FORTRA

Intermapper
6.6.3

Developer Guide
January 2025

Copyright Terms and Conditions

202412230819

Table of Contents

Creating Your Own Probes

For many Internet services, simply pinging a device is not a sufficient test of whether it is operating correctly or not. Intermapper includes built-in probes that can test device operations and whether it is a web server, router, database, LDAP server, and so on.

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

What is a Probe?

A probe is a text file that specifies how Intermapper tests a device. It is essentially a plug-in. All of Intermapper's probes use the following logic:

- The probe sends one or more queries as SNMP requests, UDP 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 responses from the device and sets the device's status accordingly.

Probe Parts

All of the probe types listed below follow a similar structure, which is outlined in Anatomy of A Probe, which explains the common sections of a probe and sections that are specific to a particular probe type.

Probe Types

Intermapper includes several kinds of probes. You can use Intermapper's built-in probes as-is, copy and modify them, or create your own probes. The following are the available probe types:

- **SNMP Probes** - Intermapper sends SNMP queries and compares the results to user-specified thresholds to test the device's status.

- **SNMP Trap Probes** - 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** - 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 use the TCP Probe Scripting Language to create a sequence of commands, branching to different parts of the script under specified conditions.
- **Command-line Probes** - Intermapper can invoke a program or script (as if they are 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.

You can modify existing files to create new probes. If you create a new probe file that might be useful, consider sending it to Fortra. For more information, see Sharing Probes.

Anatomy of a Probe

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

Each section contains lines bracketed by the following:

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

You can open a separate window with the Example TCP Probe File while reading the subsequent sections.

Sections Common to All Probes

<header> Section

The `<header>` section specifies the probe definition, including how the probe is identified, how its name appears in the Probe Type menu, and the version numbering system.

<description> Section

The `<description>` section specifies the help text that appears in the Set Probe window and explains the function of the probe and the use of its parameters. Format the description using IMML. For more information on IMML, see InterMapper's Markup Language.

<parameters> Section

The `<parameters>` section defines the probe's parameters and how they are presented in the Set Probe 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 can format the appearance of Status window using IMML. For more information on IMML, see InterMapper's Markup Language.

Type-Specific Probe Sections

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

Sections Specific to SNMP Probes

Each custom SNMP probe includes the following:

- **`<snmp-device-variables>` section** - specifies which MIB variables are collected by the device.
- **`<snmp-device-thresholds>` section** - specifies how variables are tested against thresholds to determine the status of the device.
- **`<snmp-device-display>` section** - specifies the device information and links that is displayed in the Status window.
- **`<snmp-device-properties>` section** - specifies certain aspects of the SNMP queries sent to the device.
- **`<snmp-device-alarmpoints>` Section** - allows you to define conditions where the device changes a particular device state.

Sections Specific to SNMP Trap Probes

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

- **`<snmp-device-variables>` section**- specifies which MIB variables to collect from the device. These are set automatically when a trap is received.
- **`<snmp-device-thresholds>` section** - specifies how variables are I dotested against thresholds to determine the device's status.

Sections Specific to TCP Probes

Each custom TCP probe can have the following:

- `<script>` section, which uses a sequence of commands and program flow that is similar to Basic. A rich set of commands is available. For more information on the command sets, see TCP Probe Command Reference.
- `<script-output>` section

Sections Specific to Command Line Probes

Each command-line probe includes the following:

- **`<command-line>` section** - builds the command-line, specifying the command path and command parameters.
- **`<command-exit>` section** - allows you to control the state of the device, depending on what is returned from the script.
- **`<tool>` section** - contains the code for the companion script that is run by the probe.
- **`<command-display>` section** - allows you to control what appears in the Status window for the device.

Additional Probe Information

Comments

All probes use the same format for comments, which is similar to HTML comments.

Probe File Locations and Probe File Names

To use probe files, you must import them. Follow recommended naming conventions.

Installing and Reloading Probes

Before a modification is applied to a probe, you must click **Reload probes** in the Set Probe window (circular arrow icon below the left pane of the window).

`<header>` Section

The `<header>` section of a probe file contains a formal description of the probe, with each header property having a name and a corresponding value. For example,

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

> **NOTE:**
> Information by which Intermapper uniquely identifies the probe is contained in the header. While it is not required, Fortra strongly recommends that you follow probe file naming conventions that correspond to the unique identifer 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

Specifies the first part of the probe's full identifier. Typically, this includes the domain name of the organization that created the probe, with the labels reversed.

For example, for all probes created by Fortra, the package statement is as follows:

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

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

> **NOTE:**
> The combination of [package].[probe_name] together forms the probe's full identifier. (In the example below, the full identifier is com.dartware.tcp.custom.) By default, the name of the file that includes the probe definition is the same as the probe's full identifier. This is not required, but it is recommended. For more information, see Probe File Locations and Probe File Names.

probe_name

Specifies the second part of the probe's full identifier. The probe_name can be a custom, unique string.

human_name	Specifies the string that appears in the left pane of the Set Probe window. This string helps guide you in selecting a probe for a particular device.
version	Specifies the version of the probe file. The format of the version is #.#.
address_type	Specifies a comma-separated list of one or more address types. Intermapper implements IP and AT.
port_number	Specifies the IP port used by the probe.

display_name	Specifies the display_name of the probe, using forward slashes (/) to specify the hierarchy. To do this, add the following line to the `<header>` section of the probe:
	`display_name = "[top level]/[next level]/[next level]"`
	For example,
	`display_name = "Custom/Command-line"`

url_hint	Assigns a double-click action within the probe (making it the predefined, double-click action). To do this, add the following line to your `<header>` section of the probe:
	`url_hint = "url-to-invoke"`
	For example, the following invokes the web browser to the device's IP address and port:
	`url_hint = "http://${address}:${port}"`

poll_interval	Sets the default poll interval of the device to the indicated number of seconds. This overrides the default setting of the map and can be used to avoid too-frequent polling for (physical) devices that should not be polled too often. Setting the poll interval *for the device* overrides the poll_interval setting. `poll_interval = "300"`

Sample Header Section

The following 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 custom SNMP probe file is similar to the standard `<header>` section, with the following differences:

- The custom SNMP probe type is custom-snmp.
- A FLAGS=xxx,xxx command is available that uses the following optional items as parameters:
  - **NOLINKS** - Intermapper does not poll links (interfaces) with SNMP.
  - **SNMPV2C** - Intermapper uses SNMPv2c to poll the device.
  - NOICMPFALLBACK - Intermapper does not send an ICMP ping to a device if no SNMP responses are returned.
  - **MINIMAL** - the probe queries only its own (specified) variables.
  - **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 instructs Intermapper to allow this situation. If the SNMP agent in your network device does not 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 stops.
  - **IFINDEX-BUG** - Some network equipment responds 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 device link goes down and this flag is set, the device status changes to critical instead of the default of alarm.

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

Flags for Command-Line Probes

The following Flag parameters are specific to command-line probes:

- **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. This is for Microsoft Windows systems only.
- **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. For more information, see Nagios Plugins.

Probe File Locations

Probe files are saved in the **Probes** folder of the **Intermapper Settings** directory.

Probe File Names

Probe files are named with the following parts, separated by a period:

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

  By default, the package is composed of the organization DNS domain name, with the segments reversed. For example, built-in probes for Intermapper have a package called com.dartware. Other organizations might create and share their own probes, since the file names must be unique.
- **probe name** - the name of the probe.

For example, the built-in custom TCP probe is defined in the following file:

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

The package name and probe name are defined in the probe definition's `<header>` section. Fortra 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 the following lines:

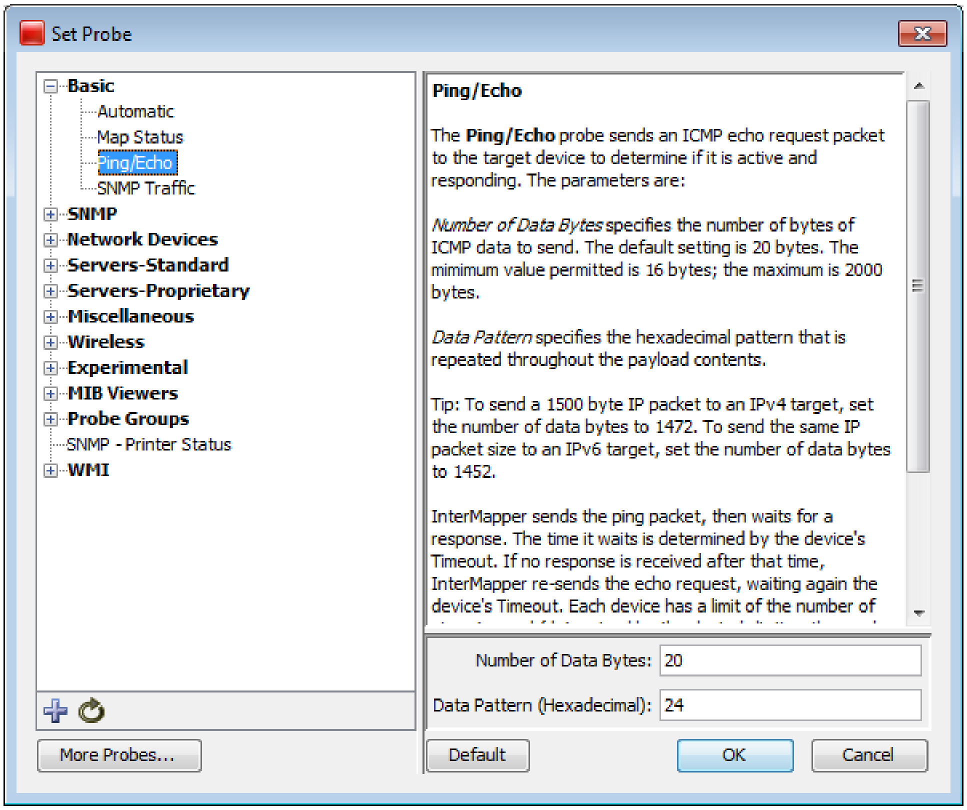
```
package = "com.dartware"
probe name = "tcp.custom"
```

`<description>` Section

The `<description>` section of a probe file contains the description of the probe in the Set Probe window. All probe types can have a description section, defined using the following tags:

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

The `<description>` section can be formatted using IMML, Intermapper's Markup Language. The Example Probe File shows a sample `<description>`section.



Ping/Echo

The **Ping/Echo** probe sends an ICMP echo request packet to the target device to determine if it is active and responding. The parameters are:

Number of Data Bytes specifies the number of bytes of ICMP data to send. The default setting is 20 bytes. The mimimum value permitted is 16 bytes; the maximum is 2000 bytes.

Data Pattern specifies the hexadecimal pattern that is repeated throughout the payload contents.

Tip: To send a 1500 byte IP packet to an IPv4 target, set the number of data bytes to 1472. To send the same IP packet size to an IPv6 target, set the number of data bytes to 1452.

InterMapper sends the ping packet, then waits for a response. The time it waits is determined by the device's Timeout. If no response is received after that time, InterMapper re-sends the echo request, waiting again the device's Timeout. Each device has a limit of the number of

Number of Data Bytes: 20

Data Pattern (Hexadecimal): 24

The Set Probe window shows the `<description>`field.

`<parameters>` Section

A probe can have one or more parameters. These parameters are customizable in the Set Probe window and 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 following format:

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

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

Probe parameters are accessed and used similarly to how variables are accessed and used. They can be used in calculations, alarm/warning thresholds, and displayed in the status window. To reference a parameter that has a name with one or more spaces, enclose the name in curly braces ({ }). For example, ${Seconds to wait}.

Input Field Types

The following input field types are available:

- **Text** - inputs a text string.
- **Password** - inputs a text string, obscuring the characters.
- **Dropdown** - selects from a menu.
- **Checkbox** - sets a variable to true or false by selecting or clearing a check box.

Text Fields

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

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

The line above sets the variable `${Text}`.

Password Fields

You can create input parameters that conceal the string from casual view (*password parameters*). The data is displayed as a string of asterisks (*****) when a user types the password. To specify a password parameter, type a single asterisk (*) after the name of the field. For example,

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

Note that the variable name remains ${Password*} and you have to reference it as such in your script. The asterisk (*) is removed before displaying the name, so the above password parameter appears as Password in the Set Probe window.

Dropdown Fields

You can create input parameter fields that present a dropdown menu that includes selectable options.

