useful for digging down into a device when you suspect there might be a problem. You can create on-demand tables to view a routing table, ARP table, or other statistics that are kept within tables.

Background on SNMP Tables

The SNMP protocol provides access to two types of variables:

- **Scalar variables** contain single values such as strings (that could represent system description or firmware version), integers (number of interfaces), counters (number of errors), gauges (CPU temperature and memory utilization), etc.
- **Table variables** hold information about similar entities within a device. These entities could be interfaces in a router or switch, users associated with a wireless access point, virtual machines in a server, etc. Each entity's information is represented by a row, whose columns are variables (which are themselves scalars) that hold information about the entity. A row is often called an "entry" in a MIB; each column is specified by an OID prefix plus a unique index that specifies a particular row.

For example, MIB-II defines a table named "ifTable" that gives information about a device's interfaces. An outline of ifTable looks like this:

```
+ ifTable
+ ifEntry [ifIndex]
- ifIndex      "Interface Index"
- ifDescr      "Description"
- ifType       "Link type"
- ifSpeed      "Link speed"
- ifPhysAddress "MAC Address"
- ifOperStatus "Operational status"
- ifAdminStatus "Administrative status"
- ... and so on...
```

Here's how to interpret this information. The *ifTable* is composed of a sequence of *ifEntries* which form the rows of the table. Each row (each *ifEntry*) has a number of variables: we show only some of them, starting with ifIndex and ending with ifAdminStatus. These variables become the columns of each row.

The image above shows the window of an on-demand table for ifTable. The columns match the variables mentioned above. The window also shows the number of rows in the table (at lower left), the time the data was retrieved, and the Refresh button to retrieve current data.

Table Indexes

Each row of an SNMP table has a unique index. The index for ifTable is the "interface index", that loosely represents the port number of the interface. Individual values are represented by the column name followed by its index. For example:

```
ifSpeed.3 (or the OID 1.3.6.1.2.1.2.2.1.3)
```

would represent the ifSpeed for row 3 of the table. The "column name" is ifSpeed (1.3.6.1.2.1.2.2.1), and the index is the ".3".

Table Syntax

An on-demand table in a custom SNMP probe mirrors the outline above. Its definition consists of a sequence of lines of comma-separated values defining the variables of one or more tables:

```
<snmp-device-variables-ondemand>
  ifTable, .1, TABLE, "Information about the physical interfaces"
  ifTable/ifIndex, 1.3.6.1.2.1.2.2.1.1, DEFAULT, "Interface Index" <!-- using OID for column -->
  ifTable/ifDescr, 1.3.6.1.2.1.2.2.1.2, DEFAULT, "Description" <!-- using OID for column -->
  ifTable/ifType, 1.3.6.1.2.1.2.2.1.3, STRING, "Link Type" <!-- using OID for column -->
  ifTable/ifSpeed, 1.3.6.1.2.1.2.2.1.5, DEFAULT, "Link Speed" <!-- using OID for column -->
  ifTable/ifPhysAddress, ifPhysAddress, HEXADECIMAL, "MAC Address" <!-- using column name from MIB -->
  ifTable/ifOperStatus, ifOperStatus, STRING, "Oper'l" <!-- using column name from MIB -->
  ifTable/ifAdminStatus, ifAdminStatus, DEFAULT, "Admin" <!-- using column name from MIB -->
</snmp-device-variables-ondemand>
```

This example shows an on-demand table for ifTable.

Note: The <snmp-device-variables-ondemand> section must contain fewer than 50 queries.

The remainder of the table is composed of comma-separated lines describing each variable.

- The first line creates a table. The first field is the table's name that can be used to represent the table elsewhere in the probe file. The second field should be ".1". The third field ("TABLE") indicates that this is a new table. The fourth field is a human-readable description that is displayed in the on-demand window.
- The remaining lines follow the format of <snmp-device-variables>. See their page for more information. The names in the first column contain the table name, a "/", and the name for the column.
- The next four lines define variables (ifIndex, ifDescr, ifType and ifSpeed) that are to be columns of the table. They are defined using the numeric OID that represents the column for those values.
- The final three lines define ifPhysAddress, ifOperStatus, and ifAdminStatus. They are defined using the name of the column from the MIB. This is entirely equivalent to writing out the full numeric OID.

The tables described here are available as the **SNMP/Table Viewer** probe that's built into InterMapper. In addition, the probe file is available on the Table Viewer page of this manual.

Augmenting Tables

Certain MIB's define a table that "augments" another table. This simply means that the augmenting table uses the same index variables as another table. Since the index variables are the same, you can visualize this as adding columns to an existing table.

For example, in the IF-MIB, the ifXTable augments ifTable, providing a number of useful additions.

InterMapper's table syntax easily supports mixing columns from one or more tables that share the same table definition:

```
<snmp-device-variables-ondemand>
  ifXTable, .1, TABLE,
  "Extended ifTable"
  ifXTable/ifIndex, IF-MIB::ifIndex, DEFAULT, "Inter-
face index"
  ifXTable/ifDescr, IF-MIB::ifDescr, DEFAULT,
  "Description"
  ifXTable/ifName, IF-MIB::ifName, DEFAULT, "Name"
  <!-- ifXTable -->
  ifXTable/ifAlias, IF-MIB::ifAlias, DEFAULT, "Alias"
  <!-- ifXTable -->
  ifXTable/ifType, IF-MIB::ifType, STRING, "Link
Type "
  ifXTable/ifSpeed, IF-MIB::ifSpeed, DEFAULT, "Link
Speed"
  ifXTable/ifHighSpeed, IF-MIB::ifHighSpeed, DEFAULT,
  "Mbit/sec"
  ifXTable/ifPhysAddress, IF-MIB::ifPhysAddress, HEXADECIMAL, "MAC
Address "
  ifXTable/ifOperStatus, IF-MIB::ifOperStatus, STRING, "Opn'l"
  ifXTable/ifAdminStatus, IF-MIB::ifAdminStatus, DEFAULT, "Admin"
</snmp-device-variables-ondemand>
```

In the example, ifName and ifAlias come from the ifXTable while the others are part of ifTable. Yet they all can be shown in the same on-demand window.

Derived Variables

Certain SNMP equipment uses the value of one of more columns as part of the row index. In many cases, the column itself is not accessible, and thus cannot be queried directly.

InterMapper has a facility that allows you to derive the values of these columns from the index itself, even from columns that are not accessible. The notation `oid[a:b]` means to fetch the OID `oid` and compute the value from the index:

`[a:b]` - start with the a'th subid and collect b subids.
`[a:]` - start with the a'th subid and collect the remaining subids

Here is an example of retrieving the four columns of `ipNetToMediaTable`. Note that the table has been given the name "ARPTable", although the OID is `ipNetToMediaEntry`.

```
<snmp-device-variables-ondemand>
  ARPTable, .1, TABLE,
  "Map from IP addresses to physical addresses."
  ARPTable/ipNetToMediaIfIndex, ipNetToMediaType[0:1],
DEFAULT, "Interface index"
  ARPTable/ipNetToMediaNetAddress, ipNetToMediaType[1:4],
DEFAULT, "IP Address"
  ARPTable/ipNetToMediaPhysAddress, ipNetToMediaPhysAddress, HEX-
ADECIMAL, "MAC Address"
  ARPTable/ipNetToMediaType, ipNetToMediaType, STRING,
  "Type"
</snmp-device-variables-ondemand>
```

When InterMapper displays table, it will retrieve only two columns: `ipNetToMediaPhysAddress` and `ipNetToMediaType`. It then uses the OID of `ipNetToMediaPhysAddress` to derive the values of `ipNetToMediaIfIndex` and `ipNetToMediaNetAddress`.

Here's how this works. The `ipNetToMediaTable` is defined to use two index values:

- `ipNetToMediaIndex` (the row number of the interface)
- `ipNetToMediaNetAddress` (the IP address of the device)

This means that the full OID used to retrieve an item from the table is its prefix followed by the `ipNetToMediaIndex` followed by `ipNetToMediaNetAddress`.

The notation `ipNetToMediaType[0:1]` returns the single value that immediately follows the OID of `ipNetToMediaType` (this is in fact the index); `ipNetToMediaType[1:4]` returns the following four values, which comprise the IP address used in the index.

Calculations within On-demand Tables

InterMapper 5.3 introduces the ability to have calculations in on-demand tables. This is useful for making calculations from values within the same row of the table. The calculations may also use constant values as well as parameters to the probe. In the example below:

- 1) declares the column "ifIndex"
- 2) calculates the value of (column "ifIndex" plus 1) times 2
- 3) uses the previous column to get the original ifIndex back
- 4) and 5) display the current value of ifInOctets and ifOutOctets
- 6) the calculated ratio between these two columns
- 7) and 8) show a circular reference which fails gracefully - Circular1 and Circular2 refer to each other and simply display "-" as a value.

```
<snmp-device-variables-ondemand>
  ifTableTest, .1, TABLE
  ifTableTest/ifIndex, IF-MIB::ifIndex, DEFAULT
  <!-- #1 -->
  ifTableTest/ifIndexPlus1Times2, ($ifIndex + 1)*2, CAL-
  CULATION <!-- #2 -->
  ifTableTest/ifIndexBack, $ifIndexPlus1Times2/2-1, CAL-
  CULATION <!-- #3 -->
  ifTableTest/TInOctets, IF-MIB::ifInOctets, DEFAULT
  <!-- #4 -->
  ifTableTest/TOutOctets, IF-MIB::ifOutOctets, DEFAULT
  <!-- #5 -->
  ifTableTest/ifRatio, $TInOctets/$TOutOctets, CAL-
  CULATION <!-- #6 -->
  ifTableTest/Circular1, $Circular2 - 1, CAL-
  CULATION <!-- #7 -->
  ifTableTest/Circular2, $Circular1 + 1, CAL-
  CULATION <!-- #8 -->
</snmp-device-variables-ondemand>
```

Displaying On-demand Tables

After you define a TABLE variable in the "ondemand" section of the probe, you can specify that status window should display a link to the ondemand window. To do this, add the variable name to the <snmp-device-display> section of the probe:

```
<snmp-device-display>
...
$ARPTable
</snmp-device-display>
```

The status window displays the table name as a hyperlink. Clicking on the hyperlink opens the on-demand table window shown at the top of the page.

If you want to replace the default table name displayed with your own text, you can specify the "alternate text" in the expanded variable form:

```
<snmp-device-display>
...
${ARPTable:View the entire ARP Table}
</snmp-device-display>
```

Limitations

- On-demand variables must be in table form with a "/" in the variable name.
- You cannot query tables in the regular <snmp-device-variables> section.
- You cannot reference on-demand tables defined in other probes.
- You cannot specify non-accessible MIB variables by their symbolic OID. Instead, use the "derived values" syntax to determine the correct index-derived OID expression.
- You must no more than 50 variables in the <snmp-device-variables-ondemand> section or the query will not work.

The <snmp-device-display> Section

Controlling the Status window in SNMP Probes with <snmp-device-display>

Use the optional <snmp-device-display> section to describe the text that appears in a custom SNMP probe's Status window. Probe variables are replaced with their values in the Status window's text.

The default font for the Status window's text is a mono-spaced font, so alignment of text is straightforward. You can change the appearance of the text in the Status window using IMML, [InterMapper's Markup Language](#).

Here is a sample <snmp-device-display> section. Note that the variables are replaced with the values retrieved from the device, and that formatting is controlled by IMML.

```
<snmp-device-display>
  \B5\Custom SNMP Probe\OP\
  \4\ipForwDatagrams:\0\ ${ipForwDatagrams} datagrams/sec
  \4\ipInHdrErrors:\0\   ${ipInHdrErrors} errors/minute
  \4\tcpCurrEstab:\0\    ${tcpCurrEstab} connections
</snmp-device-display>
```

The <snmp-device-alarmpoints> Section

InterMapper can monitor multiple conditions within a single device (e.g., a single piece of hardware) and give separate, independent notifications for each. For example, it can send notifications for a high temperature alarm independent of a low-memory condition in the same device.

Each of these conditions is called an "alarm point". InterMapper's custom SNMP probe facility allows you to define multiple alarm points for a device, along with their thresholds and the notifications to be sent.

InterMapper tracks the state of each alarm point separately. Alarms on one point will not affect the status, logging, or notifications of any other alarm point. However, the visual appearance of a device will reflect the most serious condition of any of its contained alarm points.

Alarm points have the following five severities. Each severity is assigned a color for quick visual identification.

Severity	Color	Description
Clear	Green	Nothing exceptional to report
Minor	Yellow	Device has departed from its normal "clear" state
Major	Orange	Device's operation is significantly affected
Critical	Solid Red	Device's operation is seriously degraded
Down	Blinking red	Device is unresponsive, actual state is not known

Every time an alarm point changes from one severity to another:

- The the new condition is logged to the log file.
- A notification is sent using the existing InterMapper notifiers, including sounds, e-mail, paging modem or SNPP, or running scripts.

Alarm Points - What the User Sees

InterMapper displays devices with alarm points much the way it shows "regular" devices. A device's icon is colored according to the most serious condition of all its alarm points.

Note that these colors correspond closely with InterMapper's OK/Warning/Alarm/Down coloring. The Critical state is new, and gets a solid red color to indicate that it's "worse" than the orange Alarm or Major severity.

A device's icon takes on the color of its most serious alarm point. For example, a device with two alarm points, one in Critical and one in Minor severity is colored a solid red.

Acknowledgements

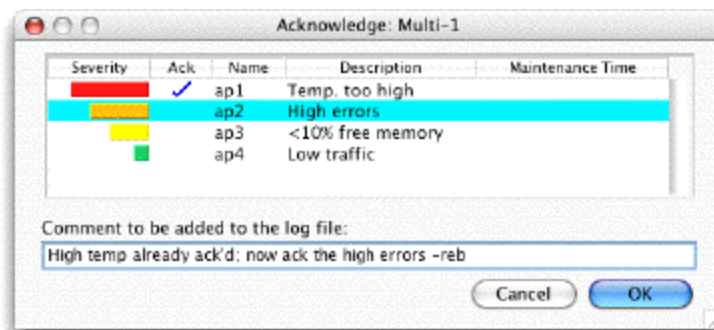
Acknowledging an alarm lets the operator indicate that they know about a problem and that they are working on it. An acknowledgement blocks further notifications for that alarm, and colors the icon blue to show that, although the problem remains, someone has taken responsibility for it.

The blue-acknowledged color makes it easy to see new problems at a glance. When all icons are green (working properly) or blue (in alarm, but being worked on) any new alarm appears as a yellow, orange, or red icon.

Alarm points can be acknowledged independently. That is, acknowledging one alarm point does not affect the state of other alarm points. Acknowledging an alarm point leaves the device's color set to its most serious un-acknowledged alarm point. When all alarm points have been acknowledged, the device icon turns blue.

The Acknowledge window for devices with alarm points will look much like the current Acknowledge window, with these differences:

- When acknowledging one device, the Acknowledge window will display a list of the alarm points, sorted in order of severity. The operator may select one, many, or all the alarm points to acknowledge at a single time.
- The operator may select one, many, or all the alarm points of a device and acknowledge them at the same time.
- Selecting multiple devices and acknowledging them at once acknowledges each alarm point of each device in that one action.
- The Acknowledge window also contains a text field that is used to enter a comment about who is acknowledging the alarm, and why.



Notifications

Alarm points can use the same notification settings as the device, or they can have independent notifications. That is, each alarm point's set of notifications can be separate from any others, and each transition to a new severity can have its own notification.

Notifications for alarm points follow the current InterMapper scheme of sending the notification to an identity. Each identity is configured to use a single notification method (sound, e-mail, modem paging, SNPP, running a script, etc.) to send the desired message.

Alarm point notifications can have independent repeats, delays, and counts, as well. They are defined in the probe file as described in the [Alarm Point Notifier List \(Pg 67\)](#) section.

Log File Messages

InterMapper writes messages to the Event Log file for individual alarm point actions. The entries will be written on a change of severity, for notifications, acknowledgements, or for maintenance mode changes. The lines will have tab-delimited fields in this order:

- **Date-time** Date and time the entry was made into the log file
- **Severity** A four or five-character severity of the event (clear, minor, major, crit, unkn)
- **Identity** The identity of the alarm point, with the map, device, and the alarm point names separated by colons. (e.g., MapName:DeviceName:PointName)
- **Explanatory-text** The condition string or result-description of the alarm point

Configuring Alarm Points

Alarm points are configured in a Custom SNMP Probe. The details are contained in the [Alarm Point Format \(Pg 61\)](#) section of the manual.

Alarm Point File Format

Alarm points are defined in the <snmp-device-alarmpoints> section that contains several lines of the format:

```
name: severity (condition-to-test) Condition-String [ => Noti-
fier-list ]
```

Here is an example:

```
<snmp-device-alarmpoints>
-- Name:      Severity (Condition-to-Test)      Condition-String
=> Notifier-List
   SiteTemp: critical ($Temp > $CriticalHighTemp) "VERY_HIGH_TEMP" =>
PageFred
   SiteTemp:   major ($Temp > $MajorHighTemp)   "HIGH_TEMP"      =>
PageFred
</snmp-device-alarmpoints>
```

The fields of each entry are:

- **Name** is the name of the alarm point. If multiple lines in this section contain the same Name, then they will be treated as the different thresholds for the same alarm point.
- **Severity** is one of 'critical', 'major', 'minor' or 'clear'. It defines the resulting severity of the given point test.
- **Condition-to-test** is an expression that evaluates to a boolean result, e.g. "(\$Temp > \$CriticalHighTemp)" You can use variables from the <snmp-device-variables> section or the <parameters> section in your expression. See the Arithmetic Expression section for details about valid expressions. In the example, \$Temp is a variable read from the SNMP device; \$CriticalHighTemp and \$MajorHighTemp are parameters set by the user.
- **Condition-String** is a string that describes the resulting status if the Condition-to-test evaluates to true.
- **Notifier-list** is an optional, comma-delimited list of notifier names. The notifier names listed here are mapped to actual "InterMapper Notifier Names" via the <snmp-device-notifiers> section.

When InterMapper evaluates alarm point expressions, it scans the list for a particular alarm point, and sets its status based on the first expression that "triggers". If no expression triggers, then InterMapper sets the alarm point severity to "Clear"

Macros

InterMapper supports several macros that show information about an alarm point:

- **`${alarmpointname}`** shows the alarmpoint's severity as a five-character string. The strings are colored to match the severity. To use this facility, enter the alarmpoint name enclosed in `${...}`. For example, to show the SiteTemp alarm point's severity (above), enter:

```
${SiteTemp}
```

and it would generate the strings "CRIT ", "MAJOR", "MINOR", or "CLEAR", with the appropriate color.

- **`${alarmpointname:condition}`** shows the alarmpoint's condition string, as defined in the `<snmp-device-alarmpoints>` section. For example, to show the SiteTemp alarm point's condition above, type:

```
${SiteTemp:condition}
```

and it would generate the string "VERY_HIGH_TEMP". This string may contain any markup as described on the [Probe File Description \(Pg 18\)](#) page.

AlarmPoint Facilities

InterMapper provides several alarm point facilities:

Underscore Feature

Use the Underscore feature to control whether an alarm point is cleared permanently or temporarily when you reset the AlarmPointList.

A device's AlarmPointList contains the following important information:

- alarm points that are **currently in alarm** (i.e., not in Clear state)
- alarm points that were **recently in alarm**, but are now Clear.

This minimizes the clutter in the AlarmPointList, so that it only contains relevant and interesting information. (All the other alarm points are assumed to be clear, and therefore can be ignored.)

The "recently in alarm" qualifier deserves explanation. There is a link in the device's Status Window that allows you to "reset the alarm point list" and remove the cleared alarm points

Alarm point names that begin with an underscore ("_") are treated in this way (hence the name of the facility.) For example: `_SWO_PROC_SWO_PROC` is an underscore alarm point, whereas `SWO_PROC_SWO_PROC` is treated as a normal alarm point.

State Transitions

- **Startup** When a map is opened, devices with alarm points are in the Unknown (grey) state, and their AlarmPointList is empty.
- **Normal Operation:** As InterMapper receives information about a device's state, either from a poll or a received trap, it sets its icon color accordingly. If this information sets a new non-underscore alarm point (one whose state is currently not known), it is added to the AlarmPointList, in the proper color/severity. If it is for a new underscore alarm point, it is added to the AlarmPointList only if the severity is not Clear. If this new information updates an existing alarm point, then that point's severity will be updated in the AlarmPointList.
- **Resetting the AlarmPointList:** When the user clicks an AlarmPointList's **Reset** link in the status window, all the Clear alarm points are removed from the device's AlarmPointList. All alarm points that are in alarm (not Clear) remain in the list.

Note: The **Reset** link removes all "clear" alarm points, but the effect is permanent only for "underscore" alarm points. non-underscore alarm points are cleared temporarily; the cleared non-underscore alarm points reappear in the status page in the next refresh (unless the condition that resulted in "clear" severity has changed).

Resetting to Neutral Alarm State

InterMapper can receive a trap or some other command that sets a device into the "Startup" state described above. This is useful for the process of re-synchronizing InterMapper's notion of a device's state with its actual state. A single trap could indicate, "I don't know the state of a device" (perhaps because it had gone down, and now came back up), and InterMapper reflects that lack of certainty by clearing the alarm point list and turning the device grey.

InterMapper provides a "reset" severity that resets the device to its "power on" state. That is, the device and all its alarm points are set to the Unknown state. Such an alarm point is never listed in the AlarmPointList.

Usage:

```
DeviceResetRule: reset ( reset-condition ) condition-string => Notifier-List
```

If reset-condition is true and there are alarms to be cleared, an event log entry will be created for the action. If the condition-string is not empty, a notification will be sent to the Notifier-List.

Facilities to speed up rule evaluation

We will implement a "Break" severity that will abort the processing of the remaining rules of the probe if ever its expression is true. Such an alarm point will never be listed in the AlarmPointList.

Usage:

```
BreakRule: break ( break-condition ) condition-string => Notifier-  
List
```

In a break rule, Notifier-List is not used. If the condition-string is not empty, an event log message will be created everytime this rule fires (this is done for debugging purposes).

Sample probe

Below a sample probe that uses all the features described above. To test the probe, use the net-snmp *snmptrap* program to send traps to the device. For example, to set \$trapVar to 5, use the following command line:

```
snmptrap.exe -v2c -c community computername ' 1.3.6.1.4.1.11898.2.1
1.3.6.1.4.1.11898.2.1.18.1.18 i 5
```

In the sample probe below, three trap variables are used in three separate alarm points. The first group of alarm points (_trapVarAP) is an "underscore" alarm point, the state is not shown unless the alarm point reaches a non-clear state (minor, major, critical). Note that setting the \$trapVar variable to 4 never brings the alarm point state to critical since there is a break rule right before the rule that brings the state to critical state:

```
<header>
  type           =      "custom-snmp"
  package        =      "com.dartware"
  probe_name     =      "snmp.testalarmpoint"
  human_name     =      "Alarm point test example"
  version        =      "0.1"
  address_type   =      "IP,AT"
  flags          =      "SNMPv2c"
  port number    =      "161"
</header>

<description>
  ...
</description>

<snmp-device-variables>
  trapVar,      1.3.6.1.4.1.11898.2.1.18.1.18, TRAPVARIABLE,
"trap variable 1"
  trapVar2,     1.3.6.1.4.1.11898.2.1.18.1.19, TRAPVARIABLE,
"trap variable 2"
  trapVar3,     1.3.6.1.4.1.11898.2.1.18.1.20, TRAPVARIABLE,
"trap variable 3"
</snmp-device-variables>

<snmp-device-notifiers>
  NotifySomeone: "NotifyFred:0:0:0"
</snmp-device-notifiers>

<snmp-device-alarmpoints>
  -- sample underscore alarm point
  -- the three reset alarm point have the same effect: resetting
the device state to initial state

  _trapVarAP: clear ($trapVar = "2") "trapVar - clear" => Noti-
fySomeone
  _trapVarAP: reset ($trapVar = "3") "trapVar - reset" =>
```

```

NotifySomeone
    trapVarAP: minor (${trapVar} = "4") "trapVar - minor" => Noti-
fySomeone
    trapVarAP: critical (${trapVar} = "5") "trapVar - major" =>
NotifySomeone
    trapVarAP: break (${trapVar} = "6") "trapVar - break" => Noti-
fySomeone
    trapVarAP: critical (${trapVar} >= "6") "trapVar - critical" =>
NotifySomeone

    -- other, normal alarm points
    trapVarAP2: clear (${trapVar2} = "2") "trapVar2 - clear" => Noti-
fySomeone
    trapVarAP2: reset (${trapVar2} = "3") "trapVar2 - reset" => Noti-
fySomeone
    trapVarAP2: break (${trapVar2} = "4") "trapVar2 - break" => Noti-
fySomeone
    trapVarAP2: critical (${trapVar2} >= "4") "trapvar2 - critical"
=> NotifySomeone

    trapVarAP3: clear (${trapVar3} = "2") "trapVar3 alarm" => Noti-
fySomeone
    trapVarAP3: reset (${trapVar3} = "3") "trapVar3 alarm" => Noti-
fySomeone
    trapVarAP3: break (${trapVar3} = "4") "trapVar4 alarm" => Noti-
fySomeone
    trapVarAP3: critical (${trapVar3} >= "4") "trapvar alarm" => Noti-
fySomeone
</snmp-device-alarmpoints>

<snmp-device-display>
    \B5\Trap variable values\0P\
    \4\trapvar:\0\ $trapVar, ${ trapVarAP:condition}\0P\
    \4\trapvar2:\0\ $trapVar2, ${trapVarAP2:condition} \0P\
    \4\trapvar3:\0\ $trapVar3, ${trapVarAP3:condition} \0P\

    $alampointlist
</snmp-device-display>

```

Alarm Point Notifier List

The `<snmp-device-notifiers>` section contains several lines of the format:

```
NotifierName: "notifier-rule" [ , "notifier-rule" ]
```

where the *NotifierName* is an identifier that can be used in the `notifier-list` section of the `alarm-points` section, and the *notifier-rule* is a quoted specification for a notification rule.

A *notifier-rule* contains the name of the actual InterMapper notifier and the notification delay, repeat and count, using the format below (the quote characters are required):

```
"name:delay:repeat:count"
```

Delay and repeat are specified in minutes. If values for delay and repeat are omitted, the value is zero. The count is the number of additional times the notification is repeated. If repeat is zero, the count will be ignored because there is no repeat. If repeat is non-zero and the count is omitted, the count is infinite (repeat forever).

```
<snmp-device-alarmpoints>
  -- Name:      Severity (Condition-to-Test)      Condition-String
=> Notifier-List
  SiteTemp: critical ($Temp > $CriticalHighTemp) "VERY_HIGH_TEMP"
=> PageFred
  SiteTemp:      major ($Temp > $MajorHighTemp)  "HIGH_TEMP"
=> PageFred
</snmp-device-alarmpoints>

<snmp-device-notifiers>
  PageFred: "Fred via Pager:0:0:0"
</snmp-device-notifiers>
```

In this example, either of the SiteTemp alarms triggers the "PageFred" notifier. Looking farther down in the `<snmp-device-notifiers>` section, we see that "PageFred" sends the notification to the "Fred via Pager" (which is defined in the **Notification list**.)