To create a dropdown field, use the following syntax:

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

The values enclosed in brackets are the available options. The value to the 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 selected value of the dropdown. To display the value of a dropdown in the Status window, use the full variable definition. For example,

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

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

Checkbox Fields

To create a checkbox, use the following syntax:

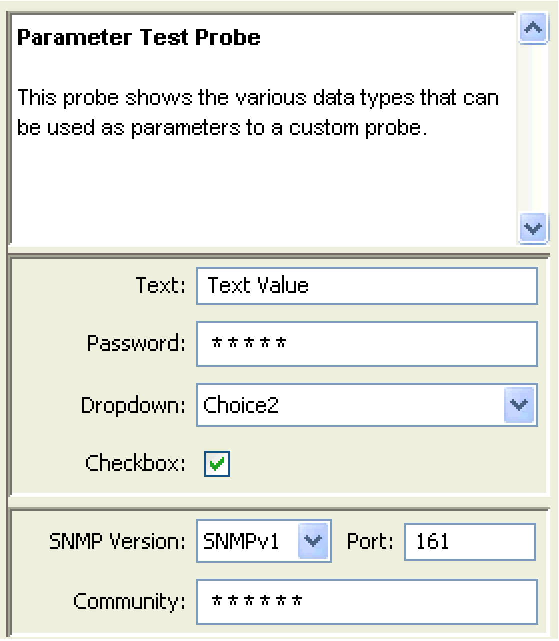
```
"Checkbox[true,false]" = "true" //Default value is "true"
```

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

Parameter Section Example

The following is an example `<parameter>` section that demonstrates the use of the four types of input fields. Each input field type appears as follows:

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



Parameter Test Probe

This probe shows the various data types that can be used as parameters to a custom probe.

Text: Text Value

Password: *****

Dropdown: Choice2

Checkbox: ☑

SNMP Version: SNMPv1 Port: 161

Community: ******

`<datasets>` Section

Use the `<datasets>` section to define the datasets for a probe. You can also specify which datasets are recorded by default.

> **NOTE:**
> The `<datasets>` section replaces the deprecated `<autorecord>` section.

The following syntax is used in the `<datasets>` section. All columns except Column 1 should be enclosed in double quotation marks (" ").

```
<datasets>
  $variablename, "tag", "unitsOfMeasure", "autorecordFlag", "legend"
  ...
</datasets>
```

where:

- **$variablename** - is a variable defined by the probe.
- **tag** - is a short tag that identifies a dataset class. Use these tags to create a report of similar variables, such as CPU% or temperature. To view pre-defined tags, see the Automatically-Recorded Data Values. Probe writers can create their own short tags as long as they do not start with an underscore (_).
- **unitsOfMeasure** - is the unit of measure used with the dataset. Select from the list of Units of Measures below.
- **autorecordFlag** - is a Boolean flag that specifies whether the dataset should be recorded or not.
- **Legend** - is a human-readable text string that appears as the legend label for the dataset. This legend overrides a legend placed in the `<snmp-device-variable>` section.

Example

```
<datasets>
  $temp, "temp-tag", "degrees C", "false", "The Temperature"
  $atemp, "atemp-tag", "degrees C", "true", "Autorecord Temperature"
</datasets>
```

Auto-Recording Values

Certain data values collected from a device are recorded to the Intermapper database automatically. You can specify other variables to record by default when data for a device is stored.

The following data is recorded for all probes:

- 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. For a list of values for each built-in probe, see Automatically-Recorded Data Values.

For your own probes, you can specify that a dataset is recorded by setting the `autorecordFlag` value to **true**.

Units of Measure

Use the following units in the `unitsOfMeasure` column of the `<datasets>` section:

Symbol	Description
percent	percentage
min	minutes
sec	seconds
msec	milliseconds
bytes	bytes
kbytes	kilobytes
packets	packets
errors	errors
discards	discards
frames/sec	frames per second
bytes/sec	bytes per second
bits/sec	bits per second
mbits/sec	megabits per second
discards/min	discards per minute
errors/min	errors per minute
errors/sec	errors per second
failures/sec	failures per second
retries/sec	retries per second
packets/sec	packets per second
requests/sec	requests per second
degrees C	degrees celsius
degrees F	degrees fahrenheit
dBm	the power ratio in decibels of the measured power referenced to one milliwatt
miles	a measure of wireless transmission range, (for how many miles it is useful)
volts	voltage

<autorecord> Section (Deprecated)

The `<autorecord>` section has been replaced by the `<datasets>` section, which provides a control for auto-recording any dataset. The `<autorecord>` section is available for backward compatibility. The `<autorecord>` section uses the following syntax:

```
<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 similar variables, such as CPU% or temperature. To view predefined tags, see the Automatically-Recorded Data Values. Probe writers can create their own short tags as long as they do not start with an underscore (_).
- **Legend X :units(xxx)** is a human-readable text string that describes the dataset and shows what kind of data is collected for a given device. Specify the units for the dataset using the optional `:units` attribute. This legend overrides the legend in the `<snmp-device-variable>` section. In the `<datasets>` section, :units(xxx) has been replaced with `unitsOfMeasure`.

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

The following values are recorded automatically from built-in probes:

Probe Name/ File Name	Variable Name	Tag(30)	Units	Legend (255)
Miscellaneous/Legacy/Cisco (v2c) com.dartware.snmpv2c.cisco	$lcpu.busyPer	cpupercent	percent	CPU Percent Busy

	$lcpu.avgBusy1	cpupercentavg	percent	Average CPU Percent over 1 min
	$lcpu.avgBusy5	cpupercentavg	percent	Average CPU Percent over 5 min
	$lmem.freeMem	freemem	bytes	Available Memory
Miscellaneous/TCP Check com.dartware.snmp.tcpcheck	$tcpCurrEstab	numconns		Number of TCP Connections
Network Devices/Cisco/Cisco - IP SLA Jitter com.dartware.snmp.cisco-ip-sla.txt	$cpmCPUTotal1min	cpupercentavg	percent	Average CPU Percent
	$AvgJitter	jitteravg	msec	Average Jitter Value
	$AvgLatency	latencymsec	msec	Average Latency
	$PercentPacketLoss	pktloss	percent	Jitter Test Packet Loss
Network Devices/Cisco/Cisco - Old CPU MIB com.dartware.snmp.cisco	$lcpu.busyPer	cpupercent	percent	CPU Percent Busy
	$lcpu.avgBusy1	cpupercentavg	percent	Avg. CPU Percent over 1 min
	$lcpu.avgBusy5	cpupercentavg	percent	Avg. CPU Percent over 5 min

	$lmem.freeMem	freemem	bytes	Available Memory
Network Devices/Cisco/Cisco - Process and Memory Pool com.dartware.snmp.cisconewmib	$lcpu.busyPer	cpupercent	percent	CPU Percent Busy
	$lcpu.avgBusy1	cpupercentavg	percent	Avg. CPU Percent over 1 min
	$lcpu.avgBusy5	cpupercentavg	percent	Avg. CPU Percent over 5 min
	$ciscoMemoryPoolFree1	freemem	bytes	Available Memory #1
	$ciscoMemoryPoolFree2	freemem	bytes	Available Memory #2
Network Devices/UPS/APC UPS - AP961x com.dartware.ups.apc-ap961x.txt	$leftCharge	pctcharge	percent	Percent Charge
	$batMin	batttimeleft	min	Time left on battery
	$inVolt	involts	volts	Input Voltage
	$batTempC	temperature	degrees C	Battery Temperature (°C)
Network Devices/UPS/APC UPS com.dartware.ups.apc.txt	$leftCharge	pctcharge	percent	Percent Charge
	$batMin	batttimeleft	min	Time left on battery
	$inVolt	involts	volts	Input Voltage

	$batTempC	temperature	degrees C	Battery Temperature (°C)
Network Devices/UPS/BestPower UPS com.dartware.ups.bestpower.txt	$cTimeOnBattery	batttimeleft	min	Time Left on Battery (min)
	$cInputVoltage	involts	volts	Input Voltage
	$cIntTempC	temperature	degrees C	Internal Temperature (C)
Network Devices/UPS/Exide UPS shef.ac.uk.ups.exide.txt	$LeftCharge	pctcharge	percent	Battery Charge Left
	$LeftMin	batttimeleft	min	Time Left on Battery
	$in1Volt	involts	volts	Input 1 Voltage
Network Devices/UPS/Liebert UPS - OpenComms com.dartware.ups.liebert-opencomms.txt	$LeftCharge	pctcharge	percent	Percent Charge
	$LeftMin	batttimeleft	min	Time Left on Battery
	$in1Volt	involts	volts	Input 1 Voltage
	$batteryTempC	temperature	degrees C	Battery Temperature (°C)
Network Devices/UPS/Standard UPS (RFC1628) com.dartware.ups.standard.txt	$LeftCharge	pctcharge	percent	Percent Charge
	$LeftMin	batttimeleft	min	Time Left on Battery

	$in1Volt	involts	volts	Input 1 Voltage
	$batTempC	temperature	degrees C	Battery Temperature
Network Devices/UPS/TrippLite UPS com.dartware.ups.tripplite.txt	$LeftCharge	pctcharge	percent	Percent Charge
	$LeftMin	batttimeleft	min	Time Left on Battery
	$in1Volt	involts	volts	Input 1 Voltage
	$envTempC	temperature	degrees C	Ambient Temperature (°C)
	$envHumid	humidity	percent	Ambient Humidity
Network Devices/UPS/Victron UPS de.medianet.freinet.ups.victron.txt	$batt.rem	batttimeleft	min	Battery Time Remaining
	$input.volt1	involts	volts	Input Voltage Phase 1
Servers-Proprietary/Apple/OS X Server/AFP com.dartware.tcp.osxserver.afp.txt	$currentConnections	connections		Connections
	$currentThroughput	throughput	bytes/sec	Throughput
Servers-Proprietary/Apple/OS X Server/FTP com.dartware.tcp.osxserver.ftp.txt	$realConnectionCount	authconns		Authenticated Connections

	$anonymousConnectionCount	anonconns		Anonymous Connections
Servers-Proprietary/Apple/OS X Server/Info com.dartware.tcp.osxserver.info.txt	$cpu	cpupercent	percent	CPU Usage
Servers-Proprietary/Apple/OS X Server/NAT com.dartware.tcp.osxserver.nat.txt	$activeTCP	tcpconns		TCP Links
	$activeUDP	udpconns		UDP Links
	$activeICMP	icmpconns		ICMP Links
Servers-Proprietary/Apple/OS X Server/Print com.dartware.tcp.osxserver.print.txt	$currentQueues	queues		Current Queues
	$currentJobs	numjobs		Spooled Jobs
Servers-Proprietary/Apple/OS X Server/QTSS com.dartware.tcp.osxserver.qtss.txt	$currentConnections	connections		Connections
	$currentThroughput	throughput	bytes/sec	Throughput
Servers-Proprietary/Apple/OS X Server/Web com.dartware.tcp.osxserver.web.txt	$currentRequestsBy10	requestrate	requests/sec	Request Rate
	$cacheCurrentRequestsBy10	requestrate cache	requests/sec	Cache Request Rate

	$currentThroughput	throughput	bytes/sec	Throughput
	$cacheCurrentThroughput	throughputcache	bytes/sec	Cache Throughput
Servers-Proprietary/Barracuda/Barracuda HTTP com.dartware.tcp.barracuda.http.txt	$in_queue_size	inqueue		Input Queue
	$out_queue_size	outqueue		Output Queue
	$avg_latency	latencysec	sec	Average Message Latency
Servers-Proprietary/Barracuda/Barracuda HTTPS com.dartware.tcp.barracuda.https.txt	$in_queue_size	inqueue		Input Queue
	$out_queue_size	outqueue		Output Queue
	$avg_latency	latencysec	sec	Average Message Latency
Servers-Proprietary/Microsoft/DHCP Lease Check com.dartware.snmp.dhcpcheck.txt	$noAddFree	dhcpfree		Number of DHCP Leases Free
	$noAddInUse	dhcpinuse		Number of DHCP Leases In Use
	$noPending	dhcppending		Number of Pending Offers

Servers-Standard/Custom TCP com.dartware.tcp.custom	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/Host Resources com.dartware.snmp.hrmib	$_CPUUtilization	cpupercentavg	percent	Average CPU Percent
Servers-Standard/HTTP & HTTPS/HTTP (Follow Redirects) com.dartware.tcp.http.follow	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTP (Post) com.dartware.tcp.http.cgi.post	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTP (Proxy) com.dartware.tcp.http.proxy	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTP (Redirect) com.dartware.tcp.http.redirect	$_connect	conntime	msec	Time to establish connection

	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTP com.dartware.tcp.http	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTPS (Follow Redirects) com.dartware.tcp.https.follow	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTPS (Post) com.dartware.tcp.https.cgi.post	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTPS (SSLv3) com.dartware.tcp.https.notls.txt	$_connect	conntime	msec	Time to establish connection
	$_active	connactive	msec	Time spent connected to host
Servers-Standard/HTTP & HTTPS/HTTPS com.dartware.tcp.https	$_connect	conntime	msec	Time to establish connection

	$_active	connactive	msec	Time spent connected to host
SNMP/Comparison com.dartware.snmp.oidcomparison.txt	$theOID	$Tag	$Units	$Legend
SNMP/High Threshold com.dartware.snmp.oidhigh.txt	$theOID	$Tag	$Units	$Legend
SNMP/Low Threshold com.dartware.snmp.oidlow.txt	$theOID	$Tag	$Units	$Legend
SNMP/Range Threshold com.dartware.snmp.oidrange.txt	$theOID	$Tag	$Units	$Legend
SNMP/Single OID Viewer com.dartware.snmp.oidsingle.txt	$theOID	$Tag	$Units	$Legend
SNMP/String Comparison com.dartware.snmp.oidstrcomparison.txt	$theOID	$Tag	$Units	$Legend

Probe Status Window

When you create a custom probe, you can override the default contents of the Status window. How you do this depends on the type of probe. For example,

- `<snmp-device-display>` **section** - for SNMP probes
- `<snmp-device-display>` **section** - for SNMP trap probes
- `<script-output>` **section** - for TCP probes
- `<command-display>` **section** - for 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 the Status window of a custom SNMP probe. Probe variables are replaced by their values in the Status window.

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

The following is a sample `<snmp-device-display>` section. Variables are replaced with the values retrieved from the device, and that formatting is controlled by IMML.

```
<snmp-device-display>
  \B5\Custom SNMP Probe\0P\
  \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 Status window for the TCP-based custom probe. 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 the Status window for a command-line-based custom probe. 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 the probe description text or to the Status window content using IMML, Intermapper's markup language. IMML consists of formatting commands bracketed by backslashes (\). There might be many markup commands between a pair of **\...** characters. The Example Probe File shows a sample description section.

> **NOTE:**
> Prior to Intermapper 4.0, the markup characters were « and » (`&le;` and `&ge;`). Intermapper still accepts these characters, although Fortra recommends that you use the **\...** in new probe files as they are easier to type and can pass unchanged through all mail systems.

How Markup Tags are Applied

- A markup command applies to all text that follows it.
- Subsequent markup tags can be added to or counteract a previous set of markup tags.

Markup Tag Summary

Tag	Action
M	Applies a mono-spaced font.
G	Sets the font to Geneva or other proportional-spaced font.
+	Increases the font size by one. Multiples (++) increase the font size by the corresponding amount.
-	Decreases the font size by 1. Multiples are allowed.
B	Sets following text in bold.

I	Sets following text in italics.	
P	Sets following text to plain. Setting text to plain overrides all other style settings.	
U	Sets following text to underlined. See **Creating a link** below for making hyperlinks.	
!	Turns off an applied format.	
digit	Sets 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:

`\b\Bold \i\Bold Italic \!b\Italic \p\Plain`	**Bold** ***Bold Italic*** *Italic* Plain
`\M1++\Big red monospace\p\`	`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 show the script code required to create a link. In both examples, \2U\ sets the color to blue and underlines 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 (for example, http://www.example.com), Intermapper 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 can be interspersed anywhere in a probe file.

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 of -- at the beginning of a line. The remainder of the line is ignored.

Example

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

Built-In Probe Variables and Macros

The following are the built-in variables available in custom probes and notifiers:

> **NOTE:**
> Some variables are available only in certain contexts. The variables are listed by context.

- Command Line Probe Variables
- SNMP Probe Variables
- TCP Probe Variables
- Command Line Notifier Variables
- ${chartable} Macro

- ${eval} Macro
- ${scalable10} and ${scalable2} Macros

Command Line Probe Variables

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

<command-line> and <command-exit> Sections

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

Variable Name	Variable Description
${address}	Specifies the network address of the device.
${devicename}	Specifies the name of the device. In some cases, the device name can resolve to the IP address of the device.
${port}	Specifies the monitored network port number.
${exit_code}	Specifies the exit code of the command line probe. The ${exit_code} variable is used in <command-exit> only.
${cscript}	Evaluates the full path to the cscript.exe utility; it also automatically adds /nologo as a command line option. This variable is only available for Microsoft Windows systems.
${python}	Evaluates the full path of the Python interpreter installed as part of the Intermapper datacenter. It also automatically adds necessary command line options for normal operation.
${community}	Specifies the community string of the device.
${mapname}	Specifies the name of the map containing the probed device.
${mapid}	Specifies the internal identifier of the map containing the probed device.

<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
${devicename}	Specifies the device's name taken from first line of the label.

${deviceaddress}	Specifies the network address of the device.
${eval:}	Specifies the eval macro.
${chartable[:fmt]:expr}	Evaluates `expr` and formats the result as a chartable value.
${scalable2:fmt:expr}** **${scalable10:fmt:expr}	Scales large numbers into smaller units for better readability. The values are chartable.
${^stdout}	Specifies any output written to the standard output of a command line script. For more information on the effect of `${^stdout}` on the `reason` string, see below.
${nagios_output}	Parses a Nagios plugin's output for display.

SNMP Probe Variables

A variable name consists of letters, digits, an underscore (_), and must begin with a letter. Variable names are not case-sensitive. A variable name can be referred to in the probe as `$VariableName` or `${VariableName}`. Use the bracketed form for variables and parameters that have one or more spaces in the name.

The variables listed below are available in SNMP Probes (probe-type = customsnmp).

In the `<snmp-device-display>` section of a probe file, the variable name is replaced with its value, rounded to the nearest integer.

For example, if a calculation variable is 3.14159265, using it in the `<display-output>` section results in the value of 3. If the variable has a value of 4.75, it is displayed as 5.

This value is chartable; clicking it creates a new chart and dragging it adds it to an existing chart. If you need to display a non-integer value for the variable, use the `${chartable}` macro as described below.

<snmp-device-display> Section

Variable Name	Variable Description
${devicename}	Specifies the name of the device taken from the first line of the label.
${deviceaddress}	Specifies the network address of the device.
${imserveraddress}	Specifies the network address of the Intermapper server.
${alarmpointlist}	Specifies the list of alarm points.
${eval:expr}	Specifies the eval macro.
${chartable[:fmt]:expr}	Specifies the chartable macro.

${scalable2:fmt:expr}** **${scalable10:fmt:expr}	Scales large numbers into smaller units for better readability. The values are chartable.
${[variablename]:legend}	Specifies the legend of the variable as specified in the <snmp-device-variables> section For more information, see SNMP Probe Variables.

<snmp-device-properties> Section

Variable Name	Variable Description
${ifIndex}	Specifies the interface index.
${ifType}	Specifies the interface type.
${ifDescr}	Specifies the interface description.
${ifAlias}	Specifies the alias of the interface.

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)
${SnmpVersion}	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. The next three macros do not use a colon (:). ${VarbindValue8} returns the value of the eighth varbind item.
${VarbindOID[NNN]}	Trap Field: *NNNth* varbind OID
${VarbindValue [NNN]}	Trap Field: *NNNth* varbind Value
${VarbindType[NNN]}	Trap Field: *NNNth* varbind Type

TCP Probe Variables

The following variables are available in TCP probes (probe-type = tcp-script):

<script> Section

Variable Name	Variable Description
${_ remoteaddress}	Specifies the network address of the remote end of the connection.
${_remoteport}	Specifies the network port number of the remote end of the connection.
${_localaddress}	Specifies the network address of the local end of the connection.
${_localport}	Specifies the network port number of the local end of the connection.
${_gmttime}	Specifies the current time in RFC 822 format.
${_version}	Specifies the version number of the Intermapper program.
${_line:len}	Specifies the text of the last line received, truncated to the specified length.
${_idletimeout}	Specifies the idle timeout for the probe, in seconds.
${_stringtomatch}	Specifies the string we attempted to match in the last EXPT or MTCH command.
${_base64:str}	Encodes the given argument into base64.
${_ cvspassword:str}	Encodes the given argument using the cvs password algorithm.
${_md5:str}	Specifies the MD5 hash of the given argument, in hexadecimal.
${_idleline}	Specifies the line number of the script where you were before the idle handler was invoked.
${_ secsconnected}	Specifies the amount of time, in seconds, the probe spent connected to the other end. This can be 0 if we were immediately disconnected or if the connection failed.
${_length:str}	Specifies the length of the given argument, in bytes.
${_float:num}	Specifies the argument pretty-printed as a floating point number using printf %g.
${_hmac:key:msg}	Specifies the HMAC-MD5 of the message, using the specified key.
${_urlencode:str}	Encodes the specified string used in URLs.

<script-output> Section

Variable Name	Variable Description
${devicename}	Specifies the name of the device taken from first line of the label.
${deviceaddress}	Specifies the network address of the device.
${eval:}	Specifies the eval macro.
${scalable2:fmt:expr}** **${scalable10:fmt:expr}	Scales large numbers into smaller units for better readability. The values are chartable.

Command Line Notifier Variables

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
${message}	Specifies the notifier's message text. (On Microsoft Windows, each double quotation mark (" ") is escaped by \".)
${stripped_ message}	Specifies the notifier's message text with single or double quotation marks (' or ") removed and newlines (\r and \n) replaced by spaces.
${escaped_ message}	Specifies the notifier's message text escaped for URL syntax (for example, 20% for space).
${urlescape:str}	Escapes a string specified in str for use in a URL. Any macros included in str are expanded prior to escaping.

Macros

Intermapper supports several macros that can control and manipulate how variables are displayed, as well as their use in charts.

${chartable} Macro

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

Usage

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

In the output section of a probe file, the `${chartable: ...}` macro creates an underlined value. Click the value to add it to a chart. The macro also controls the field width and number of decimal places. The following parameters are available:

- **min/max** - Use the min and max parameters to enter a range the chart uses for display of the data.

  > **NOTE:** To be parsed correctly, the min and max parameters must be immediately preceded by a plus sign (+) or a minus sign (-).

- **fmt** - 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 pound sign (#) or a quoted printf specifier such as those accepted by the sprintf function.
- **expr** - A variable or an expression (but not a macro). Intermapper evaluates the expression and displays the result according to the formatting string.

Examples

The following is an example of when the `$pi` variable is set to 3.14159265:

```
${chartable: #.## : $pi }:           --> 3.14
${chartable: #.####### : $pi }:      --> 3.1415927
${chartable: "%3d" : $pi }:          --> 3 (with 2 leading spaces)
${chartable: "%9.7f" : $pi}:         --> 3.1415927
${chartable: "%11.7f" : $pi}:        --> 3.1415927 (also with 2
leading spaces)
${chartable: #.####### : $pi*100}:   --> 314.1592650
${chartable: "%9.7f" : $pi*1000}:    --> 3141.5926500
${chartable: "%11.7f" : $pi*10000}: --> 31415.9265000 (no leading
spaces)
```

If the `$speed` variable can be greater than 4 GB:

```
${chartable: +0:+10E9 : "%d" : $speed }:  --> decimal integer
${chartable: +0:+10E9 : "%e" : $speed }:  --> exponential notation
```

If the `$value` variable can be a positive or negative value of more than 4 GB:

```
${chartable: -10E9:+10E9 : "%d" : $value }:  --> decimal integer
${chartable: -10E9:+10E9 : "%e" : $value }:  --> exponential
notation
```

${eval} Macro

Use the `${eval}` macro to compute a value in the output of a script. The `${eval}` macro is available in the following contexts:

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

Usage

The syntax for the `${eval}` macro is as follows:

```
${eval:[expr]}
```

This expression can use any operator or function 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

The following examples 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"-->
```

${variablename:legend} Macro

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

```
<snmp-device-variables>
   ipForwDatagrams, 1.3.6.1.2.1.4.6.0, PER-SECOND, "Forwarded
datagrams"
</snmp-device-variables>
```

the following entry in the `<snmp-device-display>` section shows forwarded datagrams:

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

${scalable10} and ${scalable2} Macros

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

- The `${scalable10}` macro scales by a factor 1000 (for msec, Mbps, and so on).
- The `${scalable2}` macro scales by a factor of 1024 (for KB, GB, TB, and so on).

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
```

${^stdout} Variable and Reason String

In command line probes, a specially formatted output string defines 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 define all or part of the contents of the lower part of the Status window.

When you use `${^stdout}`, the string is not defined. To compensate and to allow the definition of a meaningful reason string, a default is 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 similar to the following:

```
\{ProcessTime0:=100.0,CPU:=1.0,reason:="CPU utilization is below
60%"}
\B5\WMI Top Processes\0P\
   \B4\CPU Utilization:\P0\   $CPU %
      \B4\wmiprvse(3924)\P0\  $ProcessTime0 %
```

Using Persistent Variables

SNMP and command line probes can use persistent variables. Persistent variables retain their value between polls.