Probe Calculations

InterMapper can compute values from data retrieved from devices, including SNMP MIB variables, round-trip time, packet loss, availability, etc. The results of these computations can be compared to thresholds to set device status and indicate problems.

InterMapper's Expression Syntax has the following features:

- Supports arithmetic expressions using +, -, *, /, %, and unary minus.
- Supports the use of parentheses to group sub-expressions for calculation first.
- Stores all intermediate and final results as double-precision floating point numbers.
- Supports relational operators <, >, <=, >=, =, <>, ==, !=. The value for TRUE is "1.0". The value for FALSE is "0.0".
- Supports short-circuit logical operators 'and', 'or', 'not' as well as &&, ||, !, .
- Supports variables and functions from a provided symbol table. Variables may use \$var syntax or \${var} syntax.
- Supports built-in functions for bitwise operations, rounding, and other common mathematical functions.
- Supports embedded string comparisons and simple regular expression tests. A variable in double-quotes will be treated as a string. All double-quoted strings are interpolated for variables in a Perl-like fashion. The use of + as the concatenation operator is supported. See below for [an example that uses Regular Expressions \(Pg 77\)](#).

The set of capabilities are derived from C, Perl, Excel, and expr(1).

Reserved keywords

- and
- or
- not

Precedence Table (Least to Most)

1. Assignment: :=
2. Conditional Expression: ?:
3. Logical Or: 'or', ||
4. Logical And: 'and', &&
5. Equality Tests: ==, =, !=,
6. Relational Tests: <, >, <=, >=
7. Addition, Subtraction, Concatenation: +, -
8. Multiplication, Division, Modulo: *, /, %
9. String Matching: =~, !~
10. Unary: -, !, 'not'

Built-in Numeric Functions

- `abs(x)` - Absolute value of 'x'.
- [round \(Pg 70\)](#)(x), `round(x,y)` - Round 'x' to nearest integer.
- `trunc(x)` - Remove all digits after the decimal point; e.g. `trunc(3.987) = 3`.
- `min(x1, x2, ... , xn)` - Minimum value of x1, x2, ..., xn.
- `max(x1, x2, ... , xn)` - Maximum value of x1, x2, ..., xn.
- `bitand(x, y)` - Bitwise 'and' of 'x' and 'y'.
- `bitor(x, y)` - Bitwise 'or' of 'x' and 'y'.
- `bitlshift(x, y)` - Bits of 'x' shifted left by 'y' bits.
- `bitrshift(x, y)` - Bits of 'x' shifted right by 'y' bits.
- `bitxor(x, y)` - Bitwise exclusive-or of 'x' and 'y'.
- `sin(x)` - sine of 'x' where 'x' is in radians.
- `cos(x)` - cosine of 'x' where 'x' is in radians.
- `tan(x)` - tangent of 'x' where 'x' is in radians.
- `pi()` - value of PI (e.g. 3.14159...)
- `pow(x, y)` - 'x' to the power of 'y'.
- `sqrt(x)` - square root of 'x'.
- `exp(x)` - e to the power of 'x', where e is the base of the natural logarithms.
- `log(x)` - natural logarithm of 'x'.
- `log(x, y)` - logarithm of 'x' to base 'y', e.g. `log(100, 10) = 2`
- `time()` - Time in seconds since 1 January 1970 UTC.

Built-in String Functions

- [defined \(Pg 70\)](#)(str) - Takes a string argument, and returns a non-zero value (1) if the variable name specified in the input string is defined.
- [strlen \(Pg 71\)](#)(str) - Returns the length in bytes of the string 'str' or the combined length of all string arguments.
- [sprintf \(Pg 71\)](#)(fmt, ...) - Returns formatted string using format specifier 'fmt'. Format specifier 'fmt' contains format codes that begin with '%'.
 • [strftime \(Pg 73\)](#)(fmt, [secs]) - Returns formatted date/time string using format specifier 'fmt'.
- [strptime \(Pg 74\)](#)(str, fmt) - Returns the number of seconds since UTC 1970 represented by the given date/time string, as interpreted using the specified format code.
- [subid \(Pg 74\)](#)(oid, start, length) - Gets the specified length sub-OIDs from a given OID string, starting from index start (the index starts from 0).
- [substr \(Pg 75\)](#)(str, offset, len)
- [unpack \(Pg 76\)](#)(binary str, fmt)
- Regular Expressions - See below for [an example that uses Regular Expressions \(Pg 77\)](#).

Function Descriptions

defined

FUNCTION defined(variable:STRING):INTEGER;

Returns a non-zero value (1) if the variable name specified in the input string is defined (it has already been assigned a value).

Note: This function takes a string argument. Note the usage below.

Example:

```
defined("var2") = 1 ? "$var2 is defined" : "$var2 is undefined"
```

round

FUNCTION round(x:DOUBLE, y:INTEGER):DOUBLE;

FUNCTION round(x:DOUBLE):INTEGER;

Rounds a given double value (**x**) to the nearest integer or to the given number of decimal places (**y**).

Examples:

```
round(8.6) --> 9
round(3.14159, 3) = 3.142
```

strlen

FUNCTION strlen(str[, ...])

Returns the length of the string ***str*** in bytes.

Returns the combined length of all string arguments in bytes.

Examples:

```
strlen( "Dartware" ) --> 8
strlen( "Dartware", "2000" ) --> 12
```

sprintf

FUNCTION sprintf(fmt, ...)

Returns formatted string using format specifier ***fmt***. Format specifier ***fmt*** contains format codes that begin with '%'. The following format codes are supported:

- c - Formats numeric argument as ascii character
- d - Formats numeric argument as decimal integer
- o - Formats numeric argument as octal integer
- x - Formats numeric argument as hexadecimal number (lower case)
- X - Formats numeric argument as hexadecimal number (upper case)
- u - Formats numeric argument as decimal integer (unsigned)
- s - Formats argument as an ascii string (NUL terminated)
- a - Formats argument as a hexadecimal string with bytes separated by ':'
- f - Formats numeric argument as floating point (fixed precision)
- e - Formats numeric argument as floating point (scientific notation)
- g - Formats numeric argument as floating point (easy to read)
- % - Prints a percent sign

The general specification for a format code is:

```
% [-] [<width>] [. <precision> ] <code>
```

String Formatting

For string data using %s, the width specifies the minimum width of the output field, and the precision specifies the number of characters to output. If the number of output characters is less than the minimum field width, the output is padded with spaces.

Example:

```
sprintf( "%10s", "Dartware" )
Results in "   Dartware"
sprintf( "%s", "Dartware" )
Results in "Dartware"
```

The default alignment is to the right; so padding is added to the beginning of the string. To left align the output of %s, you need to include a '-' immediately following the '%':

```
sprintf( "%-10s", "Dartware" )
Results in "Dartware "
sprintf( "%-10.4s", "Dartware" )
Results in "Dart      "
```

Integer Formatting

Integers format similar to strings, except the <precision> field specifies the maximum field width, and this is enforced by padding with 0's if necessary.

```
sprintf( "%5d", 12 )
Results in "   12"
sprintf( "%-5d", 12 )
Results in "12   "
sprintf( "%6.5d", 12 )
Results in " 00012"
sprintf( "%-2X", 15 )
Results in "F  "
sprintf( "%-2.2x", 15 )
Results in "0f"
```

Floating Point Formatting

The floating point format codes use the <precision> field to specify the number of decimal places following the decimal point. %f uses the format '[-]ddd.ddd', and %e uses the format '[-]d.ddde+-dd'.

```
sprintf( "%f", 1/2 )
Results in "0.500000"
sprintf( "%5.3f", 1/2 )
Results in "0.500"
sprintf( "%e", 1/2 )
Results in "5.000000e-01"
sprintf( "%g", 1/2 )
Results in "0.5"
```

Address Formatting

The %a format code outputs a string in hexadecimal.

```
sprintf( "%a", "\x01\x02\x03" )
Results in "01:02:03"
sprintf( "%a", "Dartware" )
Results in "44:61:72:74:77:61:72:65"
```


strftime

FUNCTION `strftime(fmt [, time])`

Returns formatted date/time string using format specifier 'fmt'. Format specifier 'fmt' contains format codes that begin with '%'. If a time argument is provided, it must be in seconds since UTC 1970. If no time argument is provided, it defaults to the current time. The following format codes are supported on all platforms:

- a - Abbreviated weekday name
- A - Full weekday name
- b - Abbreviated month name
- B - Full month name
- c - Date and time formatted something like "Tue Feb 06 10:25:22 2007"
- d - Day of month (01-31)
- H - Hour number (00-23)
- I - Hour number (01-12)
- j - Day of the year number (001-366)
- m - Month number (01-12)
- M - Minute number (00-59)
- p - "AM" or "PM"
- S - Second number (00-61)
- U - Week of the year number (00-53). First Sunday is day 1 of week 1.
- w - Weekday number (0-6). Sunday is 0.
- W - Week of the year number (00-53). First Monday is day 1 or week 1.
- x - Date.
- X - Time.
- y - Two-digit year number (00-99)
- Y - Year with century (e.g. 2007)
- z - Time zone.
- % - Prints a percent sign

The `strftime` function is implemented using the identically named function in the underlying system. Other format codes may work, but these are not portable.

```
strftime( "%c")
Results in "Tue Feb  6 11:19:24 2007"
strftime( "%Y-%m-%d", 1170778895)
Results in "2007-02-06"
```

strptime

FUNCTION `strptime(str , fmt)`

Returns the number of seconds since UTC 1970 represented by the given date/time string, as interpreted using the specified format code. Basically, this function can be used to parse dates.

This function uses the same underlying format codes as `strftime`.

Example:

```
strftime( "%Y", strptime( "1990", "%Y"))
Results in "1990"
```

subid

FUNCTION `subid(oid, start, length)`

Gets the specified length sub-OIDs from a given OID string, starting from index start (the index starts from 0). When the start index is negative, it will be counted from the end of the OID string.

Examples:

```
subid("1.3.6.1.2.1.4.20.1.1.10.10.2.20", 0, 2)    --> "1.3"
subid("1.3.6.1.2.1.4.20.1.1.10.10.2.20", -4, 4)   --> "10.10.2.20"
subid("\1.3.6.1.2.1.4.20.1.1.10.10.2.20\", 0, 2)  --> \"1.3\"
subid("\1.3.6.1.2.1.4.20.1.1.10.10.2.20\", -4, 4) --> \"10.10.2.20\"
subid("\1.3.6.1.2.1.4.20.1.1.10.10.2.20\", 4, 4)  --> \"2.1.4.20\"
subid("\1.3.6.1.2.1.4.20.1.1.10.10.2.20\", -2, 4) --> \"2.20\"
subid("\1.3.6\", 3, 4)    --> \"\"
subid("\1.3.6\", 2, 4)    --> \"6\"
subid("\1.3.6\", -4, 4)   --> \"1.3.6\"
subid("\1.3.6\", -2, 4)   --> \"3.6\""
```

substr

FUNCTION substr(*str*:STRING, *offset*:INTEGER):STRING;
FUNCTION substr(*str*:STRING, *offset*:INTEGER, *length*:INTEGER):STRING;

Extract a substring out of ***str*** and return it. The substring is extracted starting at ***offset*** characters from the start of the string.

- If ***offset*** is negative, the substring starts that far from the end of the string instead; ***length*** indicates the length of the substring to extract.
- If ***length*** is omitted, everything from ***offset*** to the end of the string is returned.
- If ***length*** is negative, the length is calculated to leave that many characters off the end of the string. If neither ***offset*** nor ***length*** is supplied, the function will return ***str***. (See Perl *substr*).

Examples:

```
substr( "0123456789", 7)      -->  "789"  
substr( "0123456789", 4, 2)  -->  "45"  
substr( "0123456789", 4, -2) -->  "4567"  
substr( "0123456789", -2, 1) -->  "8"
```

unpack

FUNCTION `unpack(str:STRING, format:STRING):VALUE`

Take a string **str** representing a data value and convert it into a scalar value. The format string specifies the type of value to be unpacked. (See perl *unpack*).

- If the input string is shorter than the expected number of bytes to be unpacked, treat the input string as if it is padded with zero bytes at the end.

```
unpack("\1\2\3", ">L")
```

is the same as

```
unpack("\1\2\3\0", ">L")
```

- If the input string is longer, the remaining bytes in the input are ignored.
- If the endian modifier is not supplied, we will use the target platform's byte order (little endian on Windows, big endian on Mac).
- If format specifier is not supplied, the function will return *str*.

Format specifier	Description
c	signed character value (1 byte)
C	unsigned character value (1 byte)
l	signed long value (4 bytes)
L	unsigned long value (4 bytes)
s	signed short value (2 bytes)
S	unsigned short value (2 bytes)
#B	a base64 string (all bytes)
>	big-endian modifier
<	little-endian modifier
H	decodes the given hexadecimal value and returns an integer (up to 32-bits)
#H	decodes the given hexadecimal value and returns a string

Examples:

```
unpack( "\1", "c")           --> 1
unpack( "\1\2\3\4", ">L")     --> 16909060
unpack( "\1\2\3\4", "<L")     --> 67305985
unpack( "BS64", "#B")        --> "... " (where "... "
is "\x5\x2e\xb8")
unpack( "\1\2\3", ">L")       --> 16909056
unpack( "1F", "H")           --> 31
unpack( "42696c6c207761732068657265", "#H") --> "Bill was here"
```

Note: The `unpack()` function supports one format code in the format string.

Using Regular Expressions In Custom SNMP Probes

You can use a regular expression to divide a string into separate variables after retrieving it from a device.

Use the example below:

A customer had a piece of equipment that returned the following information in sysDescr.0:

```
FW TR6-3.1.4Rt_F213E4, 2.4GHz, 0dBi ext. antenna
```

They wanted to display several interesting values from this string, including the firmware version ("FW"), the frequency, and the antenna used.

They created a probe that retrieved sysDescr.0 and then parsed out those strings with the following commands in the <snmp-device-variables> section of the probe:

```
<snmp-device-variables>
  sysDescr, 1.3.6.1.2.1.1.1.0,
  DEFAULT, "system description"
  firmware, "$sysDescr" =~ "^FW ([^,]+), (.+)Hz, (.+) antenna"
; "${1}", CALCULATION, "Firmware"
  frequency, "${2}",
  CALCULATION, "Frequency"
  antenna, "${3}",
  CALCULATION, "Antenna"
</snmp-device-variables>
```

Here is an annotated description of the above four lines:

1. Retrieve sysDescr.0 (OID of 1.3.6.1.2.1.1.1.0) and assign it to the variable \$sysDescr.
2. Set the value of \$firmware based on the calculation. There are many things going on in this line:
 - The "=~" operator indicates that the \$sysDescr variable should be parsed using the regular expression string that follows.
 - This regular expression breaks the string at the comma characters. The "[^,]" matches any single character that isn't a comma; adding a "+" forms a pattern that matches multiple non-comma characters.
 - Parentheses around a pattern serve to memorize a string. Each pair of paren's matches a string whose value is placed in variables numbered \${1}, \${2}, \${3}, etc.

- The semicolon followed by "\${1}" indicates that the entire CALCULATION should return the value of \${1} as a string.
 - The variable \$firmware thus gets assigned the value of \${1}
3. Assign the variable \$frequency with the result of the second match (\${2}).
 4. Assign the variable \$antenna with the result of the third match (\${3}).

Note: It is beyond the scope of this manual to describe the full capabilities of regular expressions. There are a number of tutorials available on the web. One example is the [Perl Regular Expression Tutorial](#).

Probe Properties

The `<snmp-device-properties>` section specifies certain aspects of the SNMP queries sent to the device. Like other sections, it is closed with a `</snmp-device-properties>` tag. For example:

```
<snmp-device-properties>
  nomib2           = "true"
  pdutype          = "get-request"
  apcups           = "false"
  maxvars          = "10"
  interface_numbered = ($ifIndex == 2 or $ifDescr =~ "en2")
  interface_visible = ($ifIndex == 2 or $ifDescr =~ "en2")
</snmp-device-properties>
```

The properties that may be set include:

- **nomib2="true"** -- InterMapper does not query the sysUptime MIB-2 variable.
- **pdutype="get-request"** -- InterMapper uses SNMP Get-Request, instead of Get-Next-Request queries.
- **apcups="false"** -- If apcups is false, InterMapper will not query the APC-UPS MIB even for devices that auto-detect as one.
- **maxvars="10"** -- maxvars controls the maximum number of variables to put in each SNMP request. If a custom probe requires more variables than maxvars, InterMapper sends multiple queries containing up to maxvars variables.
- **interface_visible = <expression>** - specifies a "filter expression" for use in determining which interfaces are made visible. By default, InterMapper makes the numbered interfaces visible. Setting this property allows you to make certain *unnumbered* interfaces visible if they match the expression that can use \$ifIndex, \$ifDescr, \$ifType, or \$ifAlias variables.
Note: This property does not allow you to make numbered interfaces hidden.
- **interface_numbered = <expression>** - specifies a "search expression" for use in determining which interface is made numbered. By default, the ipAddrTable specifies which interface is numbered. This property allows the probe file to override that choice.

Specifying SNMP OIDs in Custom Probes

Introduction

InterMapper supports two kinds of OID's: Numeric and Symbolic. The Symbolic OIDs become available when a MIB has been imported into InterMapper.

In addition, InterMapper supports three kinds of OID expressions: Get-Next, Trap-Conditional, and Index-Derived.

Numeric OID's

Numeric OID's contain only numbers separated by periods. Preceding periods are ignored. A trailing period is allowed if there is only one subid.

Examples:

```
.1  
1.  
1.3.6
```

Invalid examples:

```
1 (no period)  
1.3.6. (trailing period but with multiple subids)  
1.3.6.blah (not numeric)  
1.3.6.1.2.1.system.sysUpTime.0 (not numeric)
```

Unlike Net-SNMP, InterMapper ascribes no special meaning to OID's that begin with a period; all numeric OID's are considered absolute.

Errors in numeric OID's are reported by the system to the Event Log when the error is in a custom probe. The error message will have the form:

```
Syntax error in OID "1.3.6.1..1.2"
```

Symbolic OID's

A symbolic OID begins with a letter, after ignoring any preceding periods. InterMapper must be able to locate a MIB file that defines the symbols used.

There are three types of symbolic OID's:

1. Simple symbols specify a starting symbol and zero or more trailing subid's.
2. Relative symbols specify a starting symbol and one or more subid symbols.
3. Scoped symbols specify the name of the MIB, the scope operator ::, followed by a simple or relative symbol.

Relative and scoped symbols are handy when a symbol is ambiguous, i.e. the same symbol name is defined differently in two separate MIB files. You should prefer the scoped OID form, when possible.

Symbolic names are case-sensitive.

Examples:

```
Simple: sysUpTime
        sysUpTime.0
        enterprises.9.2.3.4.5

Relative: system.sysUpTime
          system.sysUpTime.0

Scoped: SNMPv2-MIB::sysUpTime
        SNMPv2-MIB::system.sysUpTime.0
```

Invalid Examples:

```
Simple: sysUpTiime    (misspelled; not found)
        sysupTime    (wrong case)
        sys%pTime    (disallowed character %)
        sysUpTime.0.  (bad; trailing period)

Relative: system.ifIndex (bad; ifIndex isn't under system)

Scoped: SNMPv2-MIB.sysUpTime (bad; must use :: for scoped OID)
        IF-MIB::sysUpTime    (bad; wrong MIB module for sysUpTime)
```

Errors in symbolic OID's are reported by the system to the Event Log when the error is in a custom probe. The error message will have the form:

```
Syntax error in OID "sys%pTime"
```

OIDs indexed by Strings

Certain MIBs specify tables that are indexed by strings. The net-snmp documentation at

<http://www.net-snmp.org/tutorial/tutorial-5/commands/output-options.html>

describes this. A convenient way to enter these OIDs is:

```
NET-SNMP-EXTEND-MIB::nsExtendOutLine."LOG"
```

and a SNMP variable could be created like this:

```
outLine, NET-SNMP-EXTEND-MIB::nsExtendOutLine."LOG",
DEFAULT, ""
```

Limitations of Symbolic OID's

1. Symbolic OID's will only work if the necessary MIB file is loaded into InterMapper. If InterMapper cannot resolve the symbolic OID using a MIB file, this is considered a syntax error in the symbolic OID. At this time, there is no way to bundle a MIB file with a probe as one file; this is a future direction.
2. It may happen that two or more MIB files define the same symbol. When this happens, InterMapper may pick the wrong definition. You can avoid this by using the scoped OID form.

Get-Next OID Expressions

InterMapper has a special syntax for "get-next" style OID's - attach a + to the end of the OID.

Normally, when you specify a variable to query in a custom SNMP probe, you specify the complete OID, including the instance. For example, you might specify "sysUpTime.0" or "ifInOctets.13". For sysUpTime, the .0 specifies the (only) instance. For ifInOctets, the .13 specifies the value for ifIndex 13.

There are occasions when you want to query a variable using a preceding OID. For example, you might want to query the value of ifInOctets for the first interface, but you can't assume the ifIndex of the first interface is 1. Here's how you would specify the OID:

```
ifInOctets+
```

To retrieve the value of ifInOctets for the interface whose ifIndex follows 13, specify the OID with a plus:

```
ifInOctets.13+
```

The plus sign must immediately follow the OID. Technically, it's not part of the OID, but considered an operator in InterMapper's OID expression language.

Note: Get-Next OID expressions will not work for custom SNMP probes that specify get-request queries.

Trap-Conditional OID Expressions

Trap-conditional OID expressions allow you to assign a variable only when it occurs in the varbind list of a certain trap. For example, you might want to set the value of your probe's "sysUpTimeCrashed" variable to value of the "sysUpTime.0" variable included in the varbind list of a "systemCrashed" trap. However, you don't want to set "sysUpTimeCrashed" when you see the sysUpTime.0 value in any other received trap. To restrict the assignment of sysUpTime.0 to only the systemCrashed trap, you need to specify both the systemCrashed trap OID and the sysUpTime.0 OID using the ?: operator. This combination is called a "Trap-Conditional" OID, or "Trap OID" for short.

Examples:

```
systemCrashed?:sysUpTime.0
sysTrapOID?:sysContact
SOMEMIB::sysTrapOID.1?:SMIV2-MIB::sysContact
```

Supported in 4.4, the legacy format for trap OID's is a numeric OID followed by an OID:

```
1.3.6.1.2.1::sysUpTime.0
```

The legacy format does not allow use of a symbolic name for a trap OID; this conflicts with the scoped format above. The use of :: for Trap-conditional OID expressions is deprecated. Please use ?: in the future.

Index-Derived OID Expressions

When querying tables from SNMP devices, it is often useful to assign the value of a variable from a row's OID index. This technique will work even if the values used to index the row have an access of "not-accessible".

Take ipNetToMediaTable as an example. This table is indexed by two variables: ipNetToMediaIfIndex and ipNetToMediaNetAddress. The other two variables in this table are ipNetToMediaPhysAddress and ipNetToMediaType. Since the format of the table's row index is known, you only need to query ipNetToMediaPhysAddress and ipNetToMediaType; you can derive ipNetToMediaIfIndex and ipNetToMediaNetAddress from the index value.

To specify the value of an OID, derived from another OID, you can use the Index-Derived OID Expression.

Examples:

```
ipNetToMediaType[0:1]  
ipNetToMediaType[1:4]  
ipNetToMediaType[1:]
```

In this example, the notation <oid>[a:b] means to fetch the OID <oid> and retrieve the value from the subid's of the index as follows:

```
[a:b] means to start with the a'th subid and collect b subid's.  
[a:] means to start with the a'th subid and collect the rest.  
subid's use zero-based indexes.
```

Index-derived OID expressions are only valid when querying SNMP tables.

Enumerated Values In SNMP Probes

An enumeration in a MIB allows a designer to represent text strings as numeric values. This numeric representation is more compact, and therefore efficient, but makes it hard for humans to read. InterMapper has several techniques for displaying the enumerated values for easier interpretation.

There are several ways to do this:

1. InterMapper can automatically display these "enumerations" if the MIB has been imported. To do this, simply declare the variables to be of type "STRING" instead of "DEFAULT". This is described in [Probe Variables](#).
2. If no MIB file is available, you can create a calculation variable to select a string based on the numeric value returned. If you have two choices, you should use the "xxxx ? yyyy : zzzz" expression that can be read as:

```
if xxxx is true then return yyyy      otherwise return zzzz
```

The calculation variable will look like this:

```
xxxxStr, ($xxxx = 0 ? "yyyy" : "zzzz"), CALCULATION,  
"replacement string for xxxx"
```

3. If there are three or more possibilities, you can chain the expression in this format:

```
aaaa ? bbbb : cccc ? dddd : eeee ? ffff : gggg
```

which can be read as:

```
if aaaa is true then  
  return bbbb  
else if cccc is true  
  return dddd  
else if eeee is true  
  return ffff  
else return gggg
```

Generally, the aaaa, cccc, and eeee test to see if a single variable is equal to 1, 2, 3, etc. The calculation variable then looks like this:

```
aaaaStr, ($aaaa=0 ? "bbbb" : $aaaa=1 ? "dddd" : $aaaa=2 ?  
"ffff" : "gggg"), CALCULATION, "replacement string for aaaa"
```

About Custom SNMP Trap Probes

```
type = "custom-snmp-trap"
```

A *trap* is an unsolicited packet sent from a device to InterMapper (or other SNMP management console). The trap generally contains one or more data values that give information about the device's state.

When a trap arrives, InterMapper first determines which device(s) on the enabled maps should receive information from the trap. InterMapper examines the Agent Address (for relayed traps) or the Source IP address, and passes a copy of the trap packet to *each* device on the maps whose IP address matches. For example, if a device with the IP address is on two maps, or is present twice on the same map, each of those devices will receive a copy of the trap.

InterMapper then parses out all the values from the trap and assigns them to trap variables for use in the remainder of the probe. InterMapper then re-evaluates the expressions in the probe, and sets the device status appropriately. If a particular trap variable is not set by an incoming trap, expressions containing that variable are *not* evaluated. See the [The <snmp-device-variables> Section for Traps \(Pg 87\)](#) section below for details of defining trap variables.

Finally, as a result of receiving the trap, InterMapper re-polls the device that sent the trap. This guarantees that InterMapper has the most up-to-date information about the device's state. If another trap arrives before the final response of this new poll has returned, InterMapper will complete the current poll and initiate another round of polling to get the new state.

Note: A trap is sent as a UDP packet. If something on your network is causing packet loss, it is possible to lose a trap packet. Therefore, Dartware recommends that you don't rely completely on traps as a means for monitoring the health of a device. There is no substitute for regular polling.

Defining Trap-Only Probes

By default, InterMapper's custom SNMP probes will query certain MIB-II system group items. To define a probe that only receives traps (and does not actively poll a device using SNMP GetRequest or GetNextRequest), you should use a probe type of "custom-snmp-trap". Thus, the <header> section of the file should look like this:

```
<header>
  "type" = "custom-snmp-trap"
  ... etc ...
</header>
```

The <snmp-device-variables> Section For Traps

A **Trap Variable** is a variable defined in a custom probe file whose value is set to a value received in a trap. InterMapper has three kinds of Trap Variables, only one of which can be declared in a probe:

- **Packet Trap Variables** - a set of variables automatically set by InterMapper when a trap is received.
- **Positional Trap Variables** - a set of variables automatically set by InterMapper. Use positional Trap Variables to access data from the trap's VarBind list by position in the list.
- **Named Trap Variables** - variables you define by associating an SNMP OID with a name. If the OID exists in the trap's VarBind list, the variable is set to the value in the trap.