SNMP Probe Example

The following is an example of the use of persistent variables in an SNMP probe:

```
<!--
SNMP probe with persistent variables (com.dartware.snmp.persistent)
Custom Probe for Intermapper (http://www.intermapper.com)
Please feel free to use this as a base for further development.
Original version 24 November 2003 by reb.
Updated 29 July 2005 -
Updated 28 Oct 2005 - include display_name so it displays properly
in Intermapper 4.4 -reb
21 Apr 2006 - Changed to test conversion of $variables to a
condition string.
20 Sep 2011 - Updated to talk about variable persistence -reb
18 Oct 2011 - Minor editing polish -reb
-->

<header>
  "type"           = "custom-snmp"
  "package"        =  "com.dartware"
  "probe_name"     =  "snmp.persistent"
  "human_name"     =  "SNMP Persistent Variables"
  "version"        =  "1.1"
  "address_type"   =  "IP,AT"
  "port_number"    =  "161"
  "display_name"   =  "Miscellaneous/Test/SNMP Persistent Variables"
</header>

<snmp-device-properties>
  -- none required
</snmp-device-properties>

-- The <description> contains text that will be displayed in the Set
 Probe window.
-- Describe the probe as much as necessary so that people will
understand what it does and how it works.
<description>
\GB\SNMP Probe with Persistent Variables \P\
Sometimes probes need to compare variables from previous invocations
o the current values. Intermapper SNMP probes can retain variables
from one invocation to the next.
This is done by setting a variable in this invocation to preserve
```

```
the value of the current calculation. This becomes the "old
variable" in the next invocation. The steps are:
- Read the new (current) value into a variable ("XYZ");
- Do the computations with it;
- Save that value in a separate variable ("oldXYZ");
- Re-run the probe. The oldXYZ will still contain its previous
value.
\b\Example Probe\p\
This probe reads ifInOctets for a specified interface and computes
the difference between it and the previous ifInOctets. It also reads
the sysUpTime.0, and computes the time delta between the probe
executions. From these values, the probe computes the traffic rate.
For comparison, the probe also looks at ifInOctets using
Intermapper's standard PER-SECOND calculations.
</description>

-- Parameters are user-settable values that the probe uses for its
comparisons.
-- Specify the default values here. The customer can change them and
they will be retained for each device.
<parameters>
"Interface"  =   "24"
</parameters>

-- SNMP values to be retrieved from the device, and
-- Specify the variable name, its OID, a format (usually DEFAULT)
and a short description.
-- CALCULATION variables are computed from other values already
retrieved from the device.
<snmp-device-variables>
actInOctets,  ifInOctets.$Interface,  PER-SECOND,  "actual inOctets"
inOctets,     ifInOctets.$Interface,  INTEGER,     "current value"
deltaBytes,   $inOctets-$oldInOctets, CALCULATION, ""
curSysUpTime, sysUpTime.0, INTEGER, "current sysuptime"
deltaTime, $curSysUptime - $oldSysUpTime, CALCULATION,  ""
-- Now update the oldXXXX variables with current values for next
time
-- NB: prevInOctets is only needed for display - it is not used in
-- the calculations above
prevInOctets, $oldInOctets, CALCULATION, "from last time"
oldInOctets, $inOctets, CALCULATION,     "old inOctets for next
time"
oldSysUpTime, $curSysUpTime, CALCULATION, "old sysuptime for next
time"
</snmp-device-variables>

-- Specify rules for setting the device into Alarm or Warning state
<snmp-device-thresholds>
```

```
</snmp-device-thresholds>

-- The <snmp-device-display> section specifies the text that will be
appended
-- to the device's Status Window.
<snmp-device-display>
\B5\Persistent SNMP Variables on Interface=$interface\0P\
\4\Old Octets:\0\  $prevInOctets \3g\bytes\0mp\
\4\Cur Octets:\0\  $inOctets \3g\bytes\0mp\
\4\Delta's:\0\  $deltaBytes \3g\bytes\0mp\ in $deltaTime \3g\centi-
seconds \0mp\
\4\Computed Rate:\0\  ${eval:sprintf("%d",($deltaBytes) / $deltaTime
* 100) } \3g\bytes/sec\0mp\
\4\Actual Rate:\0\  ${scalable10: #.### : $actInOctets}
\3g\bytes/sec, using built-in PER-SECOND type\0mp\
</snmp-device-display>
```

Command Line Probe Example

The following example demonstrates the use of persistent variables in a command line probe:

```
<!--
Command-line Return (com.dartware.tool.persistent.txt) Copyright© Fortra, LLC.
All rights reserved.

Test of persistent variables by round-tripping values: pass them into script, get
results back, post to status window. 20 Sep 2011 -reb
1.1  18 Oct 2011 Minor editing -reb
1.2  22 Mar 2021 Convert Python2 to Python3 -Jerry
-->

<header>
    type = "cmd-line"
    package = "com.dartware"
    probe_name = "tool.persistent"
    human_name = "Command Line Persistent Variables"
    version = "1.2"
    address_type = "IP"
    display_name = "Miscellaneous/Test/Command Line Persistent Variables"
</header>

<description>
\GB\Command Line Persistent Variables\p\
This is an example of passing persistent variables into a command-line probe. The
attached Python script takes the variables, updates them, and returns them to be
used for the next iteration.
```

```
The script below processes two variables: \b\$SearchString\p\ and
\b\$numericParam\p\. The script appends an "a" to $SearchString, and adds 1 to
the $numericParam and returns both values. (This is a useless script, created
just to demonstrate use of persistent variables.)

Both variables are uninitialized when the probe is executed the very first time.
The script can detect this startup condition because an uninitialized variable is
passed to the script as the variable's name.
For example, the variable named $SearchString will be passed as a string -
"$SearchString".
The script can detect this value - it's the same name that will be used to return
the new result for the variable - and treat the variable as uninitialized, by
assigning a sensible default value.

The Python script tests the passed-in values to see if they match the expected
name and initializes them accordingly. \i\Note:\p\ This device's address should
be set to \i\localhost\p\.

\i\Note:\p\ These variables could be initialized by setting them in the
<parameter> section, but this exposes a lot of the script's internal variables to
the customer: this is generally not a good design.
</description>

<parameters>
-- no human-editable parameters
</parameters>

<command-line>
-- Unix/OSX: Empty path forces the InterMapper Settings:Tools directory
path=""
    cmd="${PYTHON} persistent.py $numericParam"
    arg=''
    input="${SearchString}"
</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\ Current value of SearchString and numericParam\p0\
Search String: $SearchString
Number: $numericParam
</command-display>
```

```
<tool:persistent.py>
#!/usr/local/imdc/core/python/bin/imdc -OO
# The line above is required to run as an IMDC plugin # persistent.py

# Round-trip the passed-in variables, update them, and return them
# 20 Sep 2011 -reb
import os
import sys
import getopt

try:
     opts, args = getopt.getopt(sys.argv[1:], "")
except getopt.GetoptError as err:
     searchString = "getopt error %d" % (err)

if (args[0] == "$numericParam"): # check to see if the argument is the name of
the parameter
     number = 0 # if so, set it to a good initial value
else:
     number = int(args[0]) # otherwise, convert the string to an integer

# Read the stdin file which contains the search String
f = sys.stdin # open stdin
searchString = f.readline().strip() # get the line & remove leading & trailing
whitespace

if (searchString == "$SearchString"): # check to see if the value is the name of
the parameter
     searchString = "" # if so, set it to a good initial value

retcode=0

searchString = searchString + "a" # add another "a" to the end of the string
number = number+1 # increment the number as well
retstring = "Hunky Dory!"

print("\{ $SearchString := '%s', $numericParam := '%d' } %s" % (searchString,
number, retstring))
sys.exit(retcode)
</tool:persistent.py>
```

SNMP Probes

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

SNMP probes allow you to monitor certain MIB variables that are not tested by Intermapper's built-in probes. These MIB variables can include the CPU utilization of a router, temperature inside the equipment, switch closures, and so on.

SNMP probes are invoked and return the status and condition string for the device being tested. The following is a summary of the operational flow of an 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 polls each interface for probe variables as needed.
3. Intermapper 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 specified threshold), then Intermapper sets the device status as specified in the probe if it is worse than the devices current status.
5. When you click and hold a device, Intermapper processes the relevant *display* section to produce the text for the Status window.

Common Sections of an SNMP Probe

SNMP probes follow the same general format as other probe files. The sections are as follows:

- The `<header>` section of the command-line probe specifies the probe type, name, and other properties that are fundamental to the operation of the probe.
- The `<description>` section specifies the help text that appears in the Set Probe window. Format the description using IMML Intermapper's Markup language.
- The `<parameters>` section defines the fields displayed in the Set Probe window.

Sections Specific to SNMP Probes

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

> **NOTE:**
> See SNMP Trap Probes for information on creating probes that handle SNMP traps.

Each SNMP probe also includes the following:

- **An `<snmp-device-variables>` section** - specifies which MIB variables are collected from the device.
- **An `<snmp-device-thresholds>` section** - specifies how the variables are tested against thresholds to determine the device status.
- **An `<snmp-device-display>` section** - specifies information about the device; its links are displayed in the Status window.
- **The `<snmp-device-properties>` section** - specifies certain aspects of the SNMP queries sent to the device.
- **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.

`<snmp-device-variables>` Section

Use the `<snmp-device-variables>` section to specify the values that are retrieved using an SNMP OID. These values, called *probe variables*, can be compared to thresholds to create alarms, warnings, and so on.

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

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

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
datagrams"
  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 how information is collected and appears in the Status window. Create a `<snmp-device-display>` section with the items to be displayed. For more information, see Customized Status Windows.

Intermapper retrieves MIB variables from a device and tests them against thresholds. The `<snmp-device-variables>` section defines the OIDs of MIB variables. These values are called *probe variables* and can be compared to thresholds to create alarms, warnings, and so on.

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

```
[VariableName], [OID], [Type], [Chart Legend]
```

The following are definitions for these attributes:

- **VariableName** - specifies the name that represents the MIB value in this probe. For more information, see Built-in Variable Reference.
- **OID** - specifies the SNMP Object ID for the probe variable. The OID can be expressed as a string of dotted numbers or as an OID name, if the corresponding MIB is imported into Intermapper. An OID can also be an expression, if the type is CALCULATION (see note below).
- **Type** - specifies how Intermapper displays a value. Type can 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 converted into a numeric value. If you have a string value of 78Fred, the INTEGER value is 78.
  - **Integer64** - values are converted to a numeric value (with a limit of 64-bits). If you have a string value of 78Fred, the INTEGER value is 78.
  - **Hexadecimal** - If the value is a number, it is displayed as a hexadecimal number, preceded by 0x (0xFFFFFFFF). Otherwise, it is represented as a series of hexadecimal characters separated by spaces. For example, 44 61 72 77 69 6E 20 52 69 63 68 61 72.

    > **NOTE:**
    > This type is not chartable.

  - **Hexnumber** - converts a string of hexadecimal digits into a number. For example, a string value of FE is converted to 254.

- **Total-value** - displays the actual value of a counter or gauge, not a computed rate value. This is always an unsigned number.
- **Total64-value** - displays the actual value of a counter or gauge, not a computed rate value. This is always an unsigned number (limited to 64-bits).
- **Per-second** and **Per-minute** - forces Intermapper to compute a rate for the variable by calculating 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. For more information, see Enumerated Values.

  > **NOTE:**
  > This type is not chartable.

- **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 more information on trap variables, see About Custom SNMP Trap Probes.
- **IPADDRESS** - Intermapper displays a 4-byte octet string as an IPv4 address and a 16-byte octet string as an IPv6 address.
- **Format for DEFAULT types** - All SNMP variables have an inherent type (one of the choices in the Type column below.) If a probe variable is declared as DEFAULT, Intermapper displays it according to this table:

Type	Displayed As
Counter32, Counter64	Per-Second
Unsigned32 (Gauge)	Total-Value
Integer	Integer
OctetString	String (if 1st digit printable) Hexadecimal (if 1st digit not printable)
Object ID	String
IPAddress	String
TimeTicks	String

- **Chart-legend** - an optional field that provides a text label for strip charts that incorporate this variable. Chart legends can contain embedded variable names in the form of $VariableName.

NOTE:

- Calculation variables have a slightly different format, as described below.
- See Probe Calculations for a description of the functions and operators that are available in expressions.
- See Using Persistent Variables for information on how to save variable values between device polls.
- Scalar's OIDs must end in .0 according to the SNMP specifications. See SNMP OIDs for a description of allowable OID formats.
- See On-Demand SNMP Tables 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 for more information.

The following is a sample `<snmp-device-variables>` section:

```
<snmp-device-variables>
   --Variable-name  OID ---            TYPE ----    CHART LEGEND ---
---
   sysDescr,        1.3.6.1.2.1.1.1.0, DEFAULT,
   sysLocation,     sysLocation.0,     DEFAULT,
   ipInHdrErrors,   1.3.6.1.2.1.4.4.0, PER-MINUTE,  "IP received
header err"
   ifInOctets1,     ifInOctets.1,      DEFAULT,     "bytes/sec
received on interface 1"
   ifInOctets2,     ifInOctets.$if,    DEFAULT,     "bytes/sec
received on interface ${if}"
   TempF,           ($TempC * 1.8)+32, CALCULATION, "Degrees F"
</snmp-device-variables>
```

- **$sysDescr** - set by retrieving the OID 1.3.6.1.2.1.1.1.0 and using that value. It is displayed as the default format, that is a string.
- **$sysLocation** - set by querying the OID sysLocation.0 (which is equivalent to the numeric 1.3.6.1.2.1.1.6.0). It is displayed as a string. Note that you can use a human-readable SNMP variable name instead of a numeric OID.
- **$ipInHdrErrors** - set by querying the OID 1.3.6.1.2.1.4.4.0, but is displayed as the number of errors per minute.
- **$ifInOctets1** - set by querying the OID ifInOctets.1 (1.3.6.1.2.1.2.2.1.10.1). Note that the final digit is 1, indicating that it is reading the values in row 1 of the table. It is displayed as a number of octets (bytes) per second, since Intermapper's default display format for a counter is per-second.

The following examples are controlled by variables that have been set elsewhere, perhaps manually in the <parameters> section of the probe:

- **$ifInOctets2** - set by evaluating the variable $if, then substituting that value into the OID. If $if is set to 1, then $ifInOctets retrieves ifInOctets.1 and results in the same value as $ifInOctets1. Note that $if is also used in the variable's legend.
- **$TempF** - a calculation variable that is set by evaluating the expression ($TempC * 1.8)+32 where $TempC is set elsewhere.

SNMP Scalar and Table Values

SNMP includes the following values:

- **table** - an element of a table. The variable name (for example, ifInOctets) is the name of a column and the final digit (or digits) is the index of the element. It defines the table row containing the element. Thus ifInOctets.1 is the full OID of the value in the first row of the ifInOctets column.
- **scalar** - a single value. Because of this, you must specify .0 after the name to indicate that it is the only row. For example, sysDescr can be represented as 1.3.6.1.2.1.1.1.0 or sysDescr.0. Both OIDs end in .0.

Using Variables in OIDs and Legends

You can use SNMP variables in OIDs and legends. The following example uses $if as the OID index and displays it in the legend:

```
ifInOctets2, ifInOctets.$if, DEFAULT, "bytes/sec received on
interface ${if}"
```

Calculation Variables

Calculation variables receive the result of an arithmetic expression. After all variables are polled, Intermapper calculates the expression and sets the value of its variable to the result. For example,

```
TempF,  ($TempC * 1.8)+32, CALCULATION, "Degrees F"
```

The TempF variable is set to the value of the expression (10 * sin(0.01 * time()). This provides 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, detailed in the Built-in Variable Reference topic.

Macros

Intermapper supports several macros that can help control the output of variables, as well as their use in charts. For example,

- ${chartable[:fmt]: expr}
- ${variablename:legend}
- ${eval: expr}
- ${scalable10} and ${scalable2}

See Macros section for more information on macros.

Enumerated Values

Many MIBs use integers 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 follows:

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

This means that 1 represents the up condition, 2 represents down, and 3 represents testing.

The type you use when you define the variable affects the result. For example,

- If you define a variable to retrieve this value as INTEGER or DEFAULT, the probe displays the value as a number.
- If you define a variable as a STRING, the probe uses the MIB to find the string representation, and sets the variable (in this case) to the value up, down, or testing.

```
-- If the MIB has been imported, the string is displayed in the
output if the variable is declared as STRING.

  variable1, ifOperStatus.3, STRING, ""

-- The integer value is always used in the output if the variable is
declared as DEFAULT or INTEGER.

  variable2, ifOperStatus.3, DEFAULT, ""
  variable3, ifOperStatus.3, INTEGER, ""
```

If the OID or MIB name are not defined (because the corresponding MIB has not been imported or because of a typo), the probe displays the integer value.

Alternatives to Enumerated Values

If no MIB file is available, you can create a calculation variable to select a string based on the returned numeric value.

Example Two choices

```
-- If you have two choices, use a conditional expression:
   xxxx ? yyyy : zzzz

-- It can be read as:
if xxxx is true then
   return yyyy
otherwise
   return zzzz

-- The variable looks like this:
   xxxxStr, ($xxxx == 0 ? "yyyy" : "zzzz"), CALCULATION,
"replacement string for $xxxx"
```

Example Three or more choices

```
-- Chain the expression:
   aaaa ? bbbb : cccc ? dddd : eeee ? ffff : gggg

-- 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, 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"
```

`<snmp-device-thresholds>` Section

Use the `<snmp-device-thresholds>` section to specify the comparisons 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 it exceeds its current status.

A threshold can be one of the following (they are case-sensitive) and should be presented in this order:

- down
- critical
- alarm
- warning
- okay

Sample `<snmp-device-threshold>` Section

```
<snmp-device-thresholds>
   down: ${ifOperStatus}   =  0 "Device Down"
   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
    okay: 1 = 1 "Everything is OK"
</snmp-device-thresholds>
```

Creating Comparisons

As implied above, comparisons are evaluated in order from top to bottom until a comparison is triggered (result is true). It is important to put Critical comparisons first, followed by Alarm, Warning, and OK.

If the associated status is more severe than the current status of the device, the device uses its status and condition. No further comparisons are made after one has been triggered.

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

Numeric Comparisons

The following are valid numeric comparison operators:

>, >=, <, <=, =, and !=

String Matches

By default, Intermapper performs numeric comparisons.

To compare values as strings:

- Enclose one or both of the operands in double quotation marks (" "). For example, the following comparison:

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

  performs a string comparison because the name is enclosed in double quotation marks.
- Use the =~ and !~ operators to provide partial string matches. They perform contains and does not contain comparisons, respectively.

`<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. 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 property set includes the following:

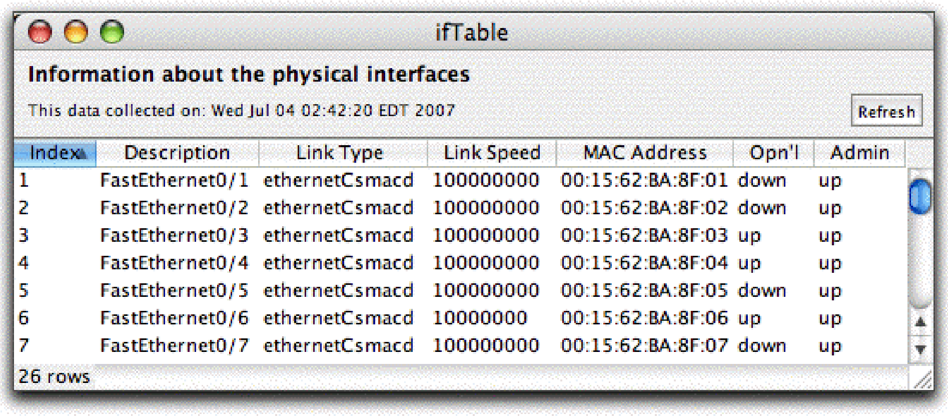
- **nomib2="true"** - Intermapper does not query the sysUptime MIB-2 variable.
- **pdutype="get-request"** - Intermapper uses SNMP Get-Request, instead of Get-Next-Request queries.
- **apcups="false"** - if apcups is set to false, Intermapper does not query the APC-UPS MIB, even for devices that auto-detect as one.
- **maxvars="10"** - 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 that determines which interfaces are 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 the $ifIndex, $ifDescr, $ifType, or $ifAlias variables.

  > **NOTE:**
  > This property does not allow you to hide numbered interfaces.

- **interface_numbered = <expression>** - specifies a search expression for determining which interface is numbered. By default, the ipAddrTable specifies which interface is numbered. This property allows the probe file to override that selection.

`<snmp-device-variables-ondemand>` Section

Many devices store information in SNMP tables. Intermapper can retrieve this tabular information and display the data on-demand, (when requested by a user). When you view a table, Intermapper immediately retrieves data from the device 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 following image shows a sample on-demand window:



ifTable

Information about the physical interfaces

This data collected on: Wed Jul 04 02:42:20 EDT 2007

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	10000000	00:15:62:BA:8F:06	up	up
7	FastEthernet0/7	ethernetCsmacd	100000000	00:15:62:BA:8F:07	down	up

26 rows

On-demand tables are useful for investigating when you suspect there might be a problem with a device. You can create on-demand tables to view a routing table, ARP table, or other statistics that are stored in tables.

Background on SNMP Tables

The SNMP protocol provides access to the following variable types:

- **Scalar variables** - contain single values such as strings (that can represent system description or a firmware version), integers (number of interfaces), counters (number of errors), gauges (CPU temperature and memory utilization), and so on.
- **Table variables** - contain information about similar entities within a device. These entities can be interfaces to a router or switch, users associated with a wireless access point, virtual machines on a server, and so on. Each entity's information is represented by a row and the columns are variables (which are themselves scalars) that contain 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 provides information about a device's interfaces. An outline of ifTable is as follows:

```
+ 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...
```

The *ifTable* is composed of a sequence of *ifEntry*s that 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 above example 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 when data is retrieved, and the Refresh button to refresh the 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)
```

represents 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 .3.

Table Syntax

An on-demand table in a custom SNMP probe mirrors the outline above. Its definition contains a sequence of lines of comma-separated values defining the variables of one or more tables. For example, the following is an on-demand ifTable:

```
<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,      "Opn'l"
        <!-- using column name from MIB -->
  ifTable/ifAdminStatus, ifAdminStatus,       DEFAULT,     "Admin"
        <!-- using column name from MIB -->
</snmp-device-variables-ondemand>
```

> **NOTE:**
> The `<snmp-device-variables-ondemand>` section is limited to 50 queries.

The remainder of the table is composed of the following comma-separated lines that describe each variable:

- The first line creates a table. The first field is the table name that can represent the table in the probe file. The second field should be called .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 `<snmp-device-variables>` format. See their respective page for more information. The first column contains the table name, a forward slash (/), and the name of the column.
- The next four lines define variables (ifIndex, ifDescr, ifType, and ifSpeed) that define table columns. They are defined using the numeric OID that represents the column of those values.
- The final three lines define ifPhysAddress, ifOperStatus, and ifAdminStatus. They are defined using the column name from the MIB. This is equivalent to the full numeric OID.

These tables are available as the **SNMP/Table Viewer** probe that is built into Intermapper. In addition, the probe file is available on the Table Viewer page of this manual.

Augmenting Tables

Certain MIBs define a table that augments another table. This means that the augmenting table uses the same index variables as another table. Since the index variables are the same, this similar to adding columns to an existing table.

For example, in the IF-MIB, the ifXTable augments the 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. For example,

```
<snmp-device-variables-ondemand>
  ifXTable,               .1,                    TABLE,
"Extended ifTable"
  ifXTable/ifIndex,       IF-MIB::ifIndex,       DEFAULT,
"Interface 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. They all can be displayed in the same on-demand window.

Index-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 cannot be queried directly.