A Trap Variable will never be polled: that is, InterMapper never sends an SNMP GetRequest or GetNextRequest to retrieve its value.

Packet Trap Variables

In addition to the variables in the VarBind List, a probe can set variables based on the fields of the trap packet's header.

- **\$GenericTrap** - The GenericTrap field in the trap (SNMPv1). This field can take on the values:
 - 0 - coldStart;
 - 1 - warmStart;
 - 2 - linkDown;
 - 3 - linkUp;
 - 4 - authenticationFailure;
 - 5 - egpNeighborLoss;
 - 6 - An enterprise-specific value.
- **\$SpecificTrap** - The value of the SpecificTrap field in the trap. If the \$GenericTrap value is 0-5, the \$SpecificTrap is zero (0); otherwise it is a positive 32-bit value specified by the vendor (SNMPv1).
- **\$TimeStamp** - The TimeStamp field in the trap, in hundredths of a second.
- **\$Enterprise** - The value of the SNMPv1 enterprise field (SNMPv1)
- **\$CommunityString** - The value of the CommunityString field in the trap (SNMPv1, SNMPv2c).
- **\$TrapOID** - The value of the TrapOID field in the trap (SNMPv2c, SNMPv3).
- **\$AgentAddress** - The IP address of the SNMP agent that generated the trap.
- **\$SenderAddress** - The IP address of the device that sent the trap. This could be different from the \$AgentAddress when the sender is forwarding traps for the agent.
- **\$SnmVersion** - Represents the version of the trap. Values can be 0 (v1), 1 (v2c) or 3 (v3).
- **\$VarbindCount** - The number of variables contained in the VarBind list.

Positional Variables from the Varbind List

You can access values from the VarBind List by ***position*** using variables of the form:

- **\$VarbindValueN** - The value of the N'th variable in the trap's VarBind List
- **\$VarbindTypeN** - The type of the N'th variable in the trap's VarBind List
- **\$VarbindOIDN** - the OID of the N'th variable in the trap's VarBind List

Note: N may be from 1 to 50.

Named Trap Variables

The only way to set a named Trap Variable value is to receive a trap that contains the OID in its VarBind List, or the set the named variable to the value of a positional variable. The [Probe Variables section \(Pg 43\)](#) of this document describes the file format. Here is an example:

```
<snmp-device-variables>
  InterMapperTimeStamp, 1.3.6.1.4.1.6306.2.1.1.0, TRAPVARIABLE,
  "Timestamp"
</snmp-device-variables>
```

In this example, the variable \$InterMapperTimeStamp is set every time a trap arrives containing the OID 1.3.6.1.4.1.6306.2.1.1.0 in the VarBind List. Trap Variables that don't have values set by an incoming trap are left undefined.

A full [example trap file \(Pg 93\)](#) is available.

Here's how several useful trap variables might be defined

```
<snmp-device-variables>
  genericTrapVar,      $GenericTrap,      TRAPVARIABLE,      "Generic
Trap"
  specificTrapVar,    $SpecificTrap,      TRAPVARIABLE,      "Specific
Trap"
  timeStampVar,       $TimeStamp,         TRAPVARIABLE,      "Timestamp"
  enterpriseVar,      $Enterprise,        TRAPVARIABLE,
  "Enterprise"
  commStringVar,      $CommunityString,   TRAPVARIABLE,      "Community
String"
  trapOIDVar,         $TrapOID,           TRAPVARIABLE,      "Trap OID"
  agentAdrsVar,       $AgentAddress,      TRAPVARIABLE,      "Agent
Address"
  senderAdrsVar,      $SenderAddress,     TRAPVARIABLE,      "Sender
Address"
  snmpVersionVar,     $SnmVersion,        TRAPVARIABLE,      "SNMP Ver-
sion"
  varbindCountVar,    $VarbindCount,      TRAPVARIABLE,      "Varbind
Count"
  — the first and second values from the Varbind List by position
  trap_var1, $VarbindValue1, TRAPVARIABLE, "First value"
  trap_var2, $VarbindValue2, TRAPVARIABLE, "Second value"
</snmp-device-variables>
```

Note: The TRAPVARIABLE type causes the value to be displayed in the most useful format. You can also use one of following to force the display to a certain format. These variables are equivalent to their non-trapvariable counterparts, whose complete descriptions of the formats are available in [Probe Variables \(Pg 43\)](#):

- TRAPVARIABLE-TOTAL-VALUE
- TRAPVARIABLE-PER-SECOND
- TRAPVARIABLE-PER-MINUTE
- TRAPVARIABLE-STRING*
- TRAPVARIABLE-INTEGER
- TRAPVARIABLE-HEXADECIMAL*
- TRAPVARIABLE-HEXNUMBER
- TRAPVARIABLE-DOUBLE

* STRING and HEXADECIMAL are both strings; they can't be charted.

Accessing Trap Variables by Position

When accessing VarBind List entries, you can access them either by name or by position. Access by name is much easier to program and understand, but there have been instances where a vendor's traps contained VarBind List entries with the same name: in those cases you must get their values by position. Below are examples of accessing VarBind List entries by name and by position.

Given this trap, InterMapper creates the following Event Log entry:

```
03/23 11:37:34 TRAP IC3 Demo System:Video Stream ENC01 LIVEWAVE-
MIB::deviceFaulted (v2c)
  { LIVEWAVE-MIB::deviceUnitID : "5",
    LIVEWAVE-MIB::deviceName : "5 - Video Stream",
    LIVEWAVE-MIB::deviceStatus : "6" }
```

The trap contains these three values: deviceUnitID, deviceName, and device-Status. (InterMapper has already imported a LIVEWAVE MIB that defines these OIDs.)

The variables are declared in the variables section:

```
<snmp-device-variables>
  deviceUnitID, LIVEWAVE-MIB::deviceUnitID, TRAPVARIABLE, "Device
  Unit ID"
  deviceName, LIVEWAVE-MIB::deviceName, TRAPVARIABLE, "Device
  Name"
  deviceStatus, LIVEWAVE-MIB::deviceStatus, TRAPVARIABLE, "Device
  Status"
</snmp-device-variables>
```

When a trap is received, the probe variables above are set to the values of the trap variables from the VarBind list. One way you could use them is as follows:

```
<snmp-device-thresholds>
  critical: deviceStatus = 3 "Problem with $deviceUnitID $device-
  eName: Device status = $deviceStatus"
  okay: deviceStatus = 1 "$deviceUnitID $deviceName functioning
  normally."
</snmp-device-thresholds>
```

You can also access the variables by position in the VarBind list:

```
<snmp-device-variables>
  deviceUnitID, LIVEWAVE-MIB::$VarbindValue1, TRAPVARIABLE, "Device
  Unit ID"
  deviceName, LIVEWAVE-MIB::$VarbindValue2, TRAPVARIABLE, "Device
  Name"
  deviceStatus, LIVEWAVE-MIB::$VarbindValue3, TRAPVARIABLE, "Device
  Status"
</snmp-device-variables>
```

The `<snmp-device-display>` Section

Use the `<snmp-device-display>` section to format the device's Status window exactly the same way as you do in an SNMP Probe. For more information, see the SNMP Probe's [<snmp-device-display> Section](#) topic.

Trap Viewing and Logging

The contents of trap message are logged to the Event Log file at the time the trap is received. There are two forms: Short and Verbose. (The format is controlled by the **Verbose Trap Logging** checkbox in the **Server Settings > SNMP** preference pane.)

Short Trap Format

```
06/08 20:50:29 TRAP TestMap:192.168.2.1 1.3.6.1.4.1.6306 (333)
{ "321", "456" } (via 192.168.1.233)<p>
```

Verbose Trap Format:

```
06/08 20:50:05 TRAP TestMap:192.168.2.1 1.3.6.1.4.1.6306 (333)
{ 1.3.6.1.4.1.6306.99.1 : "321", 1.3.6.1.4.1.6306.99.2 : "456" }
(via 192.168.1.233)<p>
```

The fields of the trap entry in the log file are defined below, with examples in "[...]":

- **Date and Time** - [06/08 20:50:05 TRAP]
The date and time followed by the word "TRAP"
- **Map Name and Device ID** - [TestMap:192.168.2.1]
The map name and device ID, separated by a colon (":")
- **Enterprise OID and Trap Field** - [1.3.6.1.4.1.6306 (333)]
The Enterprise OID, followed by the specific trap field in paren's
- **VarBind List** - The contents of the VarBind List, enclosed in curly braces, and separated by commas.
Short format: { "321", "456" } shows only the values sent for each VarBind in quotes.
or
Verbose format: { 1.3.6.1.4.1.6306.99.1 : "321", 1.3.6.1.4.1.6306.99.2 : "456" }
shows the OID, a colon (":"), and the OID's value in quotes.
- **Address** - [(via 192.168.1.233)] The address of the relaying computer, if present.

The verbose format shows all the information that was sent with the trap.

Example - Trap Viewer Probe

The following example shows the handling of traps.

```
<!--
  SNMP Trap Viewer probe (com.dartware.snmp.trapdisplay.txt)
  Probe for InterMapper (http://www.intermapper.com)

  Copyright (c) 2007, Dartware, LLC.
  Feel free to use this as the basis for creating new probes.

  25 Apr 2005 Original version - reb
  4 May 2005 Changed to "custom-snmp-trap" -reb
                Modified for IM 4.4 header/display items.
  8 May 2007 Added special trap variables to the probe and display
-reb
  29 May 2007 Changed probe name to "Trap Display", updated descrip-
tion -reb
  1 Jun 2007 Changed probe name to "Trap Viewer"; tweaked descrip-
tion;
                left canonical name alone -reb
-->

<header>
  "type"           =      "custom-snmp-trap"
  "package"        =      "com.dartware"
  "probe_name"     =      "snmp.trapdisplay"
  "human_name"     =      "Trap Viewer"
  "version"        =      "2.2"
  "address_type"   =      "IP,AT"
  "port_number"    =      "161"
  "display_name"   =      "SNMP/Trap Viewer"
</header>

<description>
\GB\Trap Viewer Probe\P\

This probe listens for trap packets to arrive and displays the con-
tents of the
trap in the Status Window. It does not actively poll the device, nor
does it
take any action based on the trap contents.

You can view all the variables that have been parsed from the trap
packet in the
device's Status Window. You can also use this as a prototype for mak-
ing your own
trap probes.

\B\How the Trap Viewer Probe Works\p\

When a trap arrives, the probe parses the trap to get the values from
the trap's
header as well as the first ten items in its Varbind List. It assigns
all these
values to variables that can be used in the probe and displayed in
```

the Status Window.

To see how this probe works, you can configure your equipment to send traps to InterMapper, or use the `net-snmp \b\snmptrap\p\` command. Either way, the Status Window will show the values present in any traps that arrive.

For more information on the `\b\snmptrap\p\` command, read the net-snmp documentation for the `\u2=http://www.net-snmp.org/tutorial/tutorial-4/c-commands/snmptrap.html\trap tutorial\p0\` and the `\u2=http://www.net-snmp.org/docs/man/snmpinform.html\snmptrap command\0p\`. The remainder of this note shows how to send a trap with variables from the Dartware MIB:

`\i\SNMPv1 Traps\p\`

- Add a device to a map with the IP address `\i\192.168.56.78\p\`
- Set it to use this probe
- Issue the `snmptrap` command below from the command line (it should all be on one line):

```
snmptrap -v 1 -c commString localhost
1.3.6.1.4.1.6306 192.168.56.78 6 123 4567890
1.3.6.1.4.1.6306.2.1.1.0 s "05/08 23:26:35"
1.3.6.1.4.1.6306.2.1.2.0 s Critical
1.3.6.1.4.1.6306.2.1.3.0 s "Big Router"
1.3.6.1.4.1.6306.2.1.4.0 s "Critical: High Traffic"
1.3.6.1.4.1.6306.2.1.5.0 s "127.0.0.1"
1.3.6.1.4.1.6306.2.1.6.0 s "SNMP Traffic Probe"
```

`\i\SNMPv2c Traps\p\`

- Add a device to the map with an IP address of `\i\localhost\p\`
- Set it to use this probe
- Issue the `snmptrap` command below from the command line (it should all be on one line)

```
snmptrap -v 2c -c commString localhost
4567890 1.3.6.1.4.1.6306
1.3.6.1.4.1.6306 192.168.56.78 6 123 4567890
1.3.6.1.4.1.6306.2.1.1.0 s "05/08 13:26:35"
1.3.6.1.4.1.6306.2.1.2.0 s Critical
1.3.6.1.4.1.6306.2.1.3.0 s "Big Router"
1.3.6.1.4.1.6306.2.1.4.0 s "Critical: High Traffic"
1.3.6.1.4.1.6306.2.1.5.0 s "127.0.0.1"
1.3.6.1.4.1.6306.2.1.6.0 s "SNMP Traffic Probe"
```

`</description>`

`<!-- Copy/paste these lines into the terminal window for testing...`

```
snmptrap -v 1 -c commString localhost 1.3.6.1.4.1.6306 192.168.56.78
6 123
```

```
4567890 1.3.6.1.4.1.6306.2.1.1.0 s "05/08 13:26:35"
1.3.6.1.4.1.6306.2.1.2.0 s
Critical 1.3.6.1.4.1.6306.2.1.3.0 s "Big Router"
1.3.6.1.4.1.6306.2.1.4.0 s
"Critical: High Traffic" 1.3.6.1.4.1.6306.2.1.5.0 s "127.0.0.1"
1.3.6.1.4.1.6306.2.1.6.0 s "SNMP Traffic Probe"

snmptrap -v 1 -c commString localhost 1.3.6.1.4.1.6306 192.168.56.78
6 123
4567890 1.3.6.1.4.1.6306.2.1.1.0 s "05/08 13:26:35"
1.3.6.1.4.1.6306.2.1.2.0 s
Critical 1.3.6.1.4.1.6306.2.1.3.0 s "Big Router"
1.3.6.1.4.1.6306.2.1.4.0 s
"Critical: High Traffic" 1.3.6.1.4.1.6306.2.1.5.0 s "127.0.0.1"
1.3.6.1.4.1.6306.2.1.6.0 s "SNMP Traffic Probe"
1.3.6.1.4.1.6306.2.1.7.0 s
"var7" 1.3.6.1.4.1.6306.2.1.8.0 s "var8" 1.3.6.1.4.1.6306.2.1.9.0 s
"var9"
1.3.6.1.4.1.6306.2.1.10.0 s "var10" 1.3.6.1.4.1.6306.2.1.11.0 s
"var11"
1.3.6.1.4.1.6306.2.1.12.0 s "var12"

snmptrap -v 2c -c commString localhost 4567890 1.3.6.1.4.1.6306
1.3.6.1.4.1.6306.2.1.1.0 s "05/08 13:26:35" 1.3.6.1.4.1.6306.2.1.2.0
s Critical
1.3.6.1.4.1.6306.2.1.3.0 s "Big Router" 1.3.6.1.4.1.6306.2.1.4.0 s
"Critical:
High Traffic" 1.3.6.1.4.1.6306.2.1.5.0 s "127.0.0.1"
1.3.6.1.4.1.6306.2.1.6.0 s
"SNMP Traffic Probe"
—>

— The parameters in this probe are unused, but could be used to
— set thresholds for various alarms.
<parameters>
    "MinValue" = "10"
    "MaxValue" = "50"
</parameters>

<snmp-device-variables>

    — TrapVariables are updated when a trap arrives.
    — This set of variables comes from the Dartware MIB
    — and would be sent in a trap from another copy of InterMapper.

    trapTimeStamp,      1.3.6.1.4.1.6306.2.1.1.0, TRAPVARIABLE, "Times-
tamp"
    DeviceStatus,      1.3.6.1.4.1.6306.2.1.2.0, TRAPVARIABLE,
"Status"
    DeviceDNS,         1.3.6.1.4.1.6306.2.1.3.0, TRAPVARIABLE, "DNS
Name of Device"
    DeviceCondition,   1.3.6.1.4.1.6306.2.1.4.0, TRAPVARIABLE, "Con-
dition String"
    TrapSourceAdrs,    1.3.6.1.4.1.6306.2.1.5.0, TRAPVARIABLE, "Source
of trap"
    ProbeType,         1.3.6.1.4.1.6306.2.1.6.0, TRAPVARIABLE, "Probe
```

```

that generated trap"

-- Variables from the trap packet itself

genericTrapVar,      $GenericTrap,      TRAPVARIABLE, "Generic
Trap"
specificTrapVar,     $$SpecificTrap,     TRAPVARIABLE, "Specific
Trap"
timeStampVar,        $TimeStamp,         TRAPVARIABLE, "Times-
tamp"
enterpriseVar,       $Enterprise,        TRAPVARIABLE,
"Enterprise"
commStringVar,       $CommunityString,   TRAPVARIABLE, "Community
String"
trapOIDVar,          $TrapOID,           TRAPVARIABLE, "Trap OID"
agentAdrsVar,        $AgentAddress,      TRAPVARIABLE, "Address"
senderAdrsVar,       $SenderAddress,     TRAPVARIABLE, "Sender
Address"
snmpVersionVar,      $SnmpVersion,       TRAPVARIABLE, "SNMP Ver-
sion"
varbindCountVar,     $VarbindCount,      TRAPVARIABLE, "Varbind
Count"

-- Positional names of Varbind List items

vbVal1,              $VarbindValue1,     TRAPVARIABLE, "Value of Varbind1"
vbType1,             $VarbindType1,      TRAPVARIABLE, "Type of Varbind1"
vbOID1,              $VarbindOID1,       TRAPVARIABLE, "OID of Varbind1"
vbVal2,              $VarbindValue2,     TRAPVARIABLE, "Value of Varbind2"
vbType2,             $VarbindType2,      TRAPVARIABLE, "Type of Varbind2"
vbOID2,              $VarbindOID2,       TRAPVARIABLE, "OID of Varbind2"
vbVal3,              $VarbindValue3,     TRAPVARIABLE, "Value of Varbind3"
vbType3,             $VarbindType3,      TRAPVARIABLE, "Type of Varbind3"
vbOID3,              $VarbindOID3,       TRAPVARIABLE, "OID of Varbind3"
vbVal4,              $VarbindValue4,     TRAPVARIABLE, "Value of Varbind4"
vbType4,             $VarbindType4,      TRAPVARIABLE, "Type of Varbind4"
vbOID4,              $VarbindOID4,       TRAPVARIABLE, "OID of Varbind4"
vbVal5,              $VarbindValue5,     TRAPVARIABLE, "Value of Varbind5"
vbType5,             $VarbindType5,      TRAPVARIABLE, "Type of Varbind5"
vbOID5,              $VarbindOID5,       TRAPVARIABLE, "OID of Varbind5"
vbVal6,              $VarbindValue6,     TRAPVARIABLE, "Value of Varbind6"
vbType6,             $VarbindType6,      TRAPVARIABLE, "Type of Varbind6"
vbOID6,              $VarbindOID6,       TRAPVARIABLE, "OID of Varbind6"
vbVal7,              $VarbindValue7,     TRAPVARIABLE, "Value of Varbind7"
vbType7,             $VarbindType7,      TRAPVARIABLE, "Type of Varbind7"
vbOID7,              $VarbindOID7,       TRAPVARIABLE, "OID of Varbind7"
vbVal8,              $VarbindValue8,     TRAPVARIABLE, "Value of Varbind8"
vbType8,             $VarbindType8,      TRAPVARIABLE, "Type of Varbind8"
vbOID8,              $VarbindOID8,       TRAPVARIABLE, "OID of Varbind8"
vbVal9,              $VarbindValue9,     TRAPVARIABLE, "Value of Varbind9"
vbType9,             $VarbindType9,      TRAPVARIABLE, "Type of Varbind9"
vbOID9,              $VarbindOID9,       TRAPVARIABLE, "OID of Varbind9"
vbVal10,             $VarbindValue10,    TRAPVARIABLE, "Value of Varbind10"
vbType10,            $VarbindType10,     TRAPVARIABLE, "Type of Varbind10"
vbOID10,             $VarbindOID10,     TRAPVARIABLE, "OID of Varbind10"
</snmp-device-variables>

```



```
<snmp-device-display>

\B5\Information about the Trap\0P\
  \4\CommunityString:\0\ $commStringVar
  \4\  TimeStamp:\0\ $timeStampVar
  \4\  AgentAddress:\0\ $agentAdrsVar
  \4\  SenderAddress:\0\ $senderAdrsVar
  \4\  GenericTrap:\0\ $genericTrapVar \3IG\ (v1 only) \POM\
  \4\  SpecificTrap:\0\ $specificTrapVar \3IG\ (v1 only) \POM\
  \4\  Enterprise:\0\ $enterpriseVar \3IG\ (v1 only) \POM\
  \4\  TrapOID:\0\ $trapOIDVar \3IG\ (v2c only) \POM\
  \4\  SnmpVersion:\0\ $snmpVersionVar \3IG\ (0=SNMPv1; 1=SNMPv2c)
\POM\
  \4\  VarbindCount:\0\ $varbindCountVar \3IG\ (total number of Var-
binds) \POM\

\B5\Varbind List Items parsed by OID\0P\
  \4\  TimeStamp:\0\ $trapTimeStamp \3IG\ \POM\
  \4\  Device Status:\0\ $deviceStatus \3IG\ \POM\
  \4\  Device DNS:\0\ $deviceDNS \3IG\ \POM\
  \4\  Condition String:\0\ $deviceCondition \3IG\ \POM\
  \4\  Trap Source Adrs:\0\ $trapSourceAdrs \3IG\ \POM\
  \4\  Probe Type:\0\ $probeType \3IG\ \POM\

\B5\Varbind List Items by Position\0P\ \3IG\ (Varbind Value / Varbind
Type / Varbind OID) \POM\
  \4\ VarBindList #1:\0\ $vbVal1 / $vbType1 / $vbOID1
  \4\ VarBindList #2:\0\ $vbVal2 / $vbType2 / $vbOID2
  \4\ VarBindList #3:\0\ $vbVal3 / $vbType3 / $vbOID3
  \4\ VarBindList #4:\0\ $vbVal4 / $vbType4 / $vbOID4
  \4\ VarBindList #5:\0\ $vbVal5 / $vbType5 / $vbOID5
  \4\ VarBindList #6:\0\ $vbVal6 / $vbType6 / $vbOID6
  \4\ VarBindList #7:\0\ $vbVal7 / $vbType7 / $vbOID7
  \4\ VarBindList #8:\0\ $vbVal8 / $vbType8 / $vbOID8
  \4\ VarBindList #9:\0\ $vbVal9 / $vbType9 / $vbOID9
  \4\ VarBindList #10:\0\ $vbVal10 / $vbType10 / $vbOID10
</snmp-device-display>
```

The Dartware MIB

Dartware LLC has registered the Enterprise 6306 for its own SNMP variables. The remainder of this page shows the Dartware MIB in ASN.1 notation.

```

-- *****
-- DARTWARE-MIB for InterMapper and other products
--
-- May 2007
--
-- Copyright (c) 2000-2007 by Dartware, LLC
-- All rights reserved.
-- *****

DARTWARE-MIB DEFINITIONS ::= BEGIN

    IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, enterprises
        FROM SNMPv2-SMI
        DisplayString
        FROM SNMPv2-TC;

    dartware MODULE-IDENTITY
        LAST-UPDATED "200507270000Z"
        ORGANIZATION "Dartware, LLC"
        CONTACT-INFO "Dartware, LLC
            Customer Service
            Postal: PO Box 130
            Hanover, NH 03755-0130
            USA
            Tel: +1 603 643-9600
            E-mail: support@dartware.com"

        DESCRIPTION
            "This MIB module defines objects for SNMP traps sent by
            InterMapper."

        REVISION      "200705300000Z"
        DESCRIPTION
            "Updated descriptions to show timestamp format, correct
            strings for intermapperMessage."

        REVISION      "200512150000Z"
        DESCRIPTION
            "Added intermapperDeviceAddress and inter-
            mapperProbeType."

        REVISION      "200507270000Z"
        DESCRIPTION
            "First version of MIB in SMIV2."

        ::= { enterprises 6306 }

    notify          OBJECT IDENTIFIER ::= { dartware 2 }

```

```
intermapper      OBJECT IDENTIFIER ::= { notify 1 }

intermapperTimestamp OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "The current date and time, as a string, in the
format 'mm/dd hh:mm:ss'."
    ::= { intermapper 1 }

intermapperMessage OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "The type of event - Down, Up, Critical, Alarm,
Warning, OK, or Trap - as a string."
    ::= { intermapper 2 }

intermapperDeviceName OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "The (first line of the) label of the device as
shown on a map, as a
        string."
    ::= { intermapper 3 }

intermapperCondition OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "The condition of the device, as it would be
printed in the log file."
    ::= { intermapper 4 }

intermapperDeviceAddress OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "The device's network address, as a string."
    ::= { intermapper 5 }
```

```

intermapperProbeType OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS   read-only
    STATUS      current
    DESCRIPTION
        "The device's probe type, as a human-readable
string."
    ::= { intermapper 6 }

-- For SMIV2, map the TRAP-TYPE macro to the corresponding NOTI-
FICATION-TYPE macro:
--
--
-- intermapperTrap TRAP-TYPE
--     ENTERPRISE      dartware
--     VARIABLES       { intermapperTimestamp, inter-
mapperMessage,
--                     intermapperDeviceName, inter-
mapperCondition }
--     DESCRIPTION
--         "The SNMP trap that is generated by InterMapper
as a notification option."
--     ::= 1

intermapperNotifications OBJECT IDENTIFIER ::= { intermapper 0 }

intermapperTrap NOTIFICATION-TYPE
    OBJECTS { intermapperTimestamp, intermapperMessage,
              intermapperDeviceName, intermapperCondition,
              intermapperDeviceAddress, intermapperProbeType }
    STATUS current
    DESCRIPTION
        "The SNMP trap that is generated by InterMapper as a noti-
fication option."
    ::= { intermapperNotifications 1 }

END

```

Chapter 4

TCP Probes

```
type="tcp-script"
```

TCP Probes connect to the specified device and port, then execute a script that sends and receives data from the device. InterMapper examines the responses, and sets the device's status and condition based on the results.

For example, the HTTP probe connects to the specified port, then issues the commands of an HTTP request to send data to the web server, and verifies the received data. If the response is not as expected, the probe sets the device into alarm or warning.

As another example, the [TCP Example \(Pg 128\)](#) shows another TCP-based probe that connects to a device. It then sends the specified string, waits several seconds and checks the response to determine the device state.

The Custom TCP probe is shown in full as an example, and can be used as the basis for making your own probes.

Common Sections of TCP Probes

Each TCP probe follows the same general format as other probe files.

- The [<header>](#) (Pg 13) section of a command-line probe specifies the probe type, name, and a number of other properties fundamental to the operation of the probe.
- The [<description>](#) (Pg 18) section specifies the help text that appears in the Probe Configuration window. Format the description using IMML, [InterMapper's Markup language](#).
- The [<parameters>](#) (Pg 19) section defines the fields presented to the user in the Probe Configuration window.