You can derive the values of these columns from the index itself, even from columns that are not accessible. The `oid[a:b]` notation fetches the OID `oid` and compute the value from the index. For example,

```
oid[a:b] - remove the subid's for "oid" then start with the a'th
subid and collect b subids.
oid[a:]  - remove the subid's for "oid" then start with the a'th
subid and collect the remaining subids
```

The following is an example of retrieving the four columns of ipNetToMediaTable. Note that the table is named ARPTable, although the OID is set to 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,
HEXADECIMAL,"MAC Address"
   ARPTable/ipNetToMediaType,        ipNetToMediaType,
STRING,     "Type"
</snmp-device-variables-ondemand>
```

The `ipNetToMediaTable` uses the following index values:

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

The full OID used to retrieve a value from the table is its prefix (for example, `ipNetToMediaType` is `1.3.6.1.2.1.4.22.1.4`, followed by a single subid for `ipNetToMediaIndex` followed by the four subids of `ipNetToMediaNetAddress`).

When Intermapper displays the table, it retrieves `ipNetToMediaType`, removes the prefix, starts at position 0 of the remainder and uses one subid for the `ipNetToMediaIfIndex`, and then starts at position 1 and takes the next four subids for the value of `ipNetToMediaNetAddress`.

Calculations Within On-Demand Tables

Intermapper provides the ability to use calculations in on-demand tables. This is useful for making calculations from values within the same row of the table. The calculations can 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) is 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 display a hyphen (-).

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

Displaying On-Demand Tables

After you define a TABLE variable in the on-demand section of the probe, you can specify that the status window displays a link to the on-demand window. To do this, add the variable name to the <snmp-device-display> section of the probe. For example,

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

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

To replace the default table name displayed with your own text, you can specify alternate text in the following expanded variable format:

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

Limitations

- On-demand variables must be in table form with a forward slash (/) 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 declare no more than 50 variables in the <snmp-device-variables-ondemand> section or the query will not work.

`<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 Status window. Probe variables are replaced with their values in the Status window text.

The default font for the Status window text is monospaced, so alignment of text is straightforward. You can change the appearance of the text in the Status window using IMML, Intermapper's Markup Language.

The following is an example `<snmp-device-display>` section. Note that the variables are replaced with the values retrieved from the device and formatting is controlled by IMML.

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

Using Disclosure Widgets

A disclosure widget, also called a disclosure control, is a user interface element that allows you to expand or collapse text in a window.

The basic syntax of a disclosure control area is as follows:

```
\#hide:[disclosureblock name]\ Title for Disclosure Block \#begin:
[disclosureblock name]\
  First line of disclosureblock
  Second line of disclosureblock
  Third line of disclosureblock
  Fourth line of disclosureblock
\#end:[disclosureblock name]\
```

Use the #hide and #show comments to specify the default state of the block.

You can nest disclosure controls. For example,

```
<!--
  Testing Disclosure Widgets(com.dartware.reb.expander.txt)
  Probe for Intermapper (http://www.intermapper.com)
  Please feel free to use this as a base for further development.

  Original version - 6 Aug 2010 -reb
-->

<header>
  "type" = "custom-snmp"
  "package" = "com.dartware"
  "probe_name" = "reb.expander_control"
  "human_name" = "Test Expander Control"
  "version" = "1.0"
  "address_type" = "IP,AT"
  "port_number" = "161"
  display_name = "Miscellaneous/Test/Test Expander Controls"
</header>

<snmp-device-properties>
  -- none required
</snmp-device-properties>

<description>
  \GB\Testing Disclosure Widgets\P\

  This probes is for testing out the disclosure widget ("expander
control") feature
  in Status Windows.
</description>

<parameters>
  -- none
</parameters>

<snmp-device-variables>
  -- none
</snmp-device-variables>

<snmp-device-thresholds>
  -- none
</snmp-device-thresholds>

<snmp-device-display>
  -- The <snmp-device-display> section specifies the text that will
be appended
  -- to the device's Staus Window.
  \B5\Displaying Expander_ Controls\0P\

  \#hide:expander_1\ Title for Expander_1 (initially hidden)
\#begin:expander_1\
  First line of Expander_1
  Second line of Expander_1
  Third line of Expander_1
  Fourth line of Expander_1
  \#end:expander_1\
  \#show:expander_2\ Title for Expander_2 (initially shown)
\#begin:expander_2\
  First line of Expander_2
  Second line of Expander_2
  Third line of Expander_2
  Fourth line of Expander_2
  \#hide:expander_3\ Expander_3 nested within Expander_2
\#begin:expander_3\
  First line of Expander_3
  Second line of Expander_3
  \#end:expander_3\
  \#end:expander_2\
  \#hide:expander_4\ Title for Expander_4 (controlled by Expander_1)
\#begin:expander_1\
  First line of Expander_4
  Second line of Expander_4
  Third line of Expander_4
  Fourth line of Expander_4
  \#end:expander_1\

</snmp-device-display>
```

The `<snmp-device-alarmpoints>` Section

Intermapper can monitor multiple conditions within a single device (for example, a single piece of hardware) and provide 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 condition 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.

> **NOTE:**
> Alarm Point probes are typically customized for a particular purpose, which specifies both the conditions under which alerts are sent and the notifiers to which they are sent. To send an alert using a notifier, do the following:
>
> - Edit the `<snmp-device-alarmpoints>` section of probe containing the alarm points as described in Alarm Point Format.
> - Enter the name of the notifier in the `<snmp-device-notifiers>` section as described in Alarm Point Notifiers.

In probes that contain notifiers, all alarm points are sent to the Default Sounds notifier by default.

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

Alarm points have the following 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 operation is significantly affected.
Critical	Solid Red	Device operation is seriously degraded.
Down	Blinking red	Device is unresponsive, actual state is not known.

When an alarm point changes from one severity to another, the following occurs:

- The new condition is logged in the log file.
- A notification is sent using the existing Intermapper notifiers, including sounds, email, paging modem or SNPP, and running scripts.

Alarm Points - What Users See

Intermapper displays devices with alarm points similarly to how it shows regular devices. A device icon is colored according to the most serious condition of 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 is worse than the orange Alarm or Major severity.

A device 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 a solid red to match the Critical color.

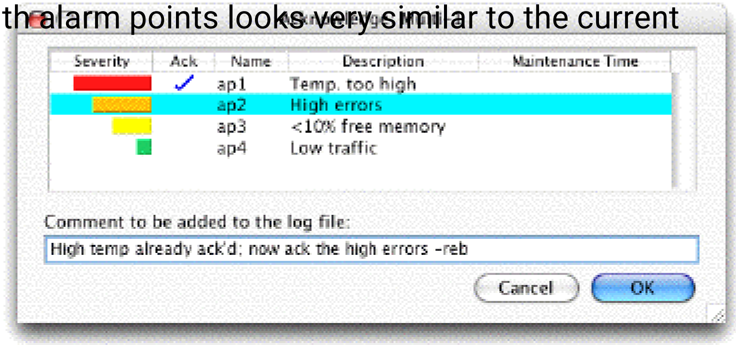
Acknowledgments

Acknowledging an alarm allows the operator to indicate that they are aware of and are working on a problem. An acknowledgment blocks further notifications for that alarm (indicated by a blue icon) and shows that, although the problem remains, someone is working on 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) new alarms appear as yellow, orange, or red.

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 color set to its most serious non-acknowledged alarm point. When all alarm points are acknowledged, the device icon turns blue.

The Acknowledge window for devices with alarm points looks very similar to the current Acknowledge window, but with the following differences:



Severity	Ack	Name	Description	Maintenance Time
🟥	✓	ap1	Temp. too high	
🟧		ap2	High errors	
🟨		ap3	<10% free memory	
🟩		ap4	Low traffic	

Comment to be added to the log file: High temp already ack'd: now ack the high errors -reb

- When acknowledging a device, the Acknowledge window displays a list of the alarm points, sorted in severity order.
- The operator can select one, many, or all alarm points of a device and acknowledge them.
- Selecting multiple devices and acknowledging them at once acknowledges the alarm point of each device in that one action.
- The Acknowledge window contains a text field where you can 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, email, modem paging, SNPP, running a script, and so on) 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 section.

Log File Messages

Intermapper logs messages in the event log file for individual alarm point actions. The entries are written on a change of severity, for notifications, acknowledgements, or for maintenance mode changes. The lines include tab-delimited fields in the following order:

- **Date-time** - the date and time the entry was logged in the log file.
- **Severity** - a four to 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. For example, 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. For more information, see Alarm Point Format.

Alarm Point File Format

Alarm points are defined in the <snmp-device-alarmpoints> section that contains several lines of the format. For example,

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

For example,

```
<snmp-device-alarmpoints>

-- Name:     Severity (Condition-to-Test)          Condition-String
=> Notifer-List
   SiteTemp: critical ($Temp > $CriticalHighTemp) "VERY_HIGH_TEMP"
=> PageFred
   SiteTemp:    major ($Temp > $MajorHighTemp)    "HIGH_TEMP"
=> PageFred

</snmp-device-alarmpoints>
```

Each entry has the following fields:

- **Name** - the name of the alarm point. If multiple lines in this section contain the same Name, they are treated as the different thresholds for the same alarm point.

- **Severity** - the resulting severity of the given point test. Valid options are critical, major, minor, and clear.
- **Condition-to-test** - an expression that evaluates to a Boolean result. For example, ($Temp > $CriticalHighTemp). You can use variables from the `<snmp-device-variables>` section or the `<parameters> more information` on valid expressions. In the example above, $Temp is a variable read from the SNMP device. $CriticalHighTemp and $MajorHighTemp are parameters set by the user.
- **Condition-String** - a string that describes the resulting status if the Condition-to-test is true.
- **Notifier-list** - an optional, comma-delimited list of notifier names. Notifier names are mapped to actual Intermapper Notifier Names in the `<snmp-device-notifiers>` section.

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

Macros

Intermapper supports the following macros that display alarm point information:

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

  to generate the CRIT, MAJOR, MINOR, or CLEAR strings with the appropriate color.
- **${alarmpointname:condition}** - shows the alarm point condition string, as defined in the `<snmp-device-alarmpoints>` section. For example, to show the SiteTemp alarm point condition above, enter the following:
  `${SiteTemp:condition}`

  to generate the VERY_HIGH_TEMP string. This string might contain markup as described on the Probe File Description page.

AlarmPoint Facilities

Intermapper provides the following 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.

The AlarmPointList contains the following information:

- alarm points that are **currently in alarm** and 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.)

For the recently in alarm qualifier, a link in the device's Status window allows you to reset the alarm point list and remove the cleared alarm points.

Alarm point names that begin with underscores (_) 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 (where the state is currently unknown), 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 set to Clear.

  If the new information updates an existing alarm point, then the severity is updated in the AlarmPointList.
- **Resetting the AlarmPointList:** - if you click an AlarmPointList's ***Reset*** link in the Status window, all Clear alarm points are removed from the AlarmPointList for the device. 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 the 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 re-synchronizing Intermapper's notion of a device state with its actual state. A single trap might indicate that the device status is unknown. That is, the device went down, but came back up. 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 its alarm points are set to Unknown. These alarm points are never listed in the AlarmPointList.

Usage

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

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

Facilities to Speed-Up Rule Evaluation

The Break severity aborts the processing of the remaining rules of the probe if its expression is true. These alarm points are never listed in the AlarmPointList.

Usage

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

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

Sample Probe

The following is a sample probe that uses all of 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) are underscore alarm points, the states are not shown unless the alarm point reaches a non-clear state (minor, major, or critical).

> **NOTE:**
> Setting the $trapVar variable to 4 does not reset the alarm point state to critical since there is a break rule right before the rule that invokes the 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 ($trapVar == "2") "trapVar - clear" =>
NotifySomeone
    _trapVarAP: reset ($trapVar == "3") "trapVar - reset" =>
NotifySomeone
```

```
    _trapVarAP: minor (${trapVar} == "4") "trapVar - minor" =>
NotifySomeone
    _trapVarAP: critical (${trapVar} == "5") "trapVar - major" =>
NotifySomeone
    _trapVarAP: break ($trapVar == "6") "trapVar - break" =>
NotifySomeone
    _trapVarAP: critical (${trapVar} >= "6") "trapVar - critical" =>
NotifySomeone

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

    trapVarAP3: clear ($trapVar3 == "2") "trapVar3 alarm" =>
NotifySomeone
    trapVarAP3: reset ($trapVar3 == "3") "trapVar3 alarm" =>
NotifySomeone
    trapVarAP3: break ($trapVar3 == "4") "trapVar4  alarm" =>
NotifySomeone
    trapVarAP3: critical (${trapVar3} >= "4") "trapvar alarm" =>
NotifySomeone
</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\

    $alarmpointlist
</snmp-device-display>
```

Alarm Point Notifier List

The `<snmp-device-notifiers>` section contains the following lines:

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

where NotifierName is an identifier that can be used in the Notifier List section of the alarm points section and 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 following format (quotation marks (" ") are required):