Sections Specific to TCP Probes

Each TCP probe also has:

- Use the [<script>](#) section of a TCP probe to define a sequence of commands the probe uses to interact with and query a device, and to interpret the responses from the device. The [<script>](#) section uses the [TCP Probe Scripting Language](#), a sequential language with a [rich set of commands](#).
- The [<script-output>](#) section of a TCP probe file formats the information retrieved from the device and sends it to the device's Status window. Format the script output using IMML, [InterMapper's Markup language](#).

InterMapper's TCP Probes establish a connection to a remote system, exchange commands and receive responses, and then set the status of the device based on those responses.

This note describes how probe writers can use *regular expressions* and *comparisons* to parse out information from the responses.

The Overall Process

There is an annotated FTP probe in the [Developer Guide](#). This gives an overview of the script language and shows how it connects and logs into a FTP server, how a script can respond to error conditions, and how to set the device's status based on those conditions.

Regular Expressions

The TCP Script Language uses the MTCH command to compare a response string to expected values. It can also use a regular expression to match on a part of a string. For example:

```
MTCH "A([BCD]+)E"r else goto @NOMATCH
STOR "testval" "${1}"
```

If the incoming line contains ABDE, then the "testval" variable will contain "BD".

In the MTCH regular expression, enclosing something in parentheses turns it into a capturing subgroup. The one or more Bs, Cs or Ds that it's matching will be stored in the \${1} variable. If you have several capturing groups, they get stored in \${2}, \${3}, etc.

For more information, see [the Regular Expressions examples](#) in Probe Calculations.

Calculations in Scripts

To perform calculations within a TCP script, you should use the EVAL command. Its argument is an expression (in quotes) that will be evaluated. It usually contains an assignment (with the ":=" operator), that sets a variable to the result of the expression. For example:

```
EVAL $celsius := (($fahrenheit - 32) * 5 / 9)
```

will set the variable \$celsius to the temperature that corresponds to the \$fahrenheit variable. The value of the \$celsius variable can be used in subsequent statements.

Comparisons in Scripts

You can use the EVAL statement to make comparisons between either strings or numeric (either integer or floating point) values. To do this, write an EVAL statement that compares the two values and set the result in a new variable. If the comparison was true, then the resulting variable will be set to 1, otherwise it will be zero.

Here are examples of comparing numeric and string values:

Comparing Numeric Values

```
EVAL $x := ($val > 50.5)
NBNE #x #0 @greater
@less:
...
GOTO @ENDIF
@greater:
...
GOTO @ENDIF
@ENDIF:
```

Comparing String Values

```
EVAL $x := ("dog" > "cat")
NBNE #x #0 @dog
@cat:
...
GOTO @ENDIF
@dog:
...
GOTO @ENDIF
@ENDIF:
```

For more information, see The [Eval Macro section](#) of Built-in Custom Probe Variables.

Simple Comparisons in Scripts

InterMapper TCP Scripts can compare two string or integer numeric values and branch based on the results. The commands below are no longer preferred as the EVAL statement described above is equally simple and more powerful.

The SBNE ("String Branch Not Equal") compares the two *string* values and branches if they are not equal. One or both of the arguments can be variables, expressed as `${variable-name}`.

The NBNE ("Numeric Branch Not Equal") and NBGT ("Numeric Branch Greater Than") compares two *numeric* values, branching on the result. The arguments to these commands are strings and are expected to be within quotes. To convert a string to a numeric value, place a number sign (#) before the parameter. For example:

```
STOR "val1" "100"
STOR "val2" "50"
NBGT #${val1} #${val2} @exit
```

In this example, the string `${val1}` will be converted to the numeric value 100, and `${val2}` will be converted to the numeric value 50, and the branch will be taken, because 100 is greater than 50.

Note: The NBGT, NBNE and other TCP probe commands expect integer arguments only (with an optional + or -). A script parses up to the first non-digit character. Thus, the value of "50.5" is 50; the remaining digits are ignored. If you wish to compare against a fraction or floating point value, use the EVAL statement described above.

These commands are described in detail in the [TCP Probe Command Reference](#) topic.

The <script> Section

Use the <script> section of a TCP probe to define a sequence of commands the probe uses to interact with and query a device, and to interpret the responses from the device. The <script> section uses the TCP Probe Scripting Language, (described below) a sequential language with a [rich set of commands](#).

```
<script>
...
</script>
```

TCP Probe Scripting Language

Use the InterMapper TCP Probe Scripting language to create custom probes. You can use script statements to send data to the device being tested, to examine responses from that device, and to return a status based on the response. To view a TCP Probe script example, see [Example TCP Probe File \(Pg 128\)](#).

- [Script Process Flow \(Pg 104\)](#)
- [Script Command Format \(Pg 104\)](#)
- [String Argument Format \(Pg 105\)](#)
- [String Matching \(Pg 106\)](#)
- [Numeric Argument Format \(Pg 107\)](#)
- [Using Labels for Program Control \(Pg 108\)](#)
- [Using Variables \(Pg 109\)](#)
- [Handling Script Failures \(Pg 110\)](#)
- [Adding Comments \(Pg 110\)](#)

Script Process Flow

Each probe has a common process flow. It sends data (as a datagram or over a TCP connection) to the device to be tested, then examines any responses. Based on responses, the probe sets the device status (*UP*, *DOWN*, *CRITICAL*, *ALARM*, *WARN*, *OK*). It also sets a *condition string*, which contains a text description of the state.

Script Command Format

All script command keywords have the following requirements:

- All commands are 4 letters long.
- All commands are case-sensitive.
- All commands MUST be in UPPER CASE.
- There must be white space between a command and each argument. You can include other text (e.g. comments) after the first argument, as long as it is separated by white space from the remaining arguments.

Example:

The **MITCH** command has the format

```
MITCH "string" #fail
```

The command statement

```
MITCH "blah" else goto #7
```

is treated exactly the same as

```
MITCH "blah" #7
```

When parsing the statement, InterMapper ignores the "else goto" part. This allows you to include comments to make the behavior of the script more obvious. This extraneous text does not have to be in uppercase.

String Argument Format

Some commands take string arguments.

- String arguments must be enclosed in double-quotes.

Example:

```
"This is a string"
```

Special Characters

The following special characters may be included by using a backslash escape code:

- \r** Carriage Return
- \n** Unix Linefeed
- \t** Horizontal Tab
- \f** Formfeed
- \b** Backspace
- \v** Vertical Tab
- \a** Alert (bell) Character
- \"** Double Quote
- ** Backslash
- \ooo** Octal Number
- \xhh** Hexadecimal Number

Special Character Example:

```
"\tThis sentence is preceded by a tab, and followed by a carriage  
return and linefeed.\r\n"
```

String Matching

The MTCH and EXPT commands both specify a string to match. When specifying the string, you can use regular expressions. See Wildcard Matching, below.

Controlling Case-Sensitivity

- By default, string-matching is case-sensitive.
- Place an 'i' after the final quote if you want the matching to be case-insensitive.

Examples:

"fred" matches only "fred".

"fred"i matches "fred", "FRED", or "FrEd".

Wild-card Character Matching

In some cases, it is convenient to match a more general pattern. You can use simple regular expressions to match patterns and place them into variables.

To use regular expressions in MTCH and EXPT:

- Place an 'r' after the closing quote of the match string to indicate that the contents of the string is a regular expression.
- Place an 'i' after the closing quote of the match string to indicate that the match is case-insensitive.
- An expression inside round brackets (parentheses) creates a match group and places matched text within a numbered variable. The first variable is \${1}, the second is \${2}, and so on.
- A subsequent MTCH or EXPT command resets the variables, so you should make a copy of the contents into another variable after a match. See the example below.

Simple Example:

"red"r matches "fred", "Fred", "tred", "bred", etc. It does not match "freD" unless you include the "i" after the string.

More Complex Example:

Given the following returned data:

```
"var1=12 var2=1234.00 var3=45"
```

You match and store each data variable into an InterMapper variable:

```
MTCH m"var1=([0-9]+)"i else goto +1 (skip the next STOR line)
STOR "var1" "${1}"
MTCH m"var2=([0-9]+)"i else goto @BLAH
STOR "var2" "${1}"
MTCH m"var3=([0-9]+)"i else goto @BLAH
STOR "var3" "${1}"
```

Note: True Regex groups and the Alternate operator (|) are not supported.

Numeric Argument Format

Some commands take numeric arguments.

- Numeric arguments are formed using a # sign followed by digits.

Example:

```
WAIT #30
```

Using Numeric Arguments with the GOTO Command

In many cases, numeric arguments are used to specify the script statement number to go to when a failure occurs. A special notation allows you to express these jumps as relative offsets.

- Include a sign ('+' or '-') after the # to express a relative offset from the current statement.

Example:

```
GOTO #+2
```

Default Values and Script Termination

- If a command takes a numeric argument, but you do not include it, the default value is 0.
- If you specify 0 as the statement to goto when the script fails, the script is terminated with a *DOWN* condition.

Using Labels for Program Control

Use a label as script marker to which you can jump from elsewhere in the script.

Labels take the form:

```
@label_name
```

- Labels must be alone on a line.

Example:

```
@IDLE
```

Jumping to a Label

Use the *GOTO* command to jump to a label.

Example:

```
WAIT #30 seconds else goto @IDLE
```

Using Relative Offsets to Transfer Control

You can specify an offset for the *GOTO* command

Specify a offset (in statements) "#+n" or "#-n" to jump forward or backward *n* statements (respectively).

Example:

```
MITCH "${WARN Response}" else #+2
```

Using Variables

You can substitute variables in a script statement before the statement is processed.

- Variables names and their default values can be defined in the <parameter> section of the probe file, or by using the STOR, NADD, or TIME command.
- Variable names are preceded by a dollar sign (\$), and are enclosed by curly braces.
- Variable names are case-insensitive.
- See the the [Built-in Variable Reference \(Pg 31\)](#) topic for detailed information on variable usage.

Example:

\${Password} and \${password} are treated as the same variable

Built-in Macros

A macro is an expression that modifies an input string to produce another string. The built-in macros are:

\${_LINE:<line>}	The first <num> characters of the last line received.
\${_BASE64:<param>}	The Base-64 encoding of the string that follows the ":",.
\${_CVSPASS-WORD:<param>}	The value of <param> encoded for use as a password over the CVS pserver protocol.

Handling Script Failures

Certain script commands may fail, either because they are malformed or because an unexpected situation occurs. For example, the script could jump to a non-existent command, it could fail to match a string it expects, or an unexpected disconnection could occur. In each case, the script immediately branches to a failure handler in the script. Each command that can fail takes the statement number of the failure statement as a numeric argument. If this number is omitted, the script will terminate with a "DOWN" status.

Example:

In the following example, the MTCH command succeeds if the incoming line of data contains "220". If the command fails, the script branches to statement 3.

```
MTCH "220" ELSE #3
```

Note: If the script is idle for too long, it may go to a special "idle" handler. See the [WAIT command \(Pg 123\)](#) for more details.

Adding Comments to your Script

There are two ways to add comments to your script:

- Add text between or after arguments to a script command.
- Add a comment using the InterMapper probe file comment format.

Adding text within a command line

You can add text between arguments in a command line, as well as adding text after the line.

Examples:

The following statements all have exactly the same effect:

```
MTCH "331 " #14
MTCH "331 " else #14
MTCH "331 " else goto -1- #14 — Unexpected or unknown response to
USER command
```

Adding text in Comment format

Use the HTML comment syntax to add comments to a probe files. Place comments anywhere in a probe file. HTML comment syntax can be simplified by following this rule:

Begin a comment with "<!--", end it with "-->", and do not use "--" within the comment.

Example:

```
<!-- This text is treated as a comment, and will be ignored -->
```

The <script-output> Section

The <script-output> section of a TCP probe file formats the information retrieved from the device and sends it to the device's Status window. Format the script output using IMML, [InterMapper's Markup language](#).

TCP Probe Command Reference

This is a list of commands defined in the InterMapper TCP Probe Scripting Language. For an example of a custom probe script, see the [Annotated Example of the FTP \(Login\) Script \(Pg 124\)](#).

Device I/O Commands

The following commands operate on a device by sending data to the device or by reading one or more lines from the input (from the connection to the device being tested). Each command that reads a device compares its string to the *current line* - which is the most recently-read line from the connection. If there is no current line (for example, if a SEND command has been executed), these statements will read one or more lines to get the current line.

- [EXPT "string" #fail \(Pg 117\)](#) - Searches incoming lines for the specified *string*.
- [MTCH "string" #fail \(Pg 119\)](#) - Searches the next incoming line for the specified string.
- [SKIP "string" #fail \(Pg 122\)](#) - Ignore all incoming lines containing the specified *string*.
- [DISC #discfail \(Pg 116\)](#) - Jump to a specified line number if the probe is suddenly disconnected.
- [CONN #timeout \["TELNET"\]\["SECURE"\] \(Pg 116\)](#) - Specifies the connect timeout of the probe and whether to process Telnet options.
- [RCON \(Pg 121\)](#) - reconnects to the specified server and port.
- [PORT #port_num #connect_timeout \(Pg 120\)](#) - No longer required (the remote port number is now a separate parameter in the configuration dialog.)
- [LINE \[ON | OFF\] \(Pg 118\)](#) - Specifies whether the script should read incoming data as lines or as raw data.
- [NEXT \(Pg 120\)](#) - Clears the input buffer so that subsequent **MTCH** commands will operate on newly-received information.
- [SEND "string" \(Pg 121\)](#) - Sends the specified string to a remote device.
- [BRVC {BER sequence} \(Pg 115\)](#) - Receive TCP data and decode from BER format into a local format.
- [BSND {BER sequence} \(Pg 115\)](#) - Encode local data in BER format and send. (See LDAP probes for examples, syntax.)

Commands that control script flow

The following commands control the order of operations in the script.

- [CHCK "string" #fail \(Pg 115\)](#) - Determines whether "string" is non-empty.
- [DONE status \["message"\] \(Pg 116\)](#) - Terminates a script with a specified condition.
- [EXIT \(Pg 117\)](#) - Terminates a script with the condition specified previously by **STAT**.
- [FAIL \(Pg 118\)](#) - specifies the line # to jump to if a CONN command fails to connect.
- [GOTO #statement \(Pg 118\)](#) - Branches immediately to the specified statement number.

- [NBGT #arg1 #arg2 #line \(Pg 120\)](#) - (**N**umeric **B**ranch **G**reater **T**han) branches to #line if #arg1 is greater than #arg2.
- [NBNE #arg1 #arg2 #line \(Pg 120\)](#) - (**N**umeric **B**ranch **N**ot **E**qual) Compares two numeric arguments and branches to the indicated #line if they are not equal.
- [SBNE "arg1" "arg2" #line \(Pg 121\)](#) - (**S**tring **B**ranch **N**ot **E**qual) Compares two string arguments and branches to the indicated #line if they are not equal.
- [STAT status \["message"\] \(Pg 122\)](#) - Specifies the status condition of a script when it ends.
- [WAIT #secs #idlefail #discfail \(Pg 123\)](#) - Specifies the number of seconds the probe waits for a response.

String processing commands

The following commands process and manipulate strings.

- [EVAL \\${result} := expression \(Pg 117\)](#) - Assigns the evaluated value of expression to \${result}
- [STOR "variable" "string" \(Pg 122\)](#) - Stores the string into the variable named "variable"
- [SCAT "variable1" "variable2" #fail \(Pg 121\)](#) - Concatenates variable1 and variable2, placing the resulting string in variable1.
- [NADD "variable" #number \(Pg 119\)](#) - (**N**umeric **A**dd) Adds a numeric value to a variable.

Commands that measure time

- [STRT \(Pg 123\)](#) - Starts a millisecond timer that InterMapper can use to determine the elapsed time for some event.
- [TIME "variable" \(Pg 123\)](#) - Sets the named variable to the current number of milliseconds from the most recent STRT command.
- [WAIT #secs #idlefail #discfail \(Pg 123\)](#) - Specifies the number of seconds the probe waits for a response.

Probe Command Details - Alphabetical

BRCV {BER Sequence}

Receive TCP data and decode from BER format into a local format, checking for expected tags and values as indicated. BER stands for "Basic Encoding Rules" for ASN.1. See LDAP probes for examples, syntax.

Documentation of BER format is beyond the scope of this manual.

InterMapper-specific BER syntax information you need is:

- { - starts a sequence (sequences may be nested)
- } - ends a sequence
- [- starts a hexadecimal tag
-] - ends a hexadecimal tag
- # - indicates a literal number follows
- " - begins and ends a literal string

Remember that `${}` is the variable format. Don't confuse these the sequence start and end characters `"{}"` with the variable delimiters.

Example:

```
BRCV { #1, [61]{ [0A]#ENUM, "", "" } } else @PARSE_
ERROR
```

BSND {BER sequence}

Encode local data in BER format and send. BER stands for "Basic Encoding Rules" for ASN.1. See LDAP probes for examples and syntax.

Documentation of BER format is beyond the scope of this manual. See [Inter-Mapper-specific BER syntax \(Pg 115\)](#) above for the information you need to use BSND.

Example:

```
BSND { #1, [60]{ #3, "${Bind Name}", [80]"${Bind Pass-
word*}" } }
```

CHCK "string" #fail

Use the **CHCK** command to determine whether "string" is non-empty. If the string is empty, the script jumps to the specified #fail line.

This command can be used to construct scripts whose control changes depending on whether an optional parameter is supplied.

Possible failures: None

CONN #timeout ["TELNET"]["SECURE"]

Use the **CONN** command to specify the connect timeout of the probe and whether to process Telnet options.

If you are going to use the **CONN** command, it *must be the first statement of the script*. When the script executes, the parameters of the **CONN** statement determine the options InterMapper uses to connect to the remote computer.

#timeout: The number of seconds to wait while trying to connect before giving up.

"TELNET": If the second parameter of the **CONN** command is "TELNET" (including the quotes), then the connection is created in a mode where the TCP stream automatically processes and negatively acknowledge any incoming Telnet options. This allows a Telnet probe to ignore the telnet options and work in simple line-by-line mode for the remainder of the script.

SECURE: To create an SSL connection, place the word SECURE at the end of the line.

SECURE:NO_TLS TLSv1 is turned off when making a secure connection. The HTTPS (SSLv3) probe uses this option.

Possible Failures: None

DISC #discfail

Use the **DISC** command to cause the script to jump to a specified line number if the probe is suddenly disconnected. You can use this command to identify scripts that fail because of a TCP disconnection.

The script's disconnect line can also be set using the third parameter to the WAIT command.

DONE status ["message"]

Use the **DONE** command to terminate the script with the specified condition, which must be one of

[OKAY | WARN | ALRM | DOWN]

The optional "message" parameter lets you provide more detail about the condition. The status values for the **DONE** command must be in *UPPERCASE*.

Example:

```
DONE ALRM "[HTTP] 500 Response received."
```

This example sets the status of the device to *ALRM*. The condition of the device (which is displayed in the device's Status window and the Device List window) is

set to "[HTTP] 500 Response received." to give the user an indication of the reason for the alarm.

Possible Failures: None.

Tip: If the final statement of your script is not a **DONE** command, the script automatically terminates with a "DONE OKAY" status.

EVAL *\${result} := expression*

Assigns a value to the variable in *\${result}* based on *expression*.

The expression can use [any operators or functions defined in Probe Calculations, allowing you to perform variable assignments, arithmetic calculations, relational and logical comparisons, as well as use built-in functions to perform bitwise, rounding and mathematical operations. You can also perform operations on strings using `sprintf` formatting and regular expressions.]

Examples:

```
example1  
example2  
example3  
example4
```

EXIT

Use the **EXIT** command to terminate the script, setting the status and condition string to whatever is specified by a previous **STAT** command.

EXPT "string" #fail

Use the **EXPT** command to "**EXPECT**", or search for the specified *string* in any number of incoming lines.

- If the string is found, the script falls through to the next statement.
- If the string is not found, the script jumps to the statement specified in the *#fail* parameter.

Notes:

EXPT is identical to MTCH, except that:

- MTCH fails if the **next** line or block does not match what is specified.
- EXPT keeps going until it finds a line or text block that matches what is specified.

Both EXPT and MTCH can use regular expressions. For more information, see [String Matching \(Pg 106\)](#).

Example:

```
EXPT "220 " #14
```

Possible Failures:

The EXPT command can fail if the expected text is not received before the connection closes. In that event, the script jumps to the statement specified by *#fail*.

However, if the timeout specified by a previous WAIT command expires before the connection closes, the script jumps to the *#idlefail* line specified by the **WAIT** command instead.

FAIL

Specify the line number to jump to if the probe fails to connect. The FAIL command must follow immediately after a CONN command line.

Possible Failures:

If the statement number is out of bounds, the script terminates with a **DONE** command and *DOWN* status.

GOTO #statement

Use the **GOTO** command to branch immediately to the specified statement number.

Possible Failures:

If the statement number is out of bounds, the script terminates with a **DONE** command and *DOWN* status.

LINE [ON | OFF]

Use the **LINE** command to specify whether the script should read incoming data as lines or as raw data.

Notes:

- By default, the script reads in **LINE ON** mode. That is, the incoming data is read until it is terminated by a CR-LF or just a plain LF, and then the line is processed.
- If you issue a **LINE OFF** command, data is read without regard for line delimiters.

Reading raw data is useful for scanning HTTP data since web pages are not necessarily broken into lines. InterMapper's TCP probe has a maximum line buffer of 4096 characters, so if lines are longer than that, they may be treated as separate lines.

Tip: Once you've matched some data in **LINE ON** mode, you shouldn't match any more because your position in the buffer is not restored and you may miss something.

Possible Failures: None

MTCH "string" #fail

Use the **MTCH** command to "**MaTCH**", or search for the specified string in the next incoming line. If found, the script falls through to the next statement.

Notes:

MTCH is identical to EXPT, except that:

- MTCH fails if the *next* line or block does not match what is specified.
- EXPT keeps going until it finds a line or text block that matches what is specified.

Both EXPT and MTCH can use regular expressions. For more information, see [String Matching \(Pg 106\)](#)

Example:

```
MTCH "331" #16
```

Possible Failures:

If the next incoming line does not contain the desired string, or if the connection closes before the next line can be read, this script fails. In either case, the script jumps to the statement specified by *#fail*.

If the idle timeout expires instead, the script jumps to the *#idlefail* line specified by the previous **WAIT** command.

NADD "variable" #number

The **NADD** (**N**umeric **Add**) command adds a numeric value to a variable. The variable is looked up and converted to a numeric value. The number is added, and the result is converted back to a string and placed into the variable.

Example:

```
NADD "fred" #3
```

adds 3 to the value of the variable `fred`. If `fred` contains "3", the result will be "6". If `fred` contains "golf", the result will be "3" (because the conversion from a string to a number yields zero).

If the number is missing, the script adds zero to the value.

Possible Failures: None.

NBGT #arg1 #arg2 #line

Use the **NBGT** (**N**umeric **B**ranch **G**reater **T**han) command to branch to *#line* if *#arg1* is greater than *#arg2*.

Example:

```
NBGT ${arg1} ${arg2} @exit  
branches to the label @exit if the numeric ${arg1} is greater than  
${arg2}.
```

Note: Use the leading **#** to force InterMapper to treat the arguments as numeric values.

Possible Failures: None.

NBNE #arg1 #arg2 #line

Use the **NBNE** (**N**umeric **B**ranch **N**ot **E**qual) command compares the two numeric arguments, and branches to the indicated *#line* if the arguments are not equal.

Example:

```
NBNE ${arg1} ${arg2} @exit
```

branches to the label `@exit` if the numeric `${arg1}` is not equal to `${arg2}`.

Possible Failures: None.

NEXT

The **NEXT** command clears the input buffer (represented by the `${LINE}` variable) so that subsequent **MTCH** commands will operate on newly-received information.

Notes:

- The **SEND** command incorporates an implicit **NEXT** command.
- The **NEXT** command has no effect if input is not in **LINE** mode.

Possible Failures: None.

PORT #port_num #connect_timeout

Deprecated This command is no longer required in a script because the remote port number is now a separate parameter in the configuration dialog.

If present, this command must be in the first statement of the script. The first parameter specifies the default TCP port to connect to on the remote computer. The `#connect_timeout` parameter is the number of seconds to wait for the probe to connect.

Possible Failures: None

RCON

Takes no parameters.

See the Barracuda probes for examples and syntax.

Possible Failures: None

SBNE "arg1" "arg2" #line

The **SBNE** (**S**tring **B**ranch **N**ot **E**qual) command compares the two string arguments, and branches to the indicated #line if the arguments are not equal.

Example:

```
SBNE "${arg1}" "${arg2}" @exit
branches to the label @exit if the string ${arg1} is not equal to
${arg2}.
```

Possible Failures: None.

SCAT "variable1" "variable2" #fail

The **SCAT** (**S**tring **C**on**C**atenate) command concatenates variable1 and variable2 and places the resulting string in variable1.

Example:

```
STOR "name" "Fred"
sets the variable ${name} to the string "Fred"

SCAT "name" "Flintstone" @TOO_LONG
sets the variable ${name} to the value "FredFlintstone"
```

Possible Failures: If the sum of the lengths of the strings exceeds 255, the SCAT command will fail, and transfer to the @TOO_LONG label.

SEND "string"

Use the **SEND** command to send the specified string to the remote device.

To send a line of data, you must explicitly specify the CR-LF using the quoting convention.

Example:

```
SEND "Greetings!\r\n"
transmits the data "Greetings!" followed by a CR-LF.
```

Possible Failures: This command can't fail. If the data can't be sent because of a network failure or device failure, the failure shows up in a subsequent **EXPT** or **MTCH** command.

SKIP "string" #fail

Use the **SKIP** command to ignore all incoming lines containing the specified *string*. The script falls through to the next statement when an incoming line does *not* contain the *string*.

Possible Failures:

If the connection closes unexpectedly, the script jumps to *#fail*.

If the **WAIT** timeout (as defined by the **WAIT** command) expires, the script jumps to *#idlefail*.

STAT status ["message"]

Use the **STAT** command to specify the status of the device when the script ends. This command *does not* terminate the script. You can also specify a condition string as the second argument.

The status must be one of

[OKAY | WARN | ALRM | DOWN | CRIT]

Example:

```
STAT ALRM "[HTTP] 500 Response received."
```

Note: A subsequent **STAT** or **DONE** command overrides the value set by this command.

STOR "variable" "string"

The **STOR** command stores the string into the variable named *variable*. If you wish to set a variable to a numeric value, enclose the number in quotes ("). Subsequent parts of the script can refer to this variable as `${variable}`.

Examples:

```
STOR "fred" "foobar"
```

sets the variable `fred` to the text string `foobar`. Subsequent parts of the script can refer to this variable as `${fred}`.

```
STOR "fred" "3"
```

sets the variable `fred` to the string value `"3"`.

Note: String variables can be any length up to 65,535 characters.

Possible Failures: None.

STRT

The ***STRT*** command starts a millisecond timer that InterMapper can use to determine the elapsed time for some event. See the **TIME** command.

Example:

```
STRT Starts the timer.
```

Possible Failures: None.

TIME "variable"

The ***TIME*** command sets the named variable to the current number of milliseconds from the most recent **STRT** command.

Example:

```
TIME "connecttime"
sets the variable connecttime to the number of milliseconds since the
most recent STRT command. If there was no previous STRT com-
mand, the variable will be set to zero.
```

Possible Failures: None.

WAIT #secs #idlefail #discfail

Use the ***WAIT*** command to specify the number of seconds the probe waits for a response.

Parameter 1 - #secs: The number of seconds to wait for a response. If you do not include a ***WAIT*** command in your script, the default timeout 60 seconds is used.

Parameter 2 - #idlefail: If present, the script jumps to this line number if the probe is idle for the specified number of seconds. This idle handler supercedes the error line number specified by the ***EXPT***, ***SKIP***, or ***MTCH*** commands. If the ***#idlefail*** parameter is not included, the script branches to the failure handler of the current command. [link]

Parameter 3 - #discfail: If present, the script jumps to this line if the probe is unexpectedly disconnected. This allows you to identify scripts that fail because of a TCP disconnection.

Possible Failures: None

Tip: You should specify all three parameters in the ***WAIT*** command.

Annotated Example of the FTP (Login) Script

```
01) PORT #21 (default tcp port)
02) WAIT #30 seconds
03) EXPT "220 " else goto -1- #14
04) SEND "USER ${User ID}\r\n"
05) MITCH "331" else goto -2- #16
06) SEND "PASS ${Password}\r\n"
07) MITCH "230" else goto -3- #20
08) SEND "NOOP\r\n"
09) MITCH "200" else goto -4- #24
10) SEND "QUIT\r\n"
11) EXPT "221" #+1 (i.e. can't fail)
12) DONE OKAY
13)
14) DONE DOWN "[FTP] Unexpected greeting from port ${_REMOTEPORT}.
${_LINE:50})" -1-
15)
16) MITCH "500" else goto #+2 -2-
17) DONE ALRM "[FTP] Port ${_REMOTEPORT} did not recognize the 'USER'
command."
18) DONE ALRM "[FTP] Unexpected response to USER command. (${_
LINE:50})"
19)
20) MITCH "530" else goto #+2 -3-
21) DONE WARN "[FTP] Incorrect login for \"${User ID}\"."
22) DONE ALRM "[FTP] Unexpected response to PASS command. (${_
LINE:50})"
23)
24) DONE ALRM "[FTP] Unexpected response to NOOP command. (${_
LINE:50})"
```

Explanation of the Script

```
01) PORT #21 (default tcp port)
02) WAIT #30 seconds
```

Line 1: The PORT command at the beginning of the script specifies the default TCP port number for FTP, port 21.

Line 2: The WAIT command specifies that if the script doesn't hear responses back within 30 seconds, it fails.

```
03) EXPT "220 " else goto -1- #14
```

Line 3: FTP servers normally send one or more "220" lines to greet new FTP control connections. Our script scans the incoming lines for "220 ".

Note the space following the 220; we don't want to match an incoming "220-"; the incoming dash indicates there are still more 220 lines to be read -- we only want to match the final 220 line.

If the script fails to find "220 " before the connection closes or within 30 seconds, the script branches to statement 14. The "-1-" is an arbitrary label used to make the destination of the branch more easily visible.

The string "else goto -1-" has no function (except readability) in the script command text; this statement could have been written equally well as EXPT "220 " #14 . Note that statement #14 also has comment of "-1-" to show it is the destination.

```
04) SEND "USER ${User ID}\r\n"
```

Line 4: Send the FTP USER command. With this command, we send the user ID specified by the user, e.g. "anonymous". Note that you must include the carriage-return and line-feed at the end of the string sent, to denote the line ending.

```
05) MTCH "331" else goto -2- #16
```

Line 5: The script looks for the 331 response to the USER command.

If something else arrives, the script jumps to statement 16. Unlike the **EXPT** command, the **MTCH** command fails immediately if the next line doesn't contain the required text.

[...] (Skipping down to statement 16).

```
16) MTCH "500" else goto #+2 -2-  
17) DONE ALRM "[FTP] Port ${_REMOTEPORT} did not recognize the  
  \"USER\" command."  
18) DONE ALRM "[FTP] Unexpected response to USER command. (${_  
  LINE:50})"
```

Line 16: Statement 16 is executed only if statement 5 fails; that is, if an unexpected response to the USER command is received. The response is checked to see if it matches "500", which would indicate that the command isn't supported. This is possible if you accidentally try to pass the USER command to a TCP service other than FTP.

If the server's response matches "500", the script is terminated with the device in the *ALARM* status (in statement 17). The message reports that the server did not recognize the USER command.

If the server's response does not match "500", the script skips two lines to statement 18. This statement terminates the script with the *ALARM* status and uses the `${LINE}` macro to include the first 50 characters of the response line in the message.

Measuring TCP Response Times

You can measure the response time of a device as it is being tested by a TCP probe. Times are measured in milliseconds.

With TCP Probes, InterMapper measures both the *time to establish the connection* and the *time for various portions of an interaction*. These times can be charted and logged.

Time Measurement Probe Variables

TCP Timers are:

Connection initiation interval	<code>\${_connect}</code>	Records the time required to establish a connection.
Connection duration interval	<code>\${_active}</code>	Records the duration from the connection request until the end of the end of the script.

TCP Script Commands

InterMapper supports two commands for measuring intervals during a script. These are:

STRT Starts the probe's custom timer.

TIME var-name Sets the variable named `${varname}` to the milliseconds elapsed since the customtimer was started.

The <script-output> Section

Use the optional <script-output> section to display the results of custom TCP probes. The data in this section appears in a Status window when you click and hold on the device. The format of this section is the same as the <snmp-device-display>, described in [Customized Status Windows \(Pg 27\)](#).

Use the `${_connect}` and `${_active}` variables, as well as any variables set with the TIME *varname* command, in the <script-output> section of the Status window.

A Note on Accuracy

InterMapper uses different techniques to measure the round-trip times of various probes.

- **Pings (ICMP and AppleTalk echoes)** - These are the most accurate timings. InterMapper detects the arrival of the Ping response at the moment it arrives, and thus, it can compute the response times with millisecond accuracy.
- **Other UDP-based and TCP-based probes** - These timings are computed by InterMapper as it does its normal polling. Thus, the measured time can be affected slightly by such things as the number of devices probed and other various other tasks, as they may affect how long it takes InterMapper to execute a single round of polling.

Example TCP Probe File

The following is the Dartware-provided probe for the Custom TCP script.

```
<!--
Custom TCP (com.dartware.tcp.custom)
Copyright © 2000-2003 Dartware, LLC.
Please feel free to use this as the basis for new probes.
-->

<header>
  type = "tcp-script"
  package = "com.dartware"
  probe_name = "tcp.custom"
  human_name = "Custom TCP" version = "1.2"
  address_type = "IP"
  port_number = "23"
</header>

<description>
\GB\Custom TCP Probe\P\
This probe lets you send your own string over the TCP connection and
set the
status of the device depending on the response received. There are
six
parameters which control the operation of this probe:
\i\String to send\p\ is the initial string sent over the TCP con-
nection. This
could be a command which indicates what to test, or a combination of
a command
and a password. The string is sent on its own line, terminated by a
CR-LF.

\i\Seconds to wait\p\ is the number of seconds to wait for a
response. If no
response is received within the specified number of seconds, the
device's status
is set to DOWN.

\i\OK Response\p\ is the substring which should match the device's
"ok
response". If it matches the first line received, the device is
reported to have
a status of OK.

\i\WARN Response\p\ is the substring which should match the device's
warning
response.

\i\ALRM Response\p\ is the substring which should match the device's
alarm
response.

\i\DOWN Response\p\ is the substring which should match the device's
down
response.
```



```

If InterMapper cannot connect to the specified TCP port, the device's
status is
set to DOWN.
</description>

```

```

<parameters>
  "String to send" = ""
  "Seconds to wait" = "30"
  "OK Response" = ""
  "WARN Response" = ""
  "ALRM Response" = ""
  "DOWN Response" = ""
</parameters>

```

```

<script>
  CONN #60 (connect timeout in secs)
  SEND "${String to send}\r\n"
  WAIT #${Seconds to wait} else goto @IDLE
  EXPT ".r else goto @DISCONNECT
  MITCH "${OK Response}" else #+2
  DONE OKAY "[Custom] Response was \"${_LINE:50}\"."
  MITCH "${WARN Response}" else #+2
  DONE WARN "[Custom] Response was \"${_LINE:50}\"."
  MITCH "${ALRM Response}" else #+2
  DONE ALRM "[Custom] Response was \"${_LINE:50}\"."
  MITCH "${DOWN Response}" else #+2
  DONE DOWN "[Custom] Response was \"${_LINE:50}\"."

  @IDLE:
    DONE DOWN "[Custom] Did not receive a line of data within ${S-
seconds to wait}
seconds. [Line ${_IDLELINE}]"

  @DISCONNECT:
    DONE DOWN "[Custom] Connection disconnected before a full line
was received."
</script>

```

```

<script-output>
  \B5\Custom TCP Information\0P\
  \4\Time to establish connection:\0\ ${_connect} msec
  \4\Time spent connected to host:\0\ ${_active} msec
</script-output>

```

Chapter 5

Command Line Probes

```
type="cmd-line"
```

InterMapper provides the ability to run a *command-line* probe, a script or program (written in perl, C, C++, or your favorite language.) Your program's return value becomes the device's status on the InterMapper map.

Common Sections of a Command-Line Probe

Each command-line probe follows the same general format as other probe files, sharing these common sections:

- The [<header>](#) (Pg 13) section of a command-line probe specifies the probe type, name, and a number of other properties fundamental to the operation of the probe.
- The [<description>](#) (Pg 18) section specifies the help text that appears in the Probe Configuration window. Format the description using IMML, [InterMapper's Markup language](#).
- The [<parameters>](#) (Pg 19) section defines the fields presented to the user in the Probe Configuration window.

Sections Specific to Command-line Probes

Each command-line probe also has:

- The [<command-line>](#) (Pg 134) section - defines the command-line, specifying the path to the executable, the command to execute, and any arguments to the command.
- The [<command-exit>](#) (Pg 136) section - controls how the device's state is set, based on the results of the command.
- The [<command-display>](#) (Pg 142) section - controls what appears in the device's Status window.

InterMapper uses the information in the probe's `<command-line>` section to invoke the program or script and pass arguments to it. InterMapper sets the device's status based on the return code from the program or script. In addition, any data written to the script's standard output file is used as the device's reason string, and appears in the status window. The total amount of data that can be returned by the program, including return code, reason string, and additional values, is 64k.

InterMapper's command-line probes are similar to [Nagios® plugins](#). You can see the [standard set of Nagios plugins](#). Many vendors and individuals have created their own Nagios plugins. You will have to download the Nagios plugins and build/compile them yourself.

If you wish to develop your own command-line probes, we recommend you follow the [developer guidelines for Nagios](#). This will result in probes/plugins that work for both InterMapper and Nagios.

For more information about InterMapper and Nagios Plugins, see the [Nagios Plugins page \(Pg 147\)](#).

See [Command Line Probe Example \(Pg 143\)](#) for a sample shell script and corresponding probe.

The `<tool>` section - embedding a companion script

You can also embed script code directly in a probe. This provides an easy way to deliver a command-line probe and a script that it runs in a single probe file, ensuring that the version of the script matches the version of the probe. WMI probes provide a number of good examples of companion scripts. For more information, see [The `<tool>` Section \(Pg 137\)](#).

Command Line Script API

When InterMapper invokes a command line program or script, it passes parameters on the command line. Use the `path`, `cmd`, and `arg` properties of the `<command-line>` section to specify the script or other executable to invoke, and any arguments to the command. As the script developer, you are responsible for parsing the arguments.

The script can return three kinds of information to InterMapper:

1. The operating system return code, or *exit code*, is used to indicate the success/failure/severity. This will be handled by the `<command-exit>` section of the probe file.
2. The script can optionally return *additional values*, such as measurements, discovered during execution. It does this by writing to the script's *stdout*. You return these values as a comma-separated list enclosed in "`\{`" ... "`}`" characters. These values can then be handled as variables in the probe's `<command-display>` section. The values themselves are *name-value* pairs in the form:

```
<name> := <value>
```

3. The script can also return a **reason string** that will be used to explain the device's condition. You specify the reason string by writing to the script's *stdout*. This text should follow the closing "`}`" of any additional values.

Example: The following output from a script sets two values to the probe:

`$rtt` and `$hop`, and sets the device's reason string to *"Round-trip time is very high"*

```
\{ $rtt := 5, $hop := 2 } Round-trip time is very high
```

You can do a significant amount when writing to stdout, using the `${^stdout}` variable. For more information, see [The `\${^stdout}` variable and the Reason string \(Pg 38\)](#).

Installing a Command-line Probe

Once you have created your probe, you need to install it before you can test it.

To install and use a command-line probe:

1. If you are using an external script or other executable, create the program, and make it runnable. If it's a Perl, Python, or other script, set the permissions so that it can run from the command line. If it's written in C, C++, or other non-interpreted language, compile the source and then place the resulting binary in an appropriate directory. (See the [path \(Pg 130\)](#) discussion below.)
Note: If you embed a script in the `<tool>` section of the probe, permissions are set by InterMapper when it writes the script to the Tools directory (when you import or reload the probe.)
2. Create a Command-line probe that references the executable program or contains the script in the `<tool>` section.
3. Import or reload the probe (from the Set Probe window) to make it available.

See [Command Line Probe Example \(Pg 143\)](#) for a sample shell script and corresponding probe.

Passing parameters to a command-line probe

Pass arguments to the command line into the probe by accessing the parameter variables with `${parametername}`. The named arguments can be added to the command line.

For example, use `${Timeout}` for an parameter as follows:

```
<parameters>  
  Timeout = "7"  
</parameters>
```

The `arg` variable could be set as follows:

```
arg = "-H ${Timeout}"
```

Note: Depending on the nature of the parameters you are passing, you may want to pass the parameters through STDIN, as described below.

Sending Data to STDIN

Using the `ps` command on Unix systems, or using the Task Manager or other utility programs on Windows systems, it is possible to see the command line arguments. This represents a security vulnerability. Use the `input` property of the `<command-line>` section to pass sensitive data to STDIN, removing this vulnerability. See [Sending Data to STDIN](#) for a detailed example.

The <command-line> Section

The <command-line> section allows you to specify the information needed to execute the commands for the probe. There are three variables:

- **path** - specify the path to the executable script/command.
- **cmd** - specify the actual script/command.
- **arg** - specify the arguments to be passed to the script/command.
- **input** - specify information to pass to STDIN to the script/command.

The Properties of the <command-line> Section

- Use the `path` property to specify the directories in which InterMapper should look for the executable to run as a probe. This is the only path InterMapper will use; the `PATH` environment variable is not used. The `path` property follows the conventions for the `PATH` environment variable on the system hosting InterMapper. The example below is for Unix or Mac OS X. A path for Windows would use "\" instead of "/" and ";" instead of ":".

Notes:

- If no path is specified, *InterMapper Settings/Tools* is used as the path.
- On Unix systems, it might be possible to see the command line arguments in the 'ps' listing. This represents a security vulnerability. Use the `input` variable to pass values to stdin, removing this vulnerability. For more information, see [Sending Data to STDIN](#), below.
- Use the `cmd` property to specify the executable you wish to run. In the example below, this is "check_ping". Note that you need to specify the exact name, including any extensions such as .exe or .cmd. You may also specify arguments as part of the `cmd` property if you'd like.
- Use the `arg` property to specify arguments to the executable. This may be instead of or in addition to specifying them in the `cmd` property. We could have just as easily written our sample `cmd` property as a command and argument, like this:

```
<command-line>
  path = ""
  cmd  = "check_ping"
  arg  = "-H ${ADDRESS} -w 100,10% -c 1000,90%"
</command-line>
```

- Use the `input` property to pass information to STDIN. See [Sending Data to STDIN](#), below.

Note the use of the "\${ADDRESS}" macro. This is replaced with the address given when the device was created. You can also use the "\${PORT}" macro to indicate the port given when the device was created.

Sending Data to STDIN

Using the `ps` command on Unix systems, or using the Task Manager or other utility programs on Windows systems, it is possible to see the command line arguments. This represents a security vulnerability. Use the `input` variable to pass sensitive data to `stdin`, removing this vulnerability.

This mechanism provides a less visible channel for sensitive communication to a probe script. Usernames, passwords, and SSL pass-phrases are likely candidates for this technique.

Example:

```
<command-line>  
  cmd = "executable"  
  input = "${User} ${Password}"  
</command-line>
```

The <command-exit> Section

The <command-exit> Section

The <command-exit> section allows you to specify which results from the command indicate the five InterMapper device states. The states are:

- down
- critical
- alarm
- warning
- okay

For each state, you need to indicate what item InterMapper should examine and what its value should be to result in that state being set. At the moment, the only thing InterMapper can look at is the exit code, which is indicated with `${EXIT_CODE}`. So, in the example below, the line:

```
down: ${EXIT_CODE} = 2
```

means, "To determine if the device is down, examine the exit code from the command; if it is 2, the device is down." If none of the criteria for the states you have defined are true, then the device is set to "unknown".

The <tool> Section

Use the <tool> section of a command-line probe to embed a script's code directly into a command-line probe. The <tool> section provides a convenient way to maintain the probe and the script in a single file.

```
<tool:scriptname>
  [script code]
</tool:scriptname>
```

Replace ***scriptname*** with the executable you want to use for the script.

Replace ***[script code]*** with the code of the script.

What happens when you load a probe with a <tool> section?

When the InterMapper server starts or when you import or reload a probe, if a tool section appears for a probe, InterMapper creates a subdirectory of Tools with the canonical name of the probe and writes the script to that subdirectory, using `scriptname` as a file name.

If a subdirectory of that name already exists, all non-hidden files are deleted before the script is written out. For this reason, you should not edit scripts directly in the subdirectories of the Tools directory, since they are overwritten when probes are reloaded.

For example, given the trivial example of a command-line probe where the canonical name is `com.dartware.cmdline.test`, where the `cmd` clause in the <command-line> section is:

```
cmd="python test.py"
```

or, using the `${PYTHON}` macro:

```
cmd="${PYTHON} test.py"
```

and the tool section is:

```
<tool:test.py>
  # Trivial example
  print "okay"
  raise SystemExit, 0
</tool:test.py>
```

When InterMapper starts or reloads probes, a subdirectory of Tools named `com.dartware.cmdline.test` is created if it doesn't exist, and (in this case) a file

named "test.py" is written into it, containing the text between <tool:test.py> and </tool:test.py>.

The WMI probes provide a number of good examples of this feature.

Calling external scripts and other executables

While using the <tool> section is recommended, it is optional. You can call an external script or other executable by providing the correct path to it in the `cmd` property of the <command-line> section of the probe. If you provide a paths to multiple directories in the `path` parameter, InterMapper looks in the specified directories for the executable. The <tool> section is appropriate only for scripts, not for compiled programs.

Python Example

```
<!--
check connect
(com.dartware.commandline.check connect.txt)
Copyright (c) 2009 Dartware, LLC. All rights reserved.
-->

<header>
  type = "cmd-line"
  package = "com.dartware"
  probe_name = "commandline.check_connect"
  human_name = "Check Connect"
  version = "1.1"
  address_type = "IP"
  display_name = "Miscellaneous/Test/Check Connect"
</header>

<description>
  \GB\Check for connect\p\

  This probe checks to see if you can connect to the given address
  and port.
</description>

<parameters>
  "CHECK_PORT" = "80"
</parameters>

<command-line>
  cmd=${PYTHON}
  arg="check connect.py ${ADDRESS} ${CHECK_PORT}"
</command-line>

<command-data>
  -- Currently unused.
</command-data>

<command-exit>
  -- These are the exit codes used by Nagios plugins
  down: ${EXIT_CODE}=4
  critical: ${EXIT_CODE}=3
```

```
    alarm: ${EXIT_CODE}=2
    warn:  ${EXIT_CODE}=1
    okay:  ${EXIT_CODE}=0
</command-exit>

<command-display>
</command-display>

<tool:check_connect.py>
    import sys
    import socket

    # constant return codes for InterMapper
    OKAY = 0
    WARNING = 1
    ALARM = 2
    CRITICAL = 3
    DOWN = 4

    retcode = OKAY
    output = ""

    try:
        host = sys.argv[1] # The remote host
        port = long(sys.argv[2]) # The port
    except:
        print "Usage: check_connect HOST PORT"
        sys.exit(DOWN)

    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        s.close()
    except IOError, e:
        retcode = DOWN
        if hasattr(e, 'reason'):
            reason = 'Reason: ' + e.reason
        elif hasattr(e, 'code'):
            reason = str(e.code)
        else:
            reason = "unknown"

    output = "Error (" + reason + ") connecting to " + str(host) + ":"
    + str(port)

    print output
    sys.exit(retcode)
</tool:check_connect.py>
```

Cscript Example

```

<!--
Check Web
Copyright (c) 2009 Dartware, LLC. All rights reserved.
-->

<header>
  type = "cmd-line"
  package = "com.dartware"
  probe_name = "commandline.check_web"
  human_name = "Check Web"
  version = "1.1"
  address_type = "IP"
  display_name = "Miscellaneous/Test/Check Web"
  visible_in = "Windows"
</header>

<description>
  \GB\Check Web\p\

  Given an address or hostname, attempts to connect to a web server.
</description>

<parameters>
</parameters>

<command-line>
  -- Empty path forces the InterMapper Settings:Tools directory
  path=
  cmd="{CSCRIPT} check_web.vbs"
  arg="{address}"
  timeout = ${Timeout (sec)}
</command-line>

<command-data>
  -- Currently unused.
</command-data>

<command-exit>
  down:${EXIT CODE}=4
  critical:${EXIT CODE}=3
  alarm:${EXIT CODE}=2
  warning: ${EXIT CODE} = 1
  okay:${EXIT CODE}=0
</command-exit>

<command-display>
  ${^stdout}
</command-display>

<tool:check_web.vbs>
  Dim web
  Set web = Nothing
  Set web = CreateObject("WinHttp.WinHttpRequest.5.1")

  numargs = wscript.arguments.count

```

```
If (numargs < 1) Then
wscript.Echo "Usage: check_web hostname"
wscript.quit(4)
End If

URL = "http://" + wscript.arguments(0)

on error resume next
web.Open "GET", URL, False
on error resume next
web.Send
If err.Number <> 0 Then
returncode = 4
Else
If err.Number = 0 and web.Status = "200" Then
returncode = 0
Else
returncode = 4
End If
End If

If returncode <> 0 Then
wscript.Echo "Error connecting to " + URL + "."
Else
wscript.Echo ""
End If
wscript.quit(returncode)
</tool:check_web.vbs>
```

The <command-display> Section

The <command-display> section displays variables in the device's Status window using the same form as the output section of other probe types. If the plugin returns a non-integer value, you should use the `${chartable:...}` macro to display digits to the right of the decimal point. As with other probe types, you format the appearance of the output using IMML, [InterMapper's markup language](#).

See [Command Line Probe Example \(Pg 143\)](#) for a sample shell script and corresponding probe.

Command Line Probe Example

The following shell script is called from the command-line probe:

```
#!/bin/sh
# Expects an address passed in. Passes out the address and a pretend
result.
# Note that we use "\$" instead of just "$" because "$" has special
meaning
# in a shell script.
echo "\{ \${addr} := \"\$1\", \${result} :=1.2345 } Note that everything
after the brace
is used as the reason."
```

```
<!--
Simple Command Line Example (com.dartware.cmd.simple)
Copyright (c) 2007 Dartware, LLC. All rights reserved.
-->

<header>
  type = "cmd-line"
  package = "com.dartware"
  probe name = "cmd.simple"
  human name = "Simple Command Line Output Example"
  version = "1.0"
  address_type = "IP"
</header>

<description>
This probe shows how to use the specially-formatted output from the
simple shell script listed above for display in the command-display
section, rather than being set to the reason as is usual for
command-line probes.
</description>

<parameters>
</parameters>

<command-line>
  path = ""
  cmd = "simple.sh ${ADDRESS}"
</command-line>

<command-exit>
  down: ${EXIT CODE} = 2
  alarm: ${EXIT CODE} = 1
  okay: ${EXIT CODE} = 0
</command-exit>

<command-display>
  \B5\Simple Probe Information\0P\
  Output from $addr is $result (${chartable: #.#### : $result})
</command-display>
```

For more information about Nagios, visit the web site at <http://www.nagios.org>.
Nagios® and the Nagios logo are registered trademarks of Ethan Galstad.

InterMapper Python Plugins

InterMapper DataCenter ships with an embedded Python interpreter. You may use this interpreter to write command-line probe scripts and command-line notifiers. This Python interpreter is a good way to give maximum compatibility across systems. In InterMapper DataCenter 5.4.2, the version of Python we ship is 2.6, with optimized system libraries.

An extensive introductory tutorial on Python is available at <http://docs.python.org/tutorial>

As shipped, this Python interpreter requires the use of optimized and stripped mode (-OO), so the interpreter must be invoked as:

MacOSX/ Linux/ Unix:	<code>/usr/local/imdc/core/python/bin/imdc -OO [script_name]</code>
Windows:	<code>c:\Program Files\I- nterMapper\dwf\core\python\imdc.exe -OO [script_name]</code>

Notes:

- Use the `${PYTHON}` macro as shown below; it automatically determines the platform and expands to the proper path to the interpreter with the `-OO` argument.
- Use the [The <tool> Section \(Pg 137\)](#) (<tools:sample.py> in the example below) to incorporate the Python script directly into the probe file itself.

Simple example

A simple sample probe that includes a Python script might look like this. The script automatically gets saved in the InterMapper Settings/Tools directory.

```
<!--
  Command Line Python Sample (com.dartware.python.sample.txt)
  Custom Probe for InterMapper (http://www.intermapper.com)
  Please feel free to use it as a base for further development.
  Original version 31 Mar 2004 by Christopher L. Sweeney, Dartware,
  LLC.
  Updated 13 Jun 2007 by Stephen P. Ryan, Dartware, LLC, for Python
  Updated 28 Dec 2007 to update text descriptions and
    include display name header line -reb
  Updated 3 Jan 2010 to include ${PYTHON} macro -reb
-->

<header>
  type="cmd-line"
  package="com.dartware"
  probe_name="python.sample"
```

```

    human_name="Python Sample"
    version="1.1"
    address_type="IP"
    display_name = "Miscellaneous/Test/Python Sample"
</header>

<description>
\GB\Python Sample Command-Line Probe\p\

A sample command line probe which executes a Python script.

The Python script generates and returns a random number which sets
the device status to one of four values Down/Alarm/Warning/OK.

</description>

<parameters>
</parameters>

<command-line>
    path=""
    cmd="${PYTHON} sample.py ${ADDRESS}"
    arg=""
</command-line>

<command-exit>
    down:${EXIT_CODE}=3
    alarm:${EXIT_CODE}=2
    warning:${EXIT_CODE}=1
    okay:${EXIT_CODE}=0
</command-exit>

<command-display>
</command-display>

<tool:sample.py>
#!/usr/local/imdc/core/python/bin/imdc -OO
# Sample Python script uses InterMapper's Python interpreter

import sys

if (len(sys.argv) < 2):
    print "Usage: %s _address_" % sys.argv[0]
    sys.exit(0)
addr = sys.argv[1]

# Code to get status from device at address addr
import random
result = random.randrange(4)

print "Pretending we got result %d from device at address %s" %
(result, addr)
sys.exit(result)

</tool:sample.py>

```

Nagios Plugins

InterMapper's command-line probes are similar to **Nagios® plugins** (<http://www.nagios.org>). You can see the [standard set of Nagios plugins](#). Many vendors and individuals have created their own Nagios plugins, many of which are available in the [development section](#). You will have to download the Nagios plugins and build/compile them yourself.