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

Delay and repeat are specified in seconds. If delay and repeat are omitted, the value is set to 0 by default. The count is the number of times the notification is repeated. If repeat is set to 0, the count is ignored because there is no repeat. If repeat is not set to 0 and the count is omitted, the count is infinite and repeats forever.

```
<snmp-device-alarmpoints>
   -- Name:     Severity (Condition-to-Test)       Condition-String
=> Notifer-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 further 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, and so on. You can compare these computations to thresholds to set device status and indicate problems.

Expression Syntax

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 and the value for FALSE is 0.0.
- Supports short-circuit logical operators and, or, not, and &&,||,!,.
- Supports variables and functions from a symbol table. Variables can use $var syntax or ${var} syntax. Persistent variables retain values between polls. For more information, see Using Persistent Variables.
- 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 quotation marks (" ") is 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.

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(x),round(x,y) - Rounds x to the nearest integer
- trunc(x) - Removes all digits after the decimal point. For example, 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. For example, log(100, 10) = 2
- time() - Time in seconds since 1 January 1970 UTC

Built-In String Functions

- defined(str) - takes a string argument and returns a non-zero value (1) if the variable name specified in the input string is defined.
- strfind(strToBeSearched STRING, substrToFind STRING) - case sensitive match returns the position of the first matching substring.

- strifind(strToBeSearched STRING, substrToFind STRING) - case insensitive match returns the position of the first matching substring.
- strlen(str) - returns the length in bytes of the string str or the combined length of all string arguments.
- sprintf(fmt, ...) - returns formatted string using format specifier fmt. Format specifier fmt contains format codes that begin with %.
- strftime(fmt, [secs]) - returns formatted date/time string using format specifier 'fmt'.
- 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.
- subid(oid, start, length) - obtains the specified length sub-OIDs from a given OID string, starting from index start (the index starts from 0).
- substr(str, offset, len)
- unpack(binary str, fmt)
- Regular Expressions - See below for an example that uses Regular Expressions.

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 (meaning 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***).

Example

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

strfind

FUNCTION strfind(strToBeSearched:STRING, substrToFind:STRING):INTEGER

Case-sensitive string match returns an integer representing the position of the first occurrence of a substring in the string. If the substring is not found, the function returns a value of -1.

Example

```
strfind( "Ethernet Interface", "int")

returns -1 (did not find the substring)
```

strifind

FUNCTION strifind(strToBeSearched:STRING, substrToFind:STRING): INTEGER

Case-insensitive string match returns an integer representing the position of the first occurrence of a substring in the string. If the substring is not found, the function returns a value of -1.

Example

```
strifind( "Ethernet Interface", "int")

returns 9 (found the substring at position 9)
```

strlen

FUNCTION strlen(str[, ...]:STRING):INTEGER

Returns the length of the string ***str*** in bytes.
Returns the combined length of all string arguments in bytes.

Example

```
strlen( "Fortra" )  -->  6
strlen( "Fortra", "2000" ) -->  10
```

sprintf

FUNCTION sprintf(fmt:STRING, ...):STRING

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 colons (:)
- 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 following is the general specification for a format code:

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

String Formatting

For string data using percent signs (%), 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( "%12s", "Fortra" )
   Results in " Fortra"
sprintf( "%s", "Fortra" )
   Results in "Fortra"
```

The default alignment is to the right; so padding is added to the beginning of the string. To left align the output of percent signs (%), include a hyphen (-) immediately following the percent sign (%). For example,

```
sprintf( "%-12s", "Fortra" )
   Results in "Fortra "
```

```
sprintf( "%-10.4s", "Fortra" )
   Results in "Help      "
```

Integer Formatting

Integers format similar to strings, except the <precision> field specifies the maximum field width. You can enforce this by padding integers with zeroes. For example,

```
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 [-]ddd.ddd format and %e uses the [-]d.ddde+-dd format. For example,

```
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. For example,

```
sprintf( "%a", "\x01\x02\x03" )
   Results in "01:02:03"
sprintf( "%a", "Fortra" )
   Results in "48:65:6C:70:53:79:73:74:65:6D:73"
```

strftime

FUNCTION strftime(fmt [, time])

Returns formatted date/time string using format specifier fmt. Format specifier fmt contains format codes that begin with a percent sign (%). 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 (Mon)
- A - Full weekday name (Monday)
- b - Abbreviated month name (Mar)
- B - Full month name (March)
- c - Formatted date and time (Mon Mar 09 10:25:22 2007)
- d - Day of month (01-31)
- H - Hour (00-23)
- I - Hour (01-12)
- j - Day of the year (001-366)
- m - Month number (01-12)
- M - Minute (00-59)
- p - AM or PM
- S - Second number (00-61)
- s - Number of seconds since the Epoch (1970-01-01 00:00:00 +0000 (UTC)).
- U - Week of the year (00-53). First Sunday is day 1 of week 1.
- w - Weekday (0-6). Sunday is 0.
- W - Week of the year (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 (2020)
- z - The +hhmm or -hhmm numeric timezone (the hour and minute offset from UTC) for the Intermapper server.
- % - Prints a percent sign when preceded by a percent sign (%)

The strftime function is implemented using the identically named function in the underlying system. Other format codes can 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 amount of time, in seconds, since UTC 1970 is represented by the specified date/time string, as interpreted using the specified format code. You can use this function 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)

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

Example

```
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;

Extracts and returns a substring out of **str**. 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 returns **str**. For more information, see Perl substr.

Example

```
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

Takes a string **str** representing a data value and converts it into a scalar value. The format string specifies the type of value to be unpacked. For more information, 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 0 bytes at the end. For example,

  ```
  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, the target platform's byte order (little endian on Microsoft Windows, big endian on Mac) is used.
- If a format specifier is not supplied, the function returns 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( "F", "c")                                  -->  70 (decimal
ASCII value)
unpack( "F", "H")                                  -->  15 (Hex converted to
decimal value)
unpack( "48656C7053797374656D732C20496E632E", "#H") ->   Fortra,
Inc.
```

> **NOTE:**
> - It is difficult to create examples that input unprintable characters. Refer to the Perl documentation for more information about `unpack()`.
> - 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. In the example below, a customer had equipment that returned the following information in sysDescr.0:

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

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>
```

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. Note the following:
   - 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. [^,] matches any single character that is not a comma (,). Adding a plus sign (+) forms a pattern that matches multiple non-comma characters.
   - Parentheses around a pattern memorize a string. Each pair of parenthesis matches a string whose value is placed in variables numbered ${1}, ${2}, ${3}, and so on.
   - Semicolons (;) followed by ${1} indicate that the entire CALCULATION should return the value of ${1} as a string.
   - The $firmware variable is assigned the value of ${1}.
3. Assign the $frequency variable with the result of the second match (${2}).
4. Assign the $antenna variable with the result of the third match (${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 internet. For example, see Perl Regular Expression Tutorial.

Specifying SNMP OIDs in Custom Probes

Intermapper supports two kinds of OIDs: numeric and symbolic. Symbolic OIDs are available when a MIB is imported into Intermapper.

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

Numeric OIDs

Numeric OIDs 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 OIDs that begin with a period; all numeric OIDs are considered absolute.

Errors in numeric OIDs are reported by the system to the event log when the error is in a custom probe. Error messages use the following format:

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

Symbolic OIDs

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. The following are types of symbolic OIDs:

1. Simple symbols specify a starting symbol and zero or more trailing subids.
2. Relative symbols specify a starting symbol and one of 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, because the same symbol name is defined differently in two separate MIB files. Fortra recommends using 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 OIDs are reported by the system to the event log when the error is in a custom probe. Error messages use the following format:

```
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.

Use the following to enter OIDs:

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

and use the following to create an SNMP variable:

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

Limitations of Symbolic OIDs

1. Symbolic OIDs only work if the necessary MIB file is loaded in Intermapper. If Intermapper cannot resolve the symbolic OID using a MIB file, this is considered a syntax error in the symbolic OID. Currently, there is no way to bundle a MIB file with a probe as one file.
2. Two or more MIB files might define the same symbol. When this happens, Intermapper picks the wrong definition. You can avoid this by using the scoped OID format.

Get-Next OID Expressions

Intermapper has a special syntax for Get-Next style OIDs when a plus sign (+) is added 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, if you want to query the value of ifInOctets for the first interface, but you cannot assume the ifIndex of the first interface is 1, specify the OID as follows:

```
ifInOctets+
```

To retrieve the ifInOctets for the interface where the ifIndex follows 13, specify the OID with a plus sign (+). For example,

```
ifInOctets.13+
```

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

> **NOTE:**
> Get-Next OID expressions do not work with custom SNMP probes that specify Get-Request queries.

Trap-Conditional OID Expressions

Trap-conditional OID expressions allow you to assign a variable when it occurs in the varbind list of a certain trap. For example, you can set the value of your probe's sysUpTimeCrashed variable to the sysUpTime.0 variable included in the varbind list of a systemCrashed trap. However, you do not 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, 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 version 4.4, the legacy format for trap OIDs is a numeric OID followed by an OID. For example,

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

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

Index-Derived OID Expressions

When querying tables from SNMP devices, you can assign the value of a variable from a row OID index. This technique works even if the values used to index the row are not accessible.

For more information, see Index-Derived Variables in the `<snmp-device-variables-ondemand>` Section topic.

SNMP Probe Example

```
<!--
  Single OID Custom Probe (com.dartware.snmp.oidsingle.txt)
    Custom Probe for Intermapper (http://www.intermapper.com)
    Please feel free to use this as a base for further development.

  10 May 2007 Cloned from High Threshold probe -reb
   3 Jul 2007 Changed probe name to Single OID Viewer -reb
   4 Sep 2012 Added a datasets section -jpd

  You can read the Developer Guide to learn more about Intermapper
Probes. It's at:
    http://intermapper.com/go.php?to=intermapper.devguide
```

```
-->

<header>
  "type"        = "custom-snmp"
  "package"     = "com.dartware"
  "probe_name"  = "snmp.oidsingle"
  "human_name"  = "SNMP - Single OID Viewer"
  "version"     = "1.4"
  "address_type"= "IP,AT"
  "port_number" = "161"
  "display_name"= "SNMP/Single OID Viewer"
  "flags"       = "Minimal"
</header>

<snmp-device-properties>
  nomib2  = "true"
  pdutype = "get-request"
</snmp-device-properties>

<description>
\GB\Single OID Viewer\P\

This probe retrieves a single SNMP MIB variable and displays it in
the device's Status Window.

\ib\Variable\p\ specifies the MIB name or OID for the value to
retrieve. If you have imported the MIB for this device, you may
enter the symbolic name for this value, otherwise, simply enter its
OID here.

\bi\Legend\p\ is a text string used to identify the variable in the
status window and any strip charts. If left blank, the variable's
name or OID will be used.

\bi\Units\p\ is a text string that will be displayed next to the
value in the Status Window. You can use it for the unit of measure
(packets/sec, degrees, etc.)

\bi\Tag\p\ is a short text string that identifies a particular class
of dataset. Tags will be used to correlate different variables from
different probes that describe the same thing, such as CPU% or
temperature.
</description>

-- Parameters are user-settable values that the probe uses for its
comparisons.
```

```
-- Specify the default values here. The customer can change them and
they will be retained for each device.

<parameters>
  "Variable"  = "ifNumber.0"
  "Legend"    = ""
  "Units"     = ""
  "Tag"       = "exampletag"
</parameters>

-- SNMP values to be retrieved from the device, and
-- Specify the variable name, its OID, a format (usually DEFAULT)
and a short description.
-- CALCULATION variables are computed from other values already
retrieved from the device.

<snmp-device-variables>

    theLegend,  ($Legend!="" ? "$Legend" : "$Variable"),
CALCULATION, "Legend/OID"
    theOID,      $Variable ,    DEFAULT,  "$theLegend"

</snmp-device-variables>

-- The <snmp-device-display> section specifies the text that will be
appended
-- to the device's Staus Window.

<snmp-device-display>
\B5\  $theLegend:\0P\  $theOID \3G\$Units\mp0\
</snmp-device-display>

<datasets>
  $theOID, "$Tag", "$Units", "false", "$Legend"
</datasets>
```

SNMP Trap Probes

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

A trap is an unsolicited packet sent from a device to Intermapper (or another SNMP management console). Traps generally contain one or more data values that provide information about the state of the device.

When a trap arrives, Intermapper determines which device 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 where the IP addresses match. For example, if a device with the IP address is on two maps, or is present twice on the same map, each device receives a copy of the trap.

Intermapper then parses out the values from the trap and assigns them to trap variables for the remainder of the probe. Intermapper re-evaluates the expressions in the probe and sets the device status appropriately. If a trap variable is not set by an incoming trap, expressions containing that variable are not evaluated. See The <snmp-device-variables> Section for Traps for more information on 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 state of the device. If another trap arrives before the final response of this new poll has returned, Intermapper completes the current poll and initiates another round of polling to obtain 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. Fortra recommends that you do not rely completely on traps for monitoring device health. There is no substitute for regular polling.

For information on how to retrieve and display trap contents, see Example Trap Probe.