The Nagios Plugin probe lets you specify a Nagios plugin to run, along with any associated parameters. You can use the `${ADDRESS}` and `${PORT}` macros in the command line--InterMapper substitutes the device's IP address and the specified port. InterMapper will invoke the plugin and use the exit value to set the condition of the device to UP/Okay, UP/Alarm, UP/Critical, or DOWN.

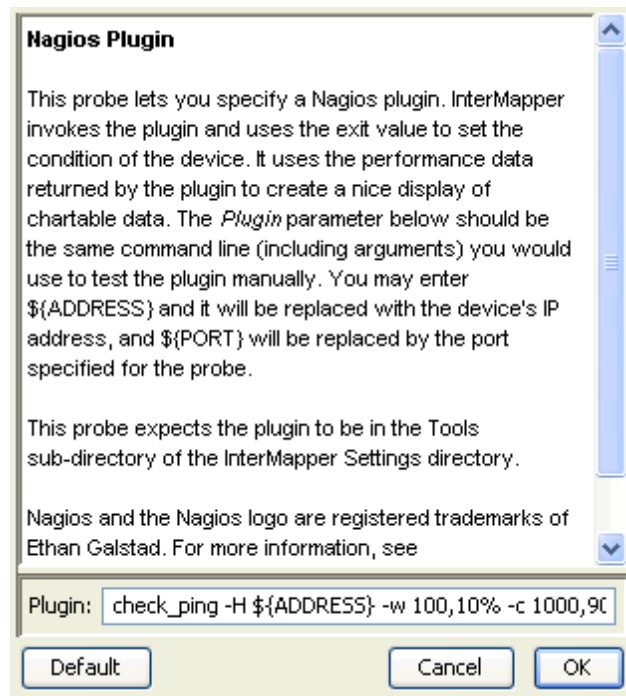
InterMapper also interprets the information written by the plugin to stdout and puts it in the InterMapper status window, nicely displaying and making chartable the performance data returned by the probe, and displaying the reason/condition provided.

The Nagios Plugin probe expects the Nagios plugin to be in the Tools sub-directory of the InterMapper Settings directory.

Nagios and the Nagios logo are registered trademarks of Ethan Galstad. For more information, see <http://www.nagios.org/>.

To install and use a Nagios plugin:

1. Download the plugin. Make it executable by following the instructions from the creator.
2. Move the executable file (or a link/alias/shortcut to it) to the *Tools* sub-directory of the *InterMapper Settings* directory.
3. Add a device to the map and set its Probe Type to **Nagios Plugin**.
4. Enter in the plugin file's name and arguments in the Plugin field of the configuration window.
5. You can use the `${ADDRESS}` and `${PORT}` macros in the



command line. InterMapper will substitute the device's IP address and the specified port.

Creating Nagios Probes

If you wish to develop your own Nagios plugins, you should follow the developer guidelines for Nagios (found at <http://nagiosplug.sourceforge.net/developer-guidelines.html>). This will result in probes/plugins that work for both InterMapper and Nagios.

As described in the Nagios Guidelines, a Nagios plugin returns:

- **a POSIX return code** as described in [section 2.4](#) of the Guidelines. InterMapper uses this to determine the device's state.
 - 0 = OK;
 - 1 = Warning (yellow);
 - 2 = Critical (red);
 - 3 = Down.
- **A single output line on STDOUT** with the following format.

```
<description of the device status>|Perfdata
```

where:

- <description of the device status> is a short text string. This becomes the InterMapper Condition string, and is described in [section 2.1](#) of the Guidelines. The output string should have the format:

```
SERVICE STATUS: information text
```

- | is the "pipe" character to separate the description from the "Perfdata"
- Perfdata (Performance Data) is a series of name/value pairs. These are described in [section 2.6](#) of the Guidelines, but are generally a space-separated list with this form:

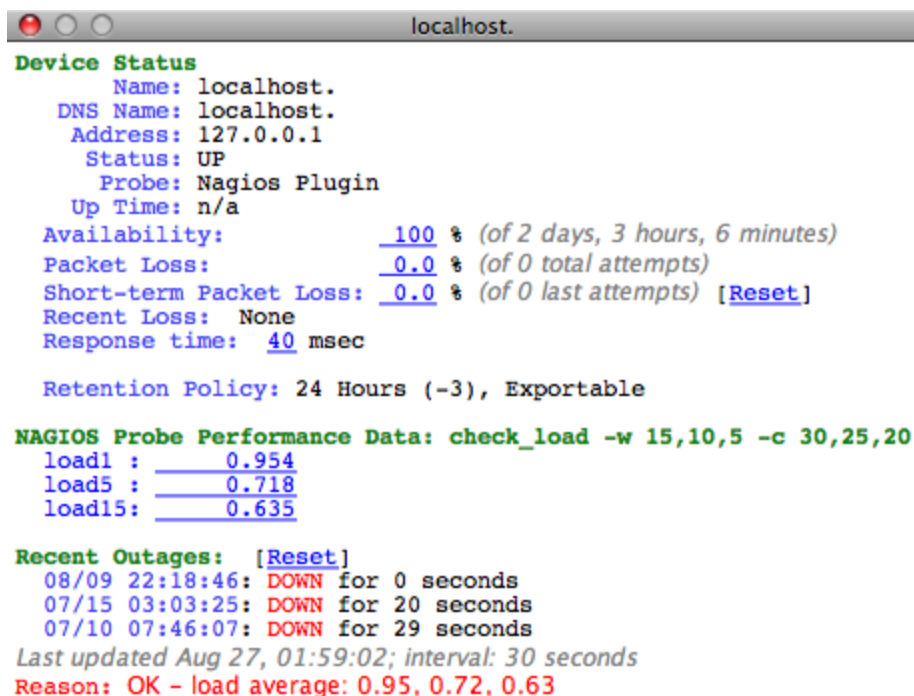
```
<code>
'label'=value[UOM];[warn];[crit];[min];[max]
</code>
```

Example Return String

The Nagios check_load string returns three values: the load average over 1, 5 and 15 minutes. When the plugin is invoked, it returns a response like this:

```
% ./check_load -w 15,10,5 -c 30,25,20
OK - load average: 0.95, 0.72, 0.64|load1=0.954;15.000;30.000;0;
load5=0.718;10.000;25.000;0; load15=0.635;5.000;20.000;0;
```

InterMapper parses the plugin's response line and uses the `${nagios_output}` macro to produce a status window as shown in the image below.



```
localhost.  
Device Status  
  Name: localhost.  
  DNS Name: localhost.  
  Address: 127.0.0.1  
  Status: UP  
  Probe: Nagios Plugin  
  Up Time: n/a  
  Availability: 100 % (of 2 days, 3 hours, 6 minutes)  
  Packet Loss: 0.0 % (of 0 total attempts)  
  Short-term Packet Loss: 0.0 % (of 0 last attempts) [Reset]  
  Recent Loss: None  
  Response time: 40 msec  
  
Retention Policy: 24 Hours (-3), Exportable  
  
NAGIOS Probe Performance Data: check_load -w 15,10,5 -c 30,25,20  
load1 : 0.954  
load5 : 0.718  
load15: 0.635  
  
Recent Outages: [Reset]  
08/09 22:18:46: DOWN for 0 seconds  
07/15 03:03:25: DOWN for 20 seconds  
07/10 07:46:07: DOWN for 29 seconds  
Last updated Aug 27, 01:59:02; interval: 30 seconds  
Reason: OK - load average: 0.95, 0.72, 0.63
```

For more information about Nagios, visit the web site at <http://www.nagios.org>. Nagios® and the Nagios logo are registered trademarks of Ethan Galstad.

Changes for InterMapper 5.0

For those familiar with the older Nagios Template probe, the new Nagios Plugin probe contains the following changes in behavior:

- The Nagios Template probe mapped plugin exit code 2 as down. The Nagios Plugin probe maps plugin exit code 2 as critical, and plugin exit code of 3 as down.
- The Nagios Template probe took anything written to stdout as the "condition" or "reason" for the status. The Nagios Plugin probe detects the presence of performance data ('PERFDATA') ([section 2.6](#) of the Guidelines) in the output, and makes a formatted and chartable display of the data.
- We have not changed the canonical name of the Nagios probe; any device which used the old Nagios Template probe will now automatically use the Nagios Plugin probe. An InterMapper probe will automatically handle a Nagios plugin if it has the following:
 - "flags" = "NAGIOS3" in the `<header>` section of the probe. See [Probe File Header](#).
 - `${nagios_output}` in the `<command-display>` section of the probe. See [Built-in Variables](#)

Nagios Plugin Example

```

<!--
Command Line Nagios Plug-in Example (com.dartware.nagiosx.template)
Copyright (c) 2008 Dartware, LLC. All rights reserved.
-->

<header>
  type      = "cmd-line"
  package   = "com.dartware"
  probe_name = "nagios.template"
  human_name = "Nagios Plugin"
  version    = "1.6"
  address_type = "IP"
  display_name = "Miscellaneous/Nagios/Nagios Plugin"
  flags      = "NAGIOS3"
</header>

<description>
\GB\Nagios Plugin\p\

This probe lets you specify a Nagios plugin. InterMapper invokes the
plugin and uses the exit value to set the condition of the device. It
uses performance data returned by the plugin to create a nice display
of chartable data. The \i\Plugin\p\ parameter below should be the
same command line (including arguments) used to test the plugin manually.
You may enter \${ADDRESS} and it is replaced with the device's
IP address, and \${PORT} is replaced by the port specified for the
probe.

This probe looks in the Tools sub-directory of the InterMapper
Settings directory for the plugin.

Nagios and the Nagios logo are registered trademarks of Ethan Gal-
stad. For more information, see \U2\http://www.nagios.org\PO\
</description>

<parameters>
  Plugin = "check_ping -H \${ADDRESS} -w 100,10% -c 1000,90%"
</parameters>

<command-line>
  -- Empty path forces the InterMapper Settings:Tools directory
  path = ""
  cmd = \${Plugin}
</command-line>

<command-exit>
  -- These are the exit codes used by Nagios plugins
  down: \${EXIT_CODE}=3
  critical: \${EXIT_CODE}=2
  alarm: \${EXIT_CODE}=1
  okay: \${EXIT_CODE}=0
</command-exit>

<command-display>
\B5\NAGIOS Probe Performance Data: \${Plugin}\PO\

```

```
{nagios output}  
</command-display>
```

NOAA Weather Probe Example

It is now easier than ever to build command-line probes. Here is a fun example that retrieves temperature data from the [US NOAA weather feed](#) in a particular city.

How does this probe work?

1. Right-click a device and choose **Select Probe...**
2. Select the **Weather Service-Temp** probe from the Miscellaneous/Test category.
3. Enter the city code for the closest weather station (KLEB is at Lebanon Municipal Airport, about a half mile to the south of us.) The Status window shows the name of the weather station, with a chartable value for the temperature reading.

Under the covers, InterMapper launches a Python program to contact the weather service, retrieve the meteorological conditions for the indicated city, and parses out the XML response to retrieve the temperature. (There's lots more information in the Weather Service feed - the program could easily be extended to display more information.) Here are some of the features of this probe:

- The `${PYTHON}` macro gives the path to the built-in python interpreter of InterMapper DataCenter no matter what platform you're using. For example, the probe can now use

```
cmd = "${PYTHON} program.py"
```

and InterMapper substitutes the proper path to invoke Python, whether on Windows, OSX, or Linux/Unix.

Note: To use this macro, the InterMapper DataCenter (IMDC) must be installed. IMDC will be installed automatically with InterMapper 5.2 on Windows and OSX; Linux and Unix systems require a separate install for IMDC.

- You can include the script directly in the text of the probe file. This makes it much easier to write scripts and keep the probe file in sync. To do this, use the `<tool:program-name>` section in your probe file. The example below contains a program named `noaa-weather.py`. When InterMapper loads the probe, it parses out this section and saves it in a folder within the Tools directory of InterMapper Settings. Programs in the `<tools>` section may also save private files in that directory.
- The example probe file uses a couple interesting Python libraries. First is `url-lib2` that makes it easy to make queries from web services. It's a few straightforward calls to build a url, issue it, and retrieve the results.
- The probe also uses the `xml.dom.minidom` library to parse out XML data returned from the NOAA web service. This library is particularly well-explained in Chapter 9 of *Dive into Python*.

The NOAA Temperature Probe

To use this probe, copy the text below, paste it to a text editor, save it to a text file, then use File->Import->Probe... in InterMapper.

```
<!--
Weather Service Temperature - Retrieve the temperature from the NOAA
weather XML (com.dartware.tool.noaa.txt)
Copyright (c) 2009 Dartware, LLC.
Please feel free to use this as a base for further development.
-->

<header>
  type      = "cmd-line"
  package   = "com.dartware"
  probe_name = "tool.noaa"
  human_name = "Weather Service-Temperature"
  version   = "1.2"
  address_type = "IP"
  display_name = "Miscellaneous/Test/Weather Service-Temp"
</header>

<description>
\GB\Retrieve the current temperature\p\

This probe retrieves the current temperature from the NOAA weather
feed. To see the proper city code, visit:

\u4=http://www.weather.gov/xml/current_obs/\http://www.-
weather.gov/xml/current_obs/\p0\
</description>

<parameters>
  "Weather Station"    = "KLEB"
</parameters>

<command-line>
  path=""
  cmd="{PYTHON} noaa-weather.py"
  arg="{Weather Station}"
</command-line>

<command-exit>
-- These are the exit codes used by Nagios plugins
  down:  ${EXIT_CODE}=4
  critical:  ${EXIT_CODE}=3
  alarm:  ${EXIT_CODE}=2
  warn:  ${EXIT_CODE}=1
  okay:  ${EXIT_CODE}=0
</command-exit>

<command-display>
```

```

\b5\ Temperature for $loc\p0\
  Temperature: $temp \3g\degrees F\p0\
</command-display>

<tool:noaa-weather.py>

# noaa-weather.py
# Scan the XML results from NOAA's XML feeds
# e.g., http://www.weather.gov/xml/current_obs/KLEB.xml
# for relevant weather-related information.
# 25 Mar 2009 -reb
import os
import re
import sys
import getopt
import urllib
import urllib2
import htmllib
from xml.dom import minidom

# httplib.HTTPConnection.debuglevel = 1 # force debugging....

# options are: station

try:
    opts, args = getopt.getopt(sys.argv[1:], "")
except getopt.GetoptError, err:
    searchString = "getopt error %d" % (err)

station = args[0]
userAgent = "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_5; en-us)
AppleWebKit/525.18 (KHTML, like Gecko) Version/3.1.2 Safari/525.20.1"
noaaString = "http://www.weather.gov/xml/current_obs/%s.xml"
noaaString = noaaString % (urllib.quote_plus(station))

# print noaaString;
retcode = 4;
try:
    request = urllib2.Request(noaaString)
    opener = urllib2.build_opener()
    request.add_header('User-Agent', userAgent)
    usock= opener.open(request)
# print buf
except IOError, e:
    if hasattr(e, 'reason'):
        resp = 'We failed to reach a server. '
        reason = 'Reason: ' + 'Wrong host name?' # e.reason[1]
    elif hasattr(e, 'code'):
        resp = 'The server couldn\'t fulfill the request. '
        reason = 'Error code: ' + str(e.code)
    print "{ $temp := '%s', $loc := 'Unknown' } %s" % (0, resp + reason)
    sys.exit(retcode) # make it look down

retcode = 0 # looks like it'll succeed
xmldoc = minidom.parse(usock)
tempList = xmldoc.getElementsByTagName('temp_f')

```

```
tempElem = tempList[0]
tempval = tempElem.firstChild.data
loclist = xmldoc.getElementsByTagName('location')
locval = loclist[0].firstChild.data
print "{ $temp := '%s', $loc := '%s' }%s" % (tempval, locval, temp-
val + ' degrees at ' + locval)
sys.exit(retcode)
</tool:noaa-weather.py>
```

See also

[\\${PYTHON} macro \(Pg 31\)](#) - the full path the to the Python interpreter.

[The <tool> Section \(Pg 137\)](#) - Include a script directly into the probe file.

Python Documentation:

urllib2 - <http://docs.python.org/library/urllib2.html>

xml.dom.minidom - <http://docs.python.org/library/xml.dom.minidom.html>

Dive into Python: A very readable chapter on XML processing in Python

http://diveintopython.org/xml_processing/

Installing and Reloading Probes

Use custom probes to enhance InterMapper's capabilities. These are probes created for special purposes or for certain devices.

To install a custom probe:

1. Download the probe, and uncompress the file if necessary.
2. From the File menu, choose **Import > Probe** command or

Click the **Plus** icon (shown at right) in the Set Probe window. A file dialog appears. Choose the probe file you want to import, and click **Open**. The probe is installed copied to the InterMapper Settings/Probes directory, and becomes available from the Set Probe window.



To test a custom probe after importing it:

1. Open an InterMapper map.
2. Add a new device with the DNS name or IP address of the device you want to test.
3. Right-click the device and choose **Set Probe...** The Select Probe window appears.
4. Select the new probe from the Select Probe window.
5. Configure the probe by filling in the fields as required.
6. When finished, click **OK**. InterMapper begins using the new probe to test the device.

Reloading a Probe

If you make changes to a probe, you need to do one of two things before the changes become active:

- Re-import the probe as described above. You can do this regardless of the probe file's location.
- Manually reload probes. If you make a change to the probe file located in the InterMapper Settings/Probes directory, you must click the **Reload Probes** button (shown at right) to activate your changes.



Modifying Built-in Probes

Built-in probes are stored in a ZIP archive named `BuiltinProbes.zip`, located in the `Intermapper Settings/Probes` directory.

To view or modify a built-in probe, you'll need to unzip the archive.

How InterMapper resolves filename conflicts

InterMapper scans the archive as well as the unzipped contents of the folder.

If a built-in probe's filename matches an unzipped version, InterMapper locates the most recent version of the probe with:

- the probe's **version number**
- then
- the probe's **last-modified date**

If you are developing or modifying a built-in probe, be sure to advance the version number to be sure that InterMapper uses the modified version.

Sharing Probes

InterMapper has an enthusiastic user-base that has contributed literally hundreds of probes, many of which have been adopted as built-in probes. Users are encouraged to check out the library of user-contributed probes, and to contribute useful probes themselves.

Sharing Your Own Probes

If you create a probe you find useful, please contribute your new probe so others can use it as well. Send us an e-mail at support@dartware.com and we'll post it to the Contributions page mentioned above.

Using Contributed Probes

You can use any of the probes that Dartware has created, or use probes contributed by our other customers. These probes are available from:

<http://dartware.com/go.php?to=probes.contrib>

Troubleshooting Probes

There are a number of different ways to troubleshoot your custom probes.

The most basic troubleshooting is done through the error messages that appear in the device's Status window. Use the [comprehensive list of Error Messages](#) to help you track down errors.

For SNMP probes, you can use [SNMPWalk](#) to view the MIB variables returned from an SNMP device. Use [SNMPWalk with the -O option](#) to redirect the output from the Debug log to an SQLite database.

You can also gain a lot of information by [measuring response times](#) of a device as it is being tested. A number of different timers are available for viewing and charting.

Chapter 6

Errors with Custom Probes

When working with custom probes, you may see results that you don't expect. Here are some common problems.

Error: "Undefined variable" in Debug log

When processing a probe, InterMapper will not evaluate an expression if it detects a variable that is not defined. A variable will be undefined if it is not in the symbol table. This could happen because:

- there is a typo in the variable name
- the value for the variable was not returned in a SNMP response
- the value was not set earlier in the probe processing

In this case, InterMapper will emit the following message in the Debug log file:

```
Calculation error in rule (probe: com.dartware.example, expression:
"$oid 0"): Undefined variable: '$oid'
```

To guard against these error messages (where it may be a legitimate case that a particular variable is undefined) you may use the "defined()" function in the expression:

```
warning: defined("oid") && ($oid > 0) "Warning condition string"
```

Error: A device shows a "Reason: No SNMP Response." at the bottom of the status window.

There are several reasons that InterMapper might not be able to retrieve SNMP information from a device. The two most common are

- The device doesn't speak SNMP
- You haven't entered the proper SNMP read-only community string.

About SNMP, located in the Troubleshooting section of the [User Guide](#), lists many other reasons.

Error: When I build a custom probe, the status window shows "[N/A]" for certain values.

This probably means that there is an error with the OID for one of the device variables.

Open the Debug window, and look for entries in this format:

```
12:57:00 router.example.net.: SNMP error status [[query = 28]] noSuch-
Name (2), index = 3
1) 1.3.6.1.2.1.1.3: NULL
2) 1.3.6.1.2.1.1.1: NULL
```



```
3) 1.3.6.1.7.1.1.4: NULL  
4) 1.3.6.1.2.1.1.6: NULL
```

Note that the first line above shows a "noSuchName" error for index 3. Look at the subsequent lines to find item 3, and check that OID very carefully. In this example, the proper OID should have a "2" in place of the "7" that's there.

Error: When I build a custom probe, the status window shows "[noSuchName]" for certain values.

This probably means that there is an error with the OID for one of the device variables.

Open the Debug window, and look for entries in this format.

```
13:17:59 OID Error: GetNextRequest from 192.168.1.1 expected  
1.3.6.1.2.1.2.2.1.2.10; got 1.3.6.1.2.1.2.2.1.3.1
```

In this case, the desired value is from a non-existent table row. (The OID 1.3.6.1.2.1.2.2.1.2 is the ifDescr for an interface on a device. The index (.10) indicates which row to retrieve. But when InterMapper requested that row, it learned it was not present.) Consequently, InterMapper displays the "noSuchName" value.

Debugging with the SNMPWalk Command

InterMapper provides a simple SNMPWalk command, available from the Monitor menu, that allows you to perform an SNMPWalk on a specified OID. In some cases this may not be sufficient. You can also execute SNMPWalk as a server command, and include specific arguments as described below.

The InterMapper server implements a simple snmpwalk facility in its debug mode.

```
snmpwalk -v [1|2c|3] -c community -o filename [-e] [-n num-OIDs] -p
161 -r 3 -t 10 IP-address startOID
```

where:

- *-v [1|2c|3]* is the version of SNMP to use: SNMPv1, SNMPv2c, or SNMPv3
- *-c community* indicates the SNMP read-only community string (see note for SNMPv3)
- *-e* if present, means to proceed to the end of the MIB
- *-n num-OIDs* if present, indicates the number of OIDs to display (*-e* and *-n* are mutually exclusive)
- *-o filename* is the name of a SQLite-format file saved in the InterMapper Settings/Temporary directory. For more information see [Using the SNMPWALK -O Option](#).
- *-p* destination port (default is 161)
- *-r* number of retries that InterMapper will attempt if a response doesn't return (default is 3)
- *-t* timeout in seconds that InterMapper waits for a response (default is 10 seconds)
- *IP-address* is the IP address of the device to query
- *startOID* if present as the final argument, indicates the first OID to request

The command will start an SNMP walk on device with the specified IP-Address, starting from the given startOID. The walk will end when the specified number of OIDs has been received. The walk will also end if the OID received from the device does not have the specified start OID as its prefix unless *-e* is specified. If *-e* is specified, the walk will continue until the end of the MIB or the specified maximum OIDs have been received.

Note: For SNMPv3, *community* should be in the following format:

```
username: [md5|sha|none] :authpassword: [des|none] :privpassword
```

Examples

Example: SNMP walk of the ifTable of a device with IP address 192.168.1.1 using SNMPv2c with community string public:

```
snmpwalk -v 2c -c public 192.168.1.1 1.3.6.1.2.1.2.2
```

Example: SNMP walk of the ifXTable of a device with IP address 10.10.2.20 using SNMPv3 with user name 'user', authentication protocol MD5, authentication password 'auth', privacy protocol DES and privacy password 'priv':

```
snmpwalk -v 3 -c user:md5:auth:des:priv 10.10.2.20 1.3.6.1.2.1.31.1.1
```

Example: SNMP walk of the ifTable of a device with IP address 192.168.1.2 using SNMPv3 with user name 'test', authentication protocol MD5, authentication password 'pass', and no privacy protocol:

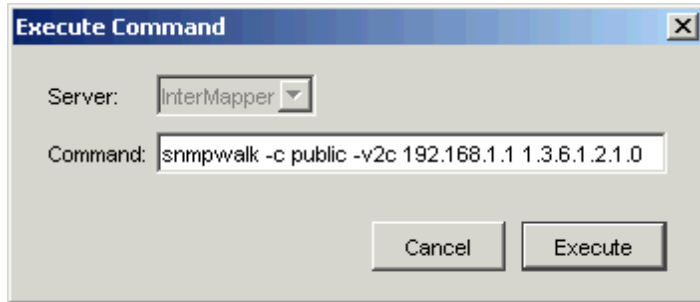
```
snmpwalk -v 3 -c test:md5:pass:none: 192.168.1.2 1.3.6.1.2.1.2.2
```

Example: Walk starting from the ifTable until the end of the device is reached or until 10,000 OIDs have been received:

```
snmpwalk -v 1 -c public -e -n10000 192.168.1.1 1.3.6.1.2.1.2.2
```

Invoking the snmpwalk command

You execute the **snmpwalk** command as a "Server command..." (available from the Help menu's Diagnostics menu) To use this command:



1. Select **Help > Diagnostics > Server Command...** The Server command window appears as shown above.
2. Enter the snmpwalk command, and click Send.
3. The output of the SNMPwalk is written to the **Debug** file, which is at this path: InterMapper Settings : InterMapper Logs : Debugyyyyymmddhhmm.txt

Note: You can use snmpwalk's -o option to direct the output of snmpwalk to an SQLite database. For more information, see [Using the SNMPWALK -o Option \(Pg 165\)](#).

4. The output of the SNMPwalk will also appear in the Debug window, as shown below:

```
SNMPWalk 192.168.1.1: prefix 1.3 (maximum number of OIDs: 2000)
— 9/16/2005 13:04:56
SNMPWalk on 192.168.1.1 started
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.1.0 = OctetString: ExampleOS
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.2.0 = OID: 1.3.6.1.4.1.9.1
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.3.0 = TimeTicks: 11058776
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.4.0 = OctetString: sup-
port@example.com
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.5.0 = OctetString: Example.com
Router
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.6.0 = OctetString: http://ww-
w.example.com
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.7.0 = Integer: 72
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.8.0 = TimeTicks: 413
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.9.1.2.1 = OID: 1.3.6.1.2.1.1.9.1
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.9.1.3.1 = OctetString: See
RFC2580
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.1.9.1.4.1 = TimeTicks: 413
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.2.1.0 = Integer: 2
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.2.2.1.1.1 = Integer: 1
SNMPWalk 192.168.1.1: 1.3.6.1.2.1.2.2.1.1.2 = Integer: 2
...
SNMPWalk 192.168.1.1: Finished (end of MIB reached) — 9/16/2005
13:09:48
```

snmpwalk stopall command

To stop all SNMPwalks for a particular server, you can enter this command in the Server command... window.

```
snmpwalk stopall
```

Error Conditions

InterMapper detects the following error conditions:

- When InterMapper walks to the end of the MIB, it displays a "Finished (end of MIB reached)" message
- When InterMapper fails to receive a response after the specified number of retries, it displays a "Finished (No response received)" message
- The snmpwalk expects that the OIDs received are increasing. When InterMapper receives an OID that is out of order, it would terminate the walk with an error message that indicates that a loop is detected in the walk.

Help from the telnet command line

There is also documentation in the InterMapper's telnet help. Typing 'help snmpwalk' in the telnet window will display a summary of the command.

Using the SNMPWALK -o Option

Instead of writing the voluminous results of the SNMPWalk to the Debug log, InterMapper can write the results to the SQLite file. To use this feature, you use the **-o** option.

When you use the **-o** option, snmpwalk stores its output into an SQLite database. To use this feature, you specify the name of the database file following the **-o** option.

Example:

To create an SQLite3 database named "foo" and store the snmpwalk results there, use the following "server command":

```
snmpwalk -v1 -c public -o foo switch 1.3
```

This writes the output to an SQLite database file named "foo", located in InterMapper Settings/Temporary directory. The database file named "foo" may contain multiple snmpwalk's. The file is created if it doesn't exist.

When the **-o** option is used, only these four lines are written to the Debug log:

```
SNMPWalk command received: 'snmpwalk -v1 -c public -o foo -n 10  
switch 1.3'  
SNMPWalk 192.168.1.36 3: prefix 1.3 (version SNMPv1 ...  
SNMPWalk on switch started  
SNMPWalk 192.168.1.36 3: Finished (10 OIDs ...
```

The SNMPWALK Schema

Here is the schema used for the snmpwalk database:

```
CREATE TABLE walks (
  id          INTEGER PRIMARY KEY,
  address     TEXT,
  port       INTEGER,
  startOid    TEXT,
  snmpVersion INTEGER,
  pktTimeout  INTEGER,
  pktRetries  INTEGER,
  maxOids     INTEGER,
  toEnd       INTEGER,
  timeStarted INTEGER,
  timeFinished INTEGER,
  oidCount    INTEGER,
  stopReason  TEXT
);

CREATE TABLE results (
  walk_id    INTEGER,
  name       TEXT,
  oid        TEXT,
  type       INTEGER,
  value      BLOB
);
```

Table: walks

The walks table stores one row for each snmpwalk command. Each walk will receive a unique id that is used to identify it (id). The other columns are:

address

the address of the snmpwalk target device

port

the udp port number of the snmpwalk target device

startOid

the starting OID specified in the snmpwalk command

snmpVersion

the SNMP version specified

pktTimeout

the packet timeout specified

pktRetries

the packet retry count specified

maxOids

the maximum number of OID's specified

toEnd

a boolean flag indicating that the walk should proceed to the end (i.e. don't limit by startOid).

timeStarted

the UTC timestamp when the walk was started

timeFinished

the UTC timestamp when the walk was completed

oidCount

the number of OID rows walked

stopReason

text indicating the reason the walk stopped where it did

Table: results

The results table stores a row for each entry of one snmpwalk, as specified by id in the walks table.

walk_id

the id of the walk (references walks.id)

oid

the text of the OID naming the SNMP variable

type

the ASN.1 type integer for the SNMP variable

value

the actual, uninterpreted binary value returned by the SNMP agent

Accessing the Data

On Mac OS X 10.4, you can use the built-in `sqlite3` command to access the data in the SQLite database file:

```
$ sqlite3 foo

sqlite> .mode csv
sqlite> select * from walks;
1,"192.168.1.1",161,1.0,0,10000,3,2000,0,1178801645,1178801685,0,"No
answer received"
2,"-
192-
.168.1.2",161,1.3,0,10000,3,2000,0,1178801708,1178801895,2000,"Finished
(2000 OIDs found)"
sqlite> select count(*) from results where walk_id = 1;
0
sqlite> select count(*) from results where walk_id = 2;
2000
sqlite> select * from results where walk_id = 2 order by oid limit
5;
2,"1.3.6.1.2.1.1.1.0",4,"HP J4813A ProCurve Switch 2524..."
2,"1.3.6.1.2.1.1.2.0",6,"+\006\001\004\001\013\002\003\007\013\023"
2,"1.3.6.1.2.1.1.3.0",67,"\035\330J\375"
2,"1.3.6.1.2.1.1.4.0",4,"Bill Fisher"
2,"1.3.6.1.2.1.1.5.0",4,"HP ProCurve Switch 2524"
```

There is a Firefox add-on called "SQLite Manager" that opens and displays SQLite database files. This makes it a cross-platform tool, requiring only Firefox 3.5 or greater, plus a small download.

To install SQLite Manager:

1. Start Firefox 3.5 or newer - Tools -> Add-ons - Click Get Add-ons
2. Enter "SQLite" into the Search field and press Return.
3. Double-click the "SQLite Manager" item to install it. Restart Firefox when instructed.

To use SQLite Manager:

1. Tools -> SQLite Manager to open it.
2. Use the Database -> Connect Database (within the window) to open a saved SQLite database
3. Click the results table to view it.

Chapter 7

InterMapper HTTP API

InterMapper provides an HTTP API for retrieving data from, and sending data to the InterMapper server ("exporting" and "importing", respectively). The API allows an external program to use standard HTTP commands (GET, POST) and a straight-forward URL syntax to make these requests.

These are the major features of the HTTP API:

- **File Import/Export** gives access to files of the InterMapper Settings folder.
- **Table Import/Export** gives access to the tables in the same formats that are currently available through the RemoteAccess Import and Export commands.
- **Acknowledgements** - send Basic Acknowledgements to an InterMapper server using HTTP. This allows you to acknowledge down devices from phones, browsers, or scripts.

Through these API interfaces you can accomplish a number of scripting tasks. Two scripts are provided, allowing you to "clone" the InterMapper Settings directory through a script.

The features of the HTTP API are described in these topics:

[Importing & Exporting Files \(Pg 170\)](#) - How to access the InterMapper Settings directory's file system.

[Importing & Exporting Tables \(Pg 173\)](#) - How to import or export map data directly.

[Acknowledging Devices \(Pg 175\)](#) - How to acknowledge down devices or interfaces using the HTTP API.

[Scripting Examples \(Pg 176\)](#) - Examples of the "clone" scripts and much more.

Importing & Exporting Files

Most of the files in the InterMapper_Settings folder can be accessed through the HTTP API. These include:

- Custom Icons
- Fonts
- Maps
- MIB Files
- Probes
- Sounds
- Web Pages

These folder contents may be exported, but not imported:

- Extensions
- Certificates folder
- Tools

The contents of the following folders are not currently available:

- Chart Data folder
- InterMapper Logs folder
- Deleted and Disabled Maps folders

All other files are served up as binary files with a MIME type of application/octet-stream. Each file has a corresponding URL to retrieve its contents; each folder in InterMapper_Settings also has a URL to retrieve the list of URLs for the files in that folder.

All URLs given below are relative to a URL composed of the InterMapper Server address and the webport, as defined in the server settings. For example, in the following discussion a URL of `/~files` would imply the full URL (either `http:` or `https:`):

```
http://imserver_address:webport/~files
```

Example: The URL above produces a text listing the URLs for the folders in the InterMapper_Settings that can be accessed over HTTP; this is provided for the convenience of scripts that might want to access all files.

A request for following URLs provides a text listing of the URLs for the files within the corresponding folder in the InterMapper_Settings folder.

InterMapper_Settings folder	Corresponding URL
Custom Icons	/~files/icons
Extensions	/~files/extensions
Fonts	/~files/fonts
Maps	/~files/maps
MIB Files	/~files/mibs
Probes	/~files/probes
Sounds	/~files/sounds
Tools	/~files/tools
Web Pages	/~files/webpages

HTTP file imports

To import these files over HTTP, issue a POST request to the appropriate URL with the file contents as a payload; the MIME type should be application/octet-stream.

Icons

You can import icons via HTTP as described above. The URL should take the following form:

```
http://imserver:port/~files/icons/Folder/Filename.type
```

If the file type is a valid image file (jpeg or png), it will be available to use immediately.

Sample curl commands (command-line) to use this facility might look like this (should be all on one line):

```
HTTP: curl --data-binary "@sample.png" http://-  
/localhost:8080/~files/icons/Default/sample.png  
HTTPS: curl -k --data-binary "@sample.png"  
https://localhost/~files/icons/Default/sample.png
```

Note: the `-k` option for HTTPS ignores an unsigned certificate.

Maps

You can import maps or map data using the HTTP API.

A sample curl command line to import a map file should take this form:

```
$ curl --user admin:Pa55w0rd --data-binary @/path/to/local/map_file  
http://imserver:port/~files/maps/map_file
```

Importing & Exporting Tables

Table-based Import/Export

The table-based functions of the InterMapper HTTP API match the capabilities of the Import -> Data file... and Export -> Data file... commands of InterMapper RemoteAccess (available from the File menu). These import and export a number of "tables" of information about the devices being monitored. These tables include:

- Devices
- Interfaces
- Vertices
- Maps
- Notifiers
- Users
- Schema

These tables have detailed descriptions in the *Advanced Data Import/Export* section of the [InterMapper User Guide](#). The URLs for importing and exporting have the following format:

```
http://imserver:port/~export/tablename.format? (options)
```

Supported formats are:

- .tab - save as a tab-delimited text file
- .csv - save as a comma-delimited text file
- .xml - save as an XML format file
- .html- display as HTML directly in the browser

The primary option is "fields=...." The list of valid fields are listed in the "schema" export. For example, this query:

```
http://imserver:port/~export/schema.html
```

provides a list of the supported tables and the fields for each table in an HTML format that you can view right in the browser. Other examples include:

```
http://imserver:port/~export/devices.tab
```

provides a list of all devices on active maps as a tab-delimited file. This URL:

```
http://imserver:port/~export/devices.tab?fields=id,name,macaddress,address
```

provides a list of all devices on active maps, but only includes the ID, Name, MAC-Address and Address fields.

Importing Table-based Data

An external program can also import table information with an HTTP POST operation by including the table data as the payload.

```
http://imserver:port/~import/filename
```

The *filename* in this URL is written to the log file, but is otherwise ignored. It is not used to determine the data to import, nor is it used to specify where the data goes. InterMapper examines the directive line of the attached file to determine what information is imported from the file. It follows the same logic that is used when importing data using the Import->Data File... command available from InterMapper RemoteAccess's File menu.

A sample curl command line to import map data should take this form:

```
$ curl --user admin:Pa55w0rd --data-binary @/path/to/import/file  
http://imserver:port/~import/file
```

Acknowledging with HTTP

You can perform a Basic Acknowledgement of a device by issuing a POST to a URL of this format:

```
http://-
/imserver:port/mapid/-
device/deviceIMID/*acknowledge.cgi?message=URL+encoded+string
```

where:

- *mapid* = the mapid of the corresponding map.
- *deviceIMID* = the IMID of the device you wish to acknowledge.
- *message*: requires a URL-encoded text string

There are several ways of finding these values:

- Look in the web interface at the status window for a device, remove the trailing "!device.html" at the end, and replace that with "*acknowledge.cgi?message=....".
- Review the device table (you can view it using the HTTP API) and get the MapId and IMID.

Example: This curl command sends the POST with the proper string to acknowledge the device:

```
curl --user admin:Pa55w0rd http://-
/imserver:port/mapid/-
device/deviceIMID/*acknowledge.cgi?message=URL+encoded+text+string -d
"dummy post data"
```

Example: This curl command retrieves the full list of devices and each device's address, MapID and IMID

```
curl --user admin:Pa55w0rd http://-
/imserver:port/~export/devices.tab?fields=MapId,IMID,address,name
```

Example: You can also use this expression in Python to create the URL to POST:

```
"http://imserver:port/%s/device/%s/*acknowledge.cgi?message=%s" %
(mapId, IMID,urllib.urlencode([('message', messageStr)]))
```

HTTP API Scripting Examples

Two command line scripts are available; these scripts provide good examples of the use of the InterMapper HTTP API:

- **Unix shell script** - for Unix/Linux platforms
- **Windows vbscript** - for Windows platforms

Both scripts make a copy of the InterMapper Settings directory. Both scripts assume they are invoked from the destination directory rather than trying to look it up - this is intended to make testing easier. Dartware recommends that you create a dummy InterMapper_Settings folder someplace and try the synchronization to that (the destination directory doesn't even require that InterMapper is installed, only the script). The remote server does not need to be the same platform as the destination machine.

The destination directory should **not** have an active InterMapper running in it, as the script assumes that it can replace files at will.

Unix shell script

The Unix shell script assumes that the current working directory is the "Temporary" folder inside the destination InterMapper_Settings (*This is likely to change.*).

The script requires bash, curl, tr, grep, sed and awk. to be installed.

Unix script options

```
clone im.sh [options]
-r [remote host name ]
-t [remote_port]
-u [remote_user ]
-p [remote_password]

Defaults:
remote_host_name = "localhost"
remote_port = 8080 (this is the InterMapper web access port)
remote_user = "admin"
remote_password = "admin"

Example:
clone_im.sh -r nitro.dartware.com -t 8080 -u IMuser -p UsErpaSS
```


Windows vbscript

The Windows vbscript assumes that the current working directory is the destination InterMapper_Settings.

Windows script options

```
clone im.vbs
/host:[remote_hostname]
/port:[remote_port]
/user:[remote_user]
/password:[remote_password]

Defaults:
remote_host_name (none, must be specified)
remote_port = 80
user (none, uses auto-login unless specified)
password (none, uses auto-login unless specified)
secure = false

Example:
clone im.vbs /host:nitro.dartware.com /port:8080 /user:IMuser /password:UsErpaSS /secure:true
```

Known bugs

- The Unix version does not yet support a "secure" switch.
- The Unix version should not have default username and password, to use the auto-login if that's available.
- The Unix version should require the current directory to be the top-level of the destination InterMapper_Settings directory, not the "Temporary" sub-directory.

clone_im.sh

InterMapper_Settings folder	Corresponding URL
Custom Icons	/~files/icons
Extensions	/~files/extensions
Fonts	/~files/fonts
Maps	/~files/maps
MIB Files	/~files/mibs
Probes	/~files/probes
Sounds	/~files/sounds
Tools	/~files/tools
Web Pages	/~files/webpages

```
#!/bin/bash

# Synchronize InterMapper_Settings folder from a remote host with
# InterMapper SDK
#
# Requires curl and gnu awk

remote=localhost
port=8080
user=admin
password=admin
auth=
while getopts 'r:t:u:p:' OPTION ; do
    case $OPTION in
        r) remote="$OPTARG"
           ;;
        t) port="$OPTARG"
           ;;
        u) user="$OPTARG"
           ;;
        p) password="$OPTARG"
           ;;
        ?) printf "Usage: %s -r remotehost\n" $(basename $0) >&2
           exit 2;;
    esac
done

if [ "$user" ] ; then
    auth="--user $user:$password"
fi
```

```

# This script requires InterMapper to be stopped before running.
#/etc/init.d/intermapperd stop

# Get list of top-level file directories from InterMapper
topdirs=$(curl $auth -s http://$remote:$sport/~files | tr '\r' '\n')

for dir in $topdirs ; do
    # Get list of files in this directory
    filelist=$(curl $auth -s $dir | tr '\r' '\n')

    webdir=$(basename $dir)
    echo "Processing $webdir..."
    localdir=""

    echo $filelist | grep "does not exist" >& /dev/null
    if [ $? != 0 ] ; then

        # Convert the web path into the corresponding Setting folder path
        case $webdir in
            icons) localdir="Custom Icons" ;;
            sounds) localdir="Sounds" ;;
            mibs) localdir="MIB Files" ;;
            probes) localdir="Probes" ;;
            tools) localdir="Tools" ;;
            webpages) localdir="Web Pages" ;;
            fonts) localdir="Fonts" ;;
            extensions) localdir="Extensions" ;;
            maps) localdir="Maps" ;;
        esac

        for file in $filelist ; do
            # Get this file and move it into the proper location
            curl $auth -s -O $file
            filename=$(basename $file)

            # Decode the URL to find the real filename
            # Modified from http://do.homeunix.org/UrlDecoding.html to work
            with gnu awk
                local filename=$(echo $filename | \
                    sed 's/+//g' | \
                    sed 's/\%0[dD]//g' | \
                    awk '/%/{while(match($0,/\%[0-9a-fA-F][0-9a-fA-
F]/)){$0=substr($0,1,RSTART-1)sprintf("%c-
",strtonum("0x"substr($0,RSTART+1,2)))substr($0,RSTART+3);}}{print}'))

                # This version works with BSD awk
                # local filename=$(echo $filename | \
                #     sed 's/+//g' | \
                #     sed 's/\%0[dD]//g' | \
                #     awk '/%/{while(match($0,/\%[0-9a-fA-F][0-9a-fA-
F]/)){$0=substr($0,1,RSTART-
1)sprintf("%c",0+("0x"substr($0,RSTART+1,2)))substr($0,RSTART+3);}}{print}'))

                # Make sure the destination directory exists
                local_diname=$(dirname "$local_filename")
                mkdir -p "../$localdir/$local_diname"

```

```
        echo " " $(basename "$local_filename")
        mv $(basename $file) "../$localdir/$local_filename"
    done
fi
done

# Preferences file is separate, since it's stored in the top level of
# the InterMapper Settings directory
curl $auth -s -O http://$remote:$port/~files/Preferences
mv Preferences ../

# restart InterMapper
#/etc/init.d/intermapperd start
```

Chapter 8

Retrieving Collected Data

InterMapper Reports Server is a PostgreSQL database that retrieves data from an InterMapper server and saves it for use by external programs.

Although the InterMapper Reports UI is the easiest way to get data from the database, you can connect to the InterMapper Reports Server database using your own techniques. Several short example reports in Crystal Reports and OpenRPT are available, as well as several example perl scripts. The perl scripts require DBI and DBD::pg.

These scripts have been packaged and zipped and placed on our downloads server, and are available at:

http://download.dartware.com/sql/sql_examples.tar.gz

Please feel free to share your own, as well.

InterMapper Database Schemas

The most up-to-date schema for the InterMapper Database is available at:

[https://\[Your InterMapper Database Server URL\]:-8182/~imdatabase/schemaddl.html](https://[Your InterMapper Database Server URL]:-8182/~imdatabase/schemaddl.html)

It is also available at:

<http://download.dartware.com/schema/imdatabaseschema.sql>

Chapter 9

Customizing Web Pages

InterMapper comes with a set of default web page layouts, and uses them to generate web pages. Use this section to learn how to customize those pages by modifying the files that InterMapper uses to create the pages delivered by its web server.

InterMapper's built-in web server generates pages based on files in its Web Pages directory. When a web request is received, InterMapper finds a corresponding file (called a *target file*) to use as the response. The target file is then formatted according to information specified in a *template file*. The resulting file is returned to the user's web browser.

InterMapper uses the following elements to control the appearance of the web pages returned from its web server:

- [Target files](#) (Pg 183) - Contain the main text of the various pages sent by the server
- [Template files](#) (Pg 185) - Control the overall format of the web pages
- [Directives](#) (Pg 186) - Commands within files to control the formatting of the web pages
- [Quoted links](#) (Pg 189) - Make it easy to create links to other pages.
- [Macros](#) (Pg 190) - Elements you can insert in your templates and target files to show blocks of useful InterMapper information.
- [Web Pages Folder](#) (Pg 196) - Controls which web pages are available to Administrators and Guests.
- [Mime Types](#) (Pg 198) - Associates templates or target files with specific MIME types.

Tip: The target and template files are simply text files. You may edit them with any text editor.

Reloading Changed Web Page Files

The changes you make to these files do not take effect until InterMapper reloads them.

To force InterMapper to reload the Web Page files:

1. From the Edit menu, choose **Server Settings...**
2. From the Server Configuration category, choose **Web Server**. The Web Server settings panel appears.
3. Stop and then restart the Web Server. The changed web pages are reloaded.

Target Files

When InterMapper receives a request for a web page, the requested URL is parsed to determine the target of the request. This *target file* contains the text content of the desired page. The target file may contain HTML markup if desired.

In addition to the page's text, the target file can contain these other elements:

- **Directives** (Pg 186) - Commands that describe or modify the way a page should be displayed.
- **Quoted Links** (Pg 189) - Provide a quick way to create a link to another page using its name, rather than specifying its full URL. If a string is placed in double-quotes (") and the text matches the title of another InterMapper web page, a link is created.
- **Macros** (Pg 190) - InterMapper variables that are replaced with text or formatted HTML in the final web page. The macro may be replaced with a static string, a device's name or network address, the contents of another file, or other information. Macros are composed of keywords and optional parameters, and are enclosed in "\${...}".

Target File Example:

```
#title "This is a test page"
This is some text to be displayed in a
web page. The page's title is "This is a test page", while the remain-
ing
text is displayed in the "body" of the page. The text may also con-
tain
plain text, HTML tagged text such as <b>bold</b> and <i>italic</i>,
and macros, such as the ${date} macro, which displays today's date.
```

- The first line is a directive that sets the title of the page to be displayed.
- The text between double-quotes is placed in <title>...</title> tags in the resulting web page.
- The remainder of this example is placed in the <body>...</body> section of the resulting page. The macro \${date} will be replaced by the current date when the page is displayed.

Quoted Links

Note that the text "This is a test page" will be displayed as a link to its own page, since it is a string in quotes that matches the #title of a web page (its own). Note, too, that the text "body" could be a link to a page with a title of "body". It is not an error if no such page exists: in that case, InterMapper will simply display the quoted string in place. For more information see [Quoted Links \(Pg 189\)](#).

What Happens When a Target File Is Read?

As the target file is read, InterMapper processes the directives, then the expands the macros and creates the tags for any quoted links it encounters. The web server does not insert any white space or paragraph marks (such as <P>) when it encounters carriage returns, etc.

Built-in Target Files

InterMapper provides a number of built-in target files. These file names all begin with "!", and are required because InterMapper refers to them explicitly. The built-in files are:

- **!index.html** - Displays the default page, when none is specified in the URL
- **!document.html** - Displays a graphical image of the specified map.
- **!network.html** - Displays detailed information about the specified network.
- **!device.html** - Displays detailed information about the specified device.
- **!link.html** - Displays detailed information about the specified link.
- **!chart.html** - Displays the specified strip chart.

Template Files

To allow all the web pages to have the same look, InterMapper uses *template files* to control the formatting of pages. A template file is composed of HTML commands that provide the skeleton for a web page. In addition, template files often contain macros and quoted links that are replaced by appropriate text when the page is generated.

Template File Example

Here is a simple template file that could be used with InterMapper:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.2//EN">
<HTML>
<HEAD>
  <TITLE>${pagetitle}</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
  ${imageref:logo.gif}
  ${bodytext}
  ${include:footer.incl}
</BODY>
</HTML>
```

This sample contains several important macros:

- **\${pagetitle}** - Replaced with the text of the #title directive of the target file.
- **\${bodytext}** - Replaced with the body text of the target file, that is, everything from the target file that is not a directive.
- **\${imageref:logo.gif}** - Replaced with an `` tag that refers to the logo.gif file in the ~GuestImages folder on the InterMapper server.
- **\${include:footer.incl}** - Replaced with the contents of the file named 'footer.incl'.

Directives

A *directive* is a special command interpreted by the web server to control the way a page is formatted.

- Directives must start with a "#" in the first column.

Summary of Directives

InterMapper has several directives that can change the way pages are formatted. They are defined below:

#template	<pre>#template "othertemplate.html"</pre> <p>A target file may specify a template file with the <code>#template</code> directive. The <code>#template</code> directive is optional; if none is present, InterMapper uses the file named <code>!template.html</code> as the page's template.</p>
#title	<pre>#title "This is a Test Page"</pre> <p>The text in quotes becomes the title of the page - it enclosed in <code><title>...</title></code> tags in the generated page. The <code>#title</code> directive also provides a destination for quoted links on other pages.</p> <p>Every target file must have a <code>#title</code> directive to give it a name.</p> <p>You may include a macro within the quoted text of this directive. This is useful for inserting the device name or other information into the title of the web page.</p>
#alt_title	<pre>#alt_title "Test Page"</pre> <p>The optional <code>#alt_title</code> directive provides a way to give a page an alternate name that can be used with a quoted link.</p>
#filename	<pre>#filename "otherpage.html"</pre> <p>The optional <code>#filename</code> directive causes InterMapper to treat the file as if named with the quoted string.</p> <p>For example, a target file named <code>"xindex.html"</code> could have a <code>#filename</code></p> <pre>"!index.html"</pre> <p>This causes the target file to be used in place of the file named <code>"!index.html"</code> if its version number is higher. This can be useful for debugging as well as experimenting with alternate pages.</p>
#version	<pre>#version "2.1"</pre> <p>The optional <code>#version</code> directive determines which file is used when there are two or more instances of the same filename (as a result of using <code>#filename</code> directives.)</p>

The optional `#version` directive is used to break "ties" between several files having the same name to determine which should be used. This comes into play in several cases:

- InterMapper has its own internal copy of the web template files, which it uses to create the original set on disk.
- If you edit one of the default web template files, you could change its `#version` to make it take precedence over InterMapper's built-in copy.
- If you edit the file without changing the `#version`, its date-last-modified value is later, causing it to take precedence over InterMapper's built-in copy.
- If you install a new version of InterMapper with updated web files, the `#version` value is incremented, causing the built-in copy to take precedence over the user-edited files,

Note: The user files are not overwritten.

- Through the use of the `#filename` directive, it is possible to have two files on-disk that have "essentially" the same name. One could be named "foo.html", and the other (named something else) can use the `#filename` directive to set its "virtual filename" to "foo.html". In such a case, the `#version` is used to determine which file takes precedence.

Version numbers must be in a 'digit.digit' format. InterMapper uses the file with the highest version number. This is useful for debugging as well as experimenting with alternate pages.

If `#version` directive is not present in the file, the default version, "1.0" is used.

#redirect

```
#redirect "otherpage.html"
```

The `#redirect` directive causes the InterMapper to find `otherpage.html` and use that in place of the original target file.

This can be used to force a well-known page (such as `!index.html`) to display a user-selected page.

Note: This directive creates a static redirection that works only for web pages that exist on the disk when the web server is started. To redirect to web pages that are

generated dynamically by InterMapper (such as map web pages), use the HTML refresh meta tag instead.

#target