`<snmp-device-variables>` Section For Traps

A **Trap Variable** is a variable defined in a custom probe file where the value is received from a trap. Intermapper includes the following 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 is never polled, meaning that 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. For example,

- **$GenericTrap** - the GenericTrap field in the trap (SNMPv1). This field can be one of the following values:
  0 - coldStart;
  1 - warmStart;
  2 - linkDown;
  3 - linkUp;
  4 - authenticationFailure;
  5 - egpNeighborLoss;
  6 - An enterprise-specific value.
- **$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 of 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 $AgentAddress when the sender is forwarding traps for the agent.
- **$SnmpVersion** - 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 the following 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 is limited 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 of this document describes the file format. For 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 $InterMapperTimeStamp variable 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 do not have values set by an incoming trap are left undefined.

For a full example trap file, see Example Trap File.

The following illustrates how several trap variables can 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,      $SnmpVersion,     TRAPVARIABLE,    "SNMP
Version"
  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 change the display to a certain format. These variables are equivalent to their non-trapvariable counterparts. For descriptions of the formats, see Probe Variables.
>
> - TRAPVARIABLE-TOTAL-VALUE
> - TRAPVARIABLE-PER-SECOND
> - TRAPVARIABLE-PER-MINUTE
> - TRAPVARIABLE-STRING*
>
>   This is a string and cannot be charted.
> - TRAPVARIABLE-INTEGER
> - TRAPVARIABLE-HEXADECIMAL*
>
>   This is a string and cannot be charted.
> - TRAPVARIABLE-HEXNUMBER
> - TRAPVARIABLE-DOUBLE

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 are instances where a vendor's traps contained VarBind List entries with the same name. If this occurs, you need to obtain their values by position. Below are examples of accessing VarBind List entries by name and by position.

With 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 the deviceUnitID, deviceName, and deviceStatus variables. (Intermapper imported a LIVEWAVE MIB that defines these OIDs.)

The following 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. You can use the probe variables in the following way:

```
<snmp-device-thresholds>
   critical: deviceStatus == 3 "Problem with $deviceUnitID
$deviceName: 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. For example,

```
<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>
```

`<snmp-device-display>` Section for Traps

You can use the `<snmp-device-display>` section to format the device's Status window in 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 in the event log file when the trap is received. There are two forms: Short and Verbose. (The format is controlled by the **Verbose Trap Logging** check box 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 parenthesis.
- **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 quotation marks.
  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 OIDs value in quotation marks.
- **Address** - [(via 192.168.1.233)] The address of the relaying system, if present.

The verbose format shows all information sent with the trap.

Trap Viewer Probe Example

The following example shows how traps are handled:

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

     Copyright© Fortra, 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
description -reb
     1 Jun 2007 Changed probe name to "Trap Viewer"; tweaked
description;
                    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
contents 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
```

```
making 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/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\

a) Add a device to a map with the IP address \i\192.168.56.78\p\
b) Set it to use this probe
c) 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\
a) Add a device to the map with an IP address of \i\localhost\p\
b) Set it to use this probe
c) 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,
"Timestamp"
  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,
"Condition 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,
"Timestamp"
  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
Version"
  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) \P0M\
  \4\   SpecificTrap:\0\  $specificTrapVar \3IG\(v1 only) \P0M\
  \4\     Enterprise:\0\  $enterpriseVar \3IG\(v1 only) \P0M\
  \4\        TrapOID:\0\  $trapOIDVar \3IG\(v2c only) \P0M\
  \4\    SnmpVersion:\0\  $snmpVersionVar \3IG\(0=SNMPv1; 1=SNMPv2c)
\P0M\
  \4\   VarbindCount:\0\  $varbindCountVar \3IG\(total number of
Varbinds) \P0M\

\B5\Varbind List Items parsed by OID\0P\
  \4\       TimeStamp:\0\  $trapTimeStamp \3IG\ \P0M\
  \4\   Device Status:\0\  $deviceStatus \3IG\ \P0M\
  \4\      Device DNS:\0\  $deviceDNS \3IG\  \P0M\
  \4\Condition String:\0\  $deviceCondition \3IG\  \P0M\
  \4\Trap Source Adrs:\0\  $TrapSourceAdrs \3IG\  \P0M\
  \4\      Probe Type:\0\  $ProbeType \3IG\  \P0M\

\B5\Varbind List Items by Position\0P\ \3IG\(Varbind Value / Varbind
Type / Varbind OID) \P0M\
  \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>
```

Dartware MIB

Fortra registered the Enterprise 6306 for its own SNMP variables. The following shows the Dartware MIB in ASN.1 notation:

```
-- ******************************************************************
-- DARTWARE-MIB for Intermapper and other products
--
-- May 2007
--
-- Copyright© Fortra, 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
intermapperProbeType."

        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
NOTIFICATION-TYPE macro:
    --
    --
    -- intermapperTrap TRAP-TYPE
    --      ENTERPRISE      dartware
    --      VARIABLES       { intermapperTimestamp,
intermapperMessage,
    --                        intermapperDeviceName,
intermapperCondition }
    --      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
notification option."
        ::= { intermapperNotifications 1 }


END
```

TCP Probes

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

TCP probes connect to the specified device and port and execute a script that sends and receives data from the device. Intermapper examines the responses and sets the device status and condition based on the results.

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

As another example, the TCP Example shows another TCP-based probe that connects to a device. It 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. This probe can be used for making your own probes.

Common Sections of TCP Probes

Each TCP probe uses the following general format that is used by other probe files:

- The `<header>` section specifies the probe type, name, and other properties fundamental to the operation of the probe.
- The `<description>` section specifies the help text displayed in the Set Probe window. You can format the description using IMML, Intermapper's Markup language.
- The `<parameters>` section defines the fields displayed in the Set Probe window.

Sections Specific to TCP Probes

Each TCP probe also includes the following:

- The `<script>` section of a TCP probe defines a sequence of commands the probe uses to interact with and query a device and specifies how 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 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 set the status of the device based on those responses.

You can use *regular expressions* and *comparisons* to parse out information from the responses.

Overall Process

There is an annotated FTP probe in the Developer Guide. This provides an overview of the script language and shows how it connects to and logs into an FTP server, how a script can respond to error conditions, and how to set the device 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, the testval variable contains 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 matches are stored in the ${1} variable. If you have several capturing groups, they are stored in ${2}, ${3}, and so on.

For more information, see the Regular Expressions examples in Probe Calculations.

Calculations in Scripts

You can use the EVAL command to perform calculations in a TCP script. The argument is an expression (in quotation marks) that is 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)
```

sets 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 compare strings or numeric (integer or floating point) values. To do this, write an EVAL statement that compares the two values and sets the result in a new variable. If the comparison is true, the resulting variable is set to 1, otherwise it is set to 0.

The following 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 Eval Macro section in 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 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 enclosed in quotation marks. To convert a string to a numeric value, add a pound sign (#) before the parameter. For example,

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

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

> **NOTE:**
> The NBGT, NBNE, and other TCP probe commands expect integer arguments only with an optional plus sign (+) or minus sign (-). A script parses up to the first non-digit character. Therefore, 50.5 is parsed to 50 and the remaining digits are ignored. To compare against a fraction or floating point value, use the EVAL statement described above.

For more information on these commands, see TCP Probe Command Reference.

`<script>` Section

You can use the `<script>` section for 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

You can 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. For information on viewing a TCP Probe script example, see Example TCP Probe File.

- Script Process Flow
- Script Command Format
- String Argument Format
- String Matching
- Numeric Argument Format
- Using Labels for Program Control
- Using Variables
- Handling Script Failures
- Adding Comments

Script Process Flow

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

Script Command Format

All script command keywords have the following requirements:

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

Example

The **MTCH** command uses the following format:

```
MTCH "string" #fail
```

The following command statement:

```
MTCH "blah" else goto #7
```

is treated exactly the same as the following:

```
MTCH "blah" #7
```

When parsing the statement, Intermapper ignores else goto to allow you to include comments to make the behavior of the script more obvious. This extraneous text does not have to be in upper case.

String Argument Format

Some commands take string arguments. String arguments must be enclosed in double quotation marks (" ").

Example

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

Special Characters

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

\r	Carriage Return
\n	Linux 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.
- Type i after the closing quotation mark to make the matching case-insensitive.

Examples

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

Wild-Card Character Matching

In some cases, you can match a more generic pattern using simple regular expressions to match patterns and place them into variables.

To use regular expressions in MTCH and EXPT:

- Type **r** after the closing quotation mark of the match string to indicate that the string content is a regular expression.
- Type **i** after the closing quotation mark of the match string to indicate that the match is case-insensitive.
- An expression inside 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. You should copy the contents into another variable after a match. For example,

Simple Example

"red"r matches "fred", "Fred", "tred", "bred", and so on. 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 pound sign (#) followed by digits.

Example

```
WAIT #30
```

Using Numeric Arguments With the GOTO Command

In many cases, numeric arguments 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 plus sign (+) or a minus sign (-) after the pound sign (#) to express a relative offset from the current statement.

Example

```
GOTO #+2
```

Default Values and Script Termination

- If a command uses 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 use the following format:

```
@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 an offset (in statements) of #+n or #-n to jump forward or backward *n* statements, respectively.

Example

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

Using Variables

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

- Variable 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 with curly braces ({ }).
- Variable names are case-insensitive.
- See Built-In Variable Reference for more 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 following are the built-in macros:

${_LINE:<line>}	The first <num> characters of the last line received.
${_BASE64:<param>}	The Base-64 encoding of the string that follows the colon (:).

${_CVSPASSWORD:<param>}	The value of <param> encoded for use as a password over the CVS pserver protocol.

Handling Script Failures

Certain script commands might fail because they are malformed or because an unexpected situation occurs. For example, the script could jump to a non-existent command, fail to match a string it expects, or unexpectedly disconnect. 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 terminates in 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 might go to an idle handler. See the WAIT command for more details.

Adding Comments to Your Script

You can add comments to your script by doing one of the following:

- Add text between or after arguments.
- Add a comment using the Intermapper probe file comment format.

Adding Text Within a Command Line

You can add text between arguments or after the line in a command-line.

Examples

The following statements all do the same thing:

```
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 using the following rules:

- Begin a comment with `<!--`.
- End a comment with `-->`.
- Do not use `--` within the comment.

Example

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

`<script-output>` Section

The `<script-output>` section of the TCP probe file formats the information retrieved from the device and sends it to the device Status window. Format the script output using IMML, Intermapper's Markup language.

TCP Probe Command Reference

The following commands are 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.

Device I/O Commands

The following commands send data to the device or read 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 read one or more lines to get the current line.

- EXPT "string" #fail - searches incoming lines for the specified string.
- MTCH "string" #fail - searches the next incoming line for the specified string.
- SKIP "string" #fail - ignores all incoming lines containing the specified string.
- DISC #discfail - jumps to a specified line number if the probe is suddenly disconnected.
- CONN #timeout ["TELNET"]["SECURE"] - specifies the connect timeout of the probe and whether to process Telnet options.
- RCON - reconnects to the specified server and port.
- PORT #port_num #connect_timeout - this is no longer required (the remote port number is now a separate parameter in the configuration dialog).

- LINE [ON | OFF] - specifies whether the script reads incoming data as lines or as raw data.
- NEXT - clears the input buffer so that subsequent **MTCH** commands operate on newly-received information.
- SEND "string" - sends the specified string to a remote device.
- BRCV {BER sequence} - receives TCP data and decodes from BER format into a local format.
- BSND {BER sequence} - encodes local data in BER format and sends. See LDAP probes for examples and syntax.

Commands That Control Script Flow

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

- CHCK "string" #fail - determines if the string is not empty.
- DONE *status* ["message"] - terminates a script with a specified condition.
- EXIT - terminates a script with the condition specified previously by **STAT**.
- FAIL - specifies the line to jump to if a CONN command fails to connect.
- GOTO #statement - branches immediately to the specified statement number.
- NBGT #arg1 #arg2 #line - (**N**umeric **B**ranch **G**reater **T**han) branches to #line if #arg1 is greater than #arg2.
- NBNE #arg1 #arg2 #line - (**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 - (**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"] - specifies the status condition of a script when it ends.
- WAIT #secs #idlefail #discfail - specifies the number of seconds the probe waits for a response.

String Processing Commands

The following commands process and manipulate strings:

- EVAL $result := expression - assigns the evaluated value of `expression` to `${result}`.
- STOR "variable" "string" - stores the string into the variable named variable.
- SCAT "variable1" "variable2" #fail - concatenates variable1 and variable2, placing the resulting string in variable1.
- NADD "variable" #number - (**N**umeric **Add**) adds a numeric value to a variable.

Commands That Measure Time

- STRT - starts a millisecond timer that Intermapper can use to determine the elapsed time for an event.
- TIME "variable" - sets the named variable to the current number of milliseconds from the most recent STRT command.
- WAIT #secs #idlefail #discfail - specifies the number of seconds