```
#target "window_name"
```

The `#target` directive forces a page to be opened in a new window named "window_name".

When generating the web page, InterMapper generates an HREF link with a `...target = "window_name" ...` reference. This causes the detailed information to appear in a separate window when the user clicks a map's device or link.

Quoted Links

You can create a link to another page by entering the page's title in double-quotes. For example, the text "Test Page" creates a link to a page with a `#title` or `#alt_title` directive that contains the text "Test Page".

Note: Two target files may have the same `#title` or `#alt_title`. When this happens, InterMapper chooses one of the target files, but you cannot predict which one is chosen.

Preventing a Quoted String From Becoming a Link

If you place a string in quotes, and the string that does not match another page's `#title` or `#alt_title`, InterMapper displays the quoted string as-is.

You may want text to appear in quotes, even when the text matches another page's `#title` or `#alt_title`. (Remember, you can create quoted links only to pages which have `#title` or `#alt_title` directives, and only quoted text that matches one of those directives results in a link.)

To prevent a string in quotes from being interpreted as a quoted link:

- Place backslashes ("`\`") in front of *both* the first and second quote character.

Macro Reference

A macro is a text string with the format `${macroname:other-information}`. The *macroname* is required, and some macros take *other-information* which follows the ":". The entire macro will be replaced by the appropriate text when the page is generated.

Macros fall into these general categories:

- [The Include Macro \(Pg 190\)](#)
- [Macros that generate "content" of an InterMapper web page \(Pg 190\)](#)
- [Macros that describe InterMapper and its environment \(Pg 192\)](#)
- [Macros to place images onto a page \(Pg 193\)](#)
- [Macros that control the interval between page refreshes \(Pg 193\)](#)
- [Macros related to links and URLs \(Pg 193\)](#)

The Include Macro

Your template files and target files may include other files.

`${include:file-to-be-included.html}` the named file is inserted into the web page. The file **`included.html`** must be in the same folder.

Macros that generate the "content" of an InterMapper web page

InterMapper often uses these macros either as the `${bodytext}` of the page, or as a major part of a page's contents. All the macros below work on the map named in the request URL. If the URL is for a page in the `~admin` directory, InterMapper displays information about all items in all maps.

`${fullstatus}` shows a list of all the devices and links for the map named in the URL.

`${errorstatus}` shows only the devices and links which are in warning or alarm states, or down for the named map.

`${errorstatus_orig}` Output the original "errors" status report. Differences from `${errorstatus}`:

- `${errorstatus_orig}` doesn't show device alarms.
 - `${errorstatus}` first outputs interfaces in error, then interfaces with high utilization.
- `${errorstatus}` lists interfaces in random order.

`${currentoutages}` shows the list of current outages -- devices or links that are currently in warning, alarm, or are down in the named map.

`${currentlinkoutages}` Output a table of current interface outages. The table's column names are ***Date***, ***Time***, ***Interface*** and ***Duration***.

`${previousoutages}` shows the list of devices that had been listed as outages but have since returned to normal.

`${previousoutages:hours=xx}` shows the list of previous outages within the last xx hours.

`${previousoutages:maxrows=x}` shows a list of the last x previous outages.

`${maplist}` shows an HTML unnumbered list (``) of the maps avail-

able.

`${chartlist}` Outputs a sorted list of charts from the current context, one per line with each line preceded by a `` tag. You are required to supply your own `` or `` tags. Each chart title is a hyperlink to the related chart web page.

Within an administrator context, **`${chartlist}`** generates a list of all charts. In a "per-map" context, **`${chartlist}`** generates a list of charts from the current map.

Output the title of the chart related to the current web page. If you are not on a chart-related page, the output is

`${chartname}` "".

Note: Similar to `${mapname}`

`${maplistwithcharts}` shows an HTML unnumbered list of the maps available, with sub-lists of the charts for each map

`${include:file-to-be-included.html}` Inserts the specified file into the web page.

Miscellaneous macros that describe InterMapper and its environment

`${abouthtml}` shows the "About" page with the current version of InterMapper.

`${statshtml}` shows InterMapper's statistics: uptime, memory usage, etc.

`${httpuserid}` the name the user typed when asked for authentication

`${httplocaladdress}` Output the IP address of the web server's side of the connection. If the InterMapper server is multi-homed, this will be the local side IP address of the current TCP connection.

Note: Use caution with this address; URL's produced using this address may break in NAT situations.

`${httpremoteaddress}` the IP address of the remote browser.

`${intermapperaddress}` the IP address of this InterMapper server

`${version}` the version of this copy of InterMapper

`${date}` the current date

`${time}` the current time

`${imagesuffix}` set to ".png" if the web client can display PNG images, or ".jpeg" otherwise.

`${telnetserverurl}` a telnet: URL that connects to this InterMapper Telnet server

`${webserverurl}` a http: URL that connects to this InterMapper server

`${mapname}` the current map's name

`${deviceaddress}` the IP or AppleTalk address of the particular device. For anything that isn't a device, an empty string is returned.

`${deviceid}` Output the device ID of the device related to the current page, in the form: "gMMM-rNN". If the current page is not device-related, output "".

`${devicelist}` Output a table showing the device list for the current context. The table's columns are **Status**, **Name**, **Condition**, **Date**, **Time**, **Probe**, and **Port**.

Within an administrator context, **`${devicelist}`** generates a list of all devices. In a "per-map" context, **`${devicelist}`** generates a list of devices from the current map.

`${devicelist_kml}` Generates a device list in [KML format](#) for use by Google Earth.

`${devicename}` the DNS name or AppleTalk NBP name of the particular device. It is an empty string for anything that isn't a device.

`${ifadmin: ADMIN : NON-ADMIN }` Outputs ADMIN if the user has admin privileges. Otherwise outputs NONADMIN.

`${pagetitle}` displays the value set by the #title directive

`${SetNameFieldWidth:xx}` Set the width of the name field. InterMapper pads the name up to xx characters wide. Use -1 to set the width of the field

to the width of its contents. The default width is 20 chars.

Macros to place images onto a page

`${imageref:IMAGEFILE [,tags]}` creates an `` tag to place an image on the page.

Example:

```
${imageref: photo, class='grade4'}
```

outputs

```
<IMG SRC="/images/photo.gif" class='grade4'>
```

Notes:

Unlike every other macro, this one uses a comma-delimiter in the parameter section instead of a colon.

This macro searches the images folder alphabetically for the first file whose name matches the IMAGEFILE parameter. If you have two files, "photo.gif" and "photo.png", photo.gif is found first.

`${imagesuffix}` set to ".png" if the web client can display PNG images, or ".jpeg" otherwise.

`${intermapperlogo}` creates an `` tag that includes the "Made with InterMapper" logo image.

Macros that control the interval between page refreshes

InterMapper's web server can automatically refresh a particular web page at a desired interval. Include these tags on your page to take advantage of this facility.

`${htmlrefreshmetatag}` is either an empty string or the previous refresh choice from the web client. (Inserts a `<meta http-equiv="refresh" ... >` tag on the resulting page.)

`${htmlrefreshmetaoptions}` is the option list that a web client can choose from. The current `${htmlrefreshmetatag}` value is selected. Note that your HTML template should supply the `<form><select>...</select></form>` surrounding this `${htmlrefreshmetaoptions}` macro.

Macros related to links and URLs

These macros all return a fully-escaped string, that is, a space character (" ") will be replaced with a %20; a "?" with %3F; etc.

Here is a sample URL. The result of using this URL is shown in parentheses after each macro:

```
http://localhost/Map1/device/192.168.0.1%3ASNMP/!device.html
```

`${webpageurl}` the full URL of the requested web page. (e.g., the full URL as shown above)

`${httppath}` the full path to the file requested (e.g., "/Map1-

/device/192.168.0.1%3ASNMP/!device.html")

`${httpdocument}` the top level directory of the page requested. Also an alias for `${mapname}` (e.g., Map1")

`${httpclass}` the second level directory of the page requested (device, chart, link, document, network) (e.g., "device")

`${httpinstance}` the third level directory of the page requested (e.g., "192.168.0.1%3ASNMP")

`${httpmethod}` the fourth-level part of the page requested (e.g., "!device.html")

`${httpinstancepath}` a concatenation of `${httpdocument}`, `${httpclass}`, `${httpinstance}` separated by "/" (e.g., "/Map1-device/192.168.0.1%3ASNMP")

`${httpparam: NAME}` Outputs the value of the HTTP parameter specified by NAME. An HTTP parameter is one passed with the originating GET request, affixed to the URL following a question mark. If there is no HTTP parameter by the given name, outputs "".

Example: Given the following URL:

```
http://www.example.com/TestMap/?color=red&style=bold
```

`${httpparam: color}` outputs "red"

`${httpparam: style}` outputs "bold"

`${httpparam: font}` outputs "" (because the parameter doesn't exist)

`${httpparams}` Output all the HTTP parameters from the originating GET request in their original format. If there were no parameters attached to the original request, output "".

Example: Given the following URL:

```
http://www.example.com/TestMap/?color=red&style=bold
```

`${httpparams}` outputs "color=red&style=bold"

`${httpparams_endchart}` `${httpparams_endchart}` - Replaces the value of the "end-time" parameter with the chart's last time. (For scrolling to the end of the chart).

`${httpparams_prevchart}`

`${httpparams_startchart}`

`${httpparams_timescale}` `${httpparams_nextchart}` - Replaces the value of the "end-time" parameter with a new time value that effectively scrolls the chart one page into the future.

`${httpparams_prevchart}` - Replaces the value of the "end-time" parameter with a new time value that effectively scrolls

the chart one page into the past.

`${httpparams_startchart}` - Replaces the value of the "end-time" parameter with the chart's starting time. (For scrolling to the beginning of the chart).

`${httpparams_timescale: VALUE}` - Replaces the value of the "timescale" parameter with the specified value.

Note: These five macros are nearly identical to `${httpparams}`. They implement support for chart scrolling and scaling in the web interface. They generate output *only* within a web page associated with a chart.

`${anchor: value}` Sets the current "anchor" value.

`${attr}` Outputs the current "anchor" parameter value as set using `${anchor: value}`. If no "anchor" value has been set, output is "".

Note: These two macros are closely related. `${anchor}` sets the value; `${attr}` retrieves it.

Example:

```
${anchor:class="header"}
  <A HREF="maplist.html" ${attr}>Map List</A>
  <A HREF="${TelnetServerURL}" ${attr}>Telnet</A>
  <A HREF="${WebServerURL}" ${attr}>Home</A>
${anchor:}
```

In this example, the anchor is set to 'class="header"'. Then the `${attr}` macro is used to place the attribute string in each link. Afterward, `${anchor:}` sets the anchor to an empty string.

Folder Structure

Web target files and template files are in the *Web Pages* folder within the *InterMapper Settings* folder. Except for the folders described below, the InterMapper web server serves out only those files located in the top level of the *Web Pages* folder.

InterMapper ships with four folders stored in the *Web Pages* folder:

~AdminHTML

This folder contains HTML templates for pages that show the overall status of the InterMapper program. People who have access to these pages may also view all the separate map pages. You can access these files from the default web URL, or by using a URL in this form:

```
http://intermapper.domainname.com/~admin/filename.html
```

~GuestHTML

This folder contains HTML templates for reporting errors such as missing or invalid file names, and for responding to web clients who are not authorized for the web server. These files bypass the usual access list mechanism; you can access them using this URL form:

```
http://intermapper.domainname.com/~error/filename.html
```

~GuestImages

This folder contains images used by the InterMapper web server. These images may be placed in a target or template file using the `${imageref: ... }` macro.

PerMapHTML

This folder contains HTML templates that are used when displaying a map's information. To view a specific document's information, use this URL form:

```
http://intermapper.domainname.com/docname
```

How the Web Page Files are Used

Main Web Page

The main web page is the `~admin/!index.html` target file. When an unqualified URL request arrives (that is, a request for `/`, without any additional path of file information), InterMapper sends out the file specified by `~admin/!index.html`.

Main Template file

By default, all target files use the same template, `!template.html` (note the exclamation point at the beginning of the filename). A target file may specify a different template file by using the `#template` directive.

Default HTML page

For both the `"~AdminHTML"` and `"PerMapHTML"` folders, the default HTML page is `!index.html`. A request for `http://intermapper.domain.com/` is treated like a request for

```
http://intermapper.domain.com/~admin/!index.html.
```

Similarly, a request for

```
http://intermapper.domain.com/docname
```

will be treated like a request for

```
http://intermapper.domain.com/docname/document/main/!index.html.
```

MIME Types

You can associate a template or target file's suffix with MIME-type information you want to send with the file. You create this association by placing a file named "mimetypes" at the top level of the *Web Pages* folder.

Below is a sample mimetypes file:

```
# Sample MIMETypes file
# Format is: <file-suffix> <whitespace> <MIME-descriptor>
wml      text/vnd.wap.wml
wmls     text/vnd.wap.wmlscript
wbmp     image/vnd.wap.wbmp
wbxml    application/vnd.wap.wbxml
wmlc     application/vnd.wap.wmlc
wmlsc    application/vnd.wap.wmlscriptc
```

Tip for Calling Charts

When calling charts through the web server, you can control the height and width of the chart by passing parameters with the URL. You can also control the time scale.

To control the height and width of the chart:

- Enter height & width tags for charts as

```
http://.../!chart.html?height=xxx&width=yyy
```

- Tip: To enter different time scale,

```
http://.../!chart.html?time=XXXXXX
```

Chapter 10

Command-line Options for InterMapper

You can call InterMapper and InterMapper RemoteAccess from a command line, and control a significant number of functions. This can be useful for automating the updating of maps, or for various testing purposes.

For detailed information on the use of the command-line for scripting InterMapper and InterMapper RemoteAccess, see *Command-line Options for InterMapper* and *Command-line Options for InterMapper RemoteAccess* in the User Guide's *Reference* section as well as [InterMapper HTTP API](#) in this manual.

Index

A

Abs	69
Acknowledgements	6
Add	104
Comments	104
Addition, Subtraction	68
ADDRESS	148
Address_type	15, 128
Admin	190
Admin/!index.html	197
AdminHTML	196
Administrators	182
Agreement	6
ALARM	104, 116
Alarm Point	58, 61, 67
Alarm/warning	19
Alert	12
ALRM	116
ALRM Response	128
Alt_title	186, 189
anchor	195
Annotated Example	124
FTP	113
APC-UPS MIB	42, 79
AppleTalk	127, 192
AppleTalk datagrams	8
AppleTalk NBP	192
Argument Format	104
ASN.1	98
Auth	162

Authpassword	162
autorecord	22

B

B/Bold	28
B5/Custom TCP Information/OP	128
Base-64	109
Base64	76, 109
Big-endian	76
Bitand	69
Bitlshift	69
Bitor	69
Bitrshift	69
Bitwise	68
functions	68
Bitxor	69
BODY	185
BODY BGCOLOR	185
Bodytext	185, 190
Branch Not Equal	113
Buffer	113
Build/compile	130, 147
Built-in Macros	109
Built-in Numeric Functions	69
Built-in String Functions	70
Built-in Target Files	184
Builtin	14

C

CALCULATION	43
Calculation Variables	43

Index

Calling Charts	198	Comments	104
Carriage-return	125	Adding	104
Carriage Return	106	Comparisons	48
Case-insensitive	106	Creating	48
Case-Sensitivity	106	ConCATenate	114
Controlling	106	Concatenation	68
Changed Web Page Files	182	Conditional Expression	68
Reloading	182	CONN	113, 128
Chart	43	Connect_timeout	113
CHART LEGEND	43	Connecttime	123
Chart.html	184	Consequently	161
Chart.html?height	198	Contributions	158
Chart.html?time	198	Controlling	106
Chartable	45	Case-Sensitivity	106
Chartlist	191	Copy	6
chartname	191	Software	6
CHCK	113	Cos	69
Com.dartware	14, 17, 128	CPU	40
Com.dartware.tcp.custom	14, 17, 128	CR-LF	118, 128
Comma-separated	41, 43	specify	116
Comma-separated list	15	Creating	48, 130, 148
Command-line	14, 130	Command-line	130
Create	130	Comparisons	48
Command-line Probe	130	Nagios Probes	148
Installing	133	Currentlinkoutages	190
Command Line Interface	199	Currentoutages	190
Command Line Nagios Plug-in Example	143	Custom-snmp	14
Command Line Probes	130	Custom Probe File Format	10
Command List	115	Custom Probes	156
		Installing	156

Custom SNMP Probes	14, 27, 40, 57	Deprecated	120
Header Section	16	DES	162
Custom TCP	14, 17, 21, 128	Des none	162
Custom TCP script	128	Description	18
Dartware-provided probe	128	Device I/O Commands	113
Customizing	27, 182	Device List window	113
Status windows	27	Device.html	184, 193
Web Pages	182	Deviceaddress	192
Customtimer	126	Deviceid	192
CVSPASSWORD	109	Devicelist	192
D		Devicename	192
		Digit.digit	187
Daemon	130	Directives	186
Dartware-provided probe	128	DISC	113
Custom TCP script	128	Discfail	113
Dartware MIB	98	DISCONNECT	129
Datagram	104	display_name	15
Datagrams/sec	27, 57	DNS	17, 192
Date-last-modified	187	DNS name/IP	156
Debug file	162	DOCTYPE HTML PUBLIC	185
Debug window	160	Document.html	184
Open	160	DoD	6
See	162	DONE	113
Debugyyyymmddhhmm.txt	162	Use	114
Default		values	116
Per-second	43	DONE ALRM	116, 129
DEFAULT	43	DONE DOWN	124, 129
Default HTML	197	DONE OKAY	124, 129
Default Values	108	DONE WARN	124, 129
Defines	43	Double-precision	68
Defines:OIDs	43	Double Quote	106

Index

DOWN	8, 104, 116, 128	Extract	75
set	128	substring	75
DOWN Response	128		
		F	
	E		
E-mail	5	FAIL	113, 118
Edit menu	182	FALSE	
ELSE	105	value	68
End	162	File-to-be-included.html	190
MIB	162	File Names	17
Endian	76	FLAGS	16
Enter	164	Fmt	70
snmpwalk	162	Folder Structure	196
Enterprise 6306	98	Footer.incl	185
Equality Tests	68	Format	40, 198
Error Conditions	165	Formfeed	106
Errors/minute	27, 57	Forwarded datagrams	41, 45
Errorstatus	190	From	
Errorstatus_orig	190	Server Settings	182
EVAL	114	FTP	113
Example TCP Probe FileThe	128	Annotated Example	124
Excel	68	number	113
EXIT	113	FTP USER	125
Use	114	Send	113
EXIT_CODE	136	Fullstatus	190
Exp	68	FUNCTION	68
EXPeCT	118	Function Descriptions	70
Expr	68	FUNCTION substr	75
EXPT	113, 129	Functions	68
Use	114	bitwise	68

G		HREF	184, 188
<hr/>		HTML	30, 110, 183, 185-186, 190, 196
GB/Custom TCP Probe/P	128	shows	190
Geneva	28	Use	186
Get-Next-Request	42, 79	Htmlrefreshmetaoptions	193
Get Info window	12	Htmlrefreshmetatag	193
GetNextRequest	161	httpparams_endchart	194
GOTO	105, 113, 129	httpparams_nextchart	194
Use	114	httpparams_prevchart	194
GOTO Command	108	httpparams_startchart	194
Government	6	httpparams_timescale	194
Government End Users	6	Human_name	15, 128
Greetings	121	Hyperlinks	28
Greetings!/r/n	113	making	28
GuestHTML	196	I	
GuestImages	185, 196	<hr/>	
Guests	182	ICMP	16, 127
H		send	16
<hr/>		ID	114
Handling	104	IDLE	108, 129
Script Failures	104	Idlefail	114
HEAD	185	IDLELINE	129
Header Parts	14	If apcups	42, 79
Header Section	16	Ifadmin	192
Custom SNMP Probes	14	IfDescr	161
Hexadecimal	43	IfTable	162
Hexadecimal - Displays	40	IfXTable	163
Hexadecimal Number	106	Ignore	113
Horizontal Tab	106	telnet	113
Hosting		Imagefile	193
InterMapper	130		

Index

Imageref	185, 193, 196	Invoking	164
Imagesuffix	192	snmpwalk	162
Img	185, 193	IP	13, 40, 45, 128, 130, 148, 162, 192
Include	40, 68, 123	IP-address	162
carriage-return	125	IP-address startOID	162
CPU	40	IpForwarded datagrams	48
SNMP MIB	68	IpForwDatagrams	27, 41, 45, 57
WAIT	114	IpInHdrErrors	27, 41, 45, 57
Incoming	113	Issue	118
Telnet	113	LINE OFF command	113
Incorrect login	124		
Index.html	184, 186, 197	J	
Indicates	162	Jpeg	192
SNMP	162	Jumping	108
Installing	133, 156	Label	104
Command-line Probe	130		
Custom Probes	156	L	
Integer	69	Label	104
InterMapper Logs	164	Jumping	108
InterMapper Remote	199	Label_name	108
InterMapper Scripting Language	113	LDAP	8
InterMapper Server		Len	70
testing	199	Level	196, 198
InterMapper Settings	17, 147, 164, 196	Web Pages	196, 198
Intermapper.domain.com	197	LF	118
InterMapper®	6	Limit	17
Intermapperaddress	192	Macintosh file	17
Intermapperlogo	193	LINE	106, 113, 128
InterMapperTimeStamp	45	Use	114
		LINE OFF	118

LINE ON	118	MD5	162
Link	189	Md5 sha none	162
Link.html	184	Measuring	126
Link/alias/shortcut	147	Response Times	126
Little-endian	76	Merchantability	6
Localhost/Map1/device/192.168.0.1%3ASNMIPto193		Meta http-equiv	187
Logical And	68	MIB	40, 43, 162
Logical Or	68	end	162
Login	113	MIME-descriptor	198
Logo.gif file	185	MIME-type	198
M		MIME Types	198
M1++/Big	28	Mimetypes	198
Mac OS	130	Mimetypes file	198
Macintosh file	17	Min	68
limit	17	MINIMAL	16
Macro Reference	190	Modulo	68
Macroname	190	Monaco	28
Main Template file	197	Monospace	28
Main Web Page	197	Monospaced	27-28, 57
maintenance mode	60	Msecs	129
Maplist	190	MTCH	105, 113, 129
Maplistwithcharts	191	Multiplication, Division	68
Mapname	192	MUST	104
Markup Commands	28	N	
Markup Tag Summary	28	N/A	160
MaTCH	118	NADD	114
Matches	49, 68, 106	Nagios	130, 147
String	40, 68, 104	uses	130
Maxrows	190	Nagios Plugins	130, 147
Maxvars	42, 79		

Index

Nagios Probes	148
Creating	148
Nagios Template	147
Name	130
program/script	130
Name/value	19
NBGT	114
NBNE	114
Network.html	184
NEXT	113
NOICMPFALLBACK	16
NOLINKS	16
Nomib2	42, 79
Non-interpreted	133
NOOP	124
NOOP/r/n	113
NoSuchName	160
NTCREDENTIALS	16
NULL	160
Num	104
Num-OIDs	162
Number	41, 45, 113, 162
FTP	113
OIDs	162
TCP	41
Number:TCP	45
Numeric Add	113
Numeric Argument Format	104
Numeric Branch Greater Than	113
Numeric Branch Not Equal	113
Numeric Comparisons	48

O

Object ID	43
Octal Number	106
OctetString	164
OFF	113
OID	41, 43, 160, 162
number	162
OID Error	161
OID:defines	43
OID:request	43
OK	104, 128
OK Response	128
OKAY	116
Old_protocol	17
Old_script	17
ON	113

P

P/Plain Text	28
Pagetitle	185, 192
Param	109
Parameter Section Example	21
PASS	124
PASSWORD	19
Password Fields	19
PATH	134
Pdtype	42, 79
Per-minute	43
PER-SECOND	43
Perl	68, 130

Index

Runnable/executable	147	Set	128
		DOWN	128
		SetNameFieldWidth	192
		Simple snmpwalk facility	162
		SineValue	45
		SKIP	113
		SNMP	8, 14, 27, 40, 43, 79, 98, 160, 162
		Snmp-device-display	10, 27, 41, 45, 57, 126
		Snmp-device-properties	10, 41, 79
		Snmp-device-threshold	10-11, 41
		Snmp-device-thresholds	10-11, 41
		Snmp-device-variables	10-11, 41, 43
		SNMP FAQ	160
		SNMP Get-Next-Request	43
		SNMP Get-Request	42, 79
		SNMP MIB	68
		SNMP OID	41
		SNMP Probe Variables	41
		SNMP Query Settings	42
		SNMP Response	160
		SNMPv1	162
		SNMPV2C	16, 162
		SNMPv3	162
		Snmpwalk	162
		Invoking	164
		Snmpwalk stopall	164
		SNMPwalks	164
		Special Character Example	106
		Special Characters	106
<hr/>			
S			
<hr/>			
Sample <snmp-device-threshold	40		
Sample <snmp-device-variables	40		
Sample Header Section	16		
Sample MIMETypes file	198		
SBNE	114		
Scanning	118		
Probe Command Reference	113		
SCAT	114		
Script	113		
Script Command Format	104		
Script Failures	104		
Handling	104		
Script Process Flow	104		
Script Termination	108		
Seconds	19		
Secs	70, 114, 129		
Select Misc	162		
Select Probe window	15		
SEND	16, 113, 128		
FTP USER	125		
ICMP	16		
Use	114		
Sending	8, 162		
SNMP	8		
Server Command	164		
Server Settings	182		
Servers	162, 182		

Specifies	113	Substr	70
CR-LF	118	Substring	75, 128
Sprintf	70	Extract	75
Sqrt	69	SysContact	49
Start	162	SysDescr	41, 45
SNMP	162	SysUptime MIB-2	42, 79
StartOID	162		
STAT	113	T	
STAT ALRM	122	Target File Example	183
State	6	Target File Is Read	184
Statshtml	192	TCP 8, 14, 27, 41, 45, 104, 116, 128	
Status	27	Number	41
Status Window Text	41	TCP-based	27
Status windows	27, 41	Tcp-script	14, 128
Customizing	27	TCP Script Commands	126
STOR	114	TCP Timers	126
Str	68	Tcp.custom	14, 17, 128
Strftime	70	TCP:Number	45
String	40, 68, 104, 113	TcpCurrEstab	27, 41, 45, 57
Argument Format	104	Telnet	113, 165, 192
Branch Not Equal	113	allows	116
ConCATenate	114	ignore	113
foobar	122	incoming	113
Matches	49	Telnet window	165
Matching	68, 104	Telnetserverurl	192
Strlen	70	Template File Example	185
Strptime	70	Template Files	185
STRT	114, 126	Template.html	186, 197
STRT Starts	113	Test Page	186, 189
Stylings	28	Thresholds	40
Sub-expressions	68	TIME	113

Index

Time Measurement Probe Variables	126	Uptime	192
TIME varname	126	URL	183, 190, 196, 198
Timeout	113, 129, 162	Url-to-invoke	15
Timeout 60	123	Url_hint	15
Timestamp	45	Used	196
TimeTicks	164	USER	116
Title	185	recognize	124
TOO_LONG	121	User ID	124
Tools	147	Username	162
Total-value	43		
Transfer Control	109	V	
Using Relative Offsets	109	VALUE	68
Traps		Values	21, 43, 68, 116
TrapVariable	87	DONE	113
TrapVariable	87	FALSE	68
TRUE		PI	69
value	68	TRUE	68
Trunc	69	x1	69
Type	43	Values:3.14159265	45
		Var	43, 68
U		Variable-name OID	40, 43
UDP	8	VariableName	43
UL	190	Variables	104
Unary	68	Varname	126
Unexpected	110	Vertical Tab	106
Unix	130		
Unix Linefeed	106	W	
Unless-e	162	WAIT	107, 114, 128
UP	104	include	123
UPPER CASE	104	Use	114
UPPERCASE	116		

WAIT timeout	113
Wait/p	128
i/Seconds	128
WARN	104, 116
WARN Response	109, 128
Web Page Files	197
Web Pages	182, 196, 198
Customizing	182
level	196, 198
Web Server	182
Webpageurl	193
Webserverurl	192
Whitespace	198
Wild-card Character Matching	107
Window_name	188
Windows	130
probes	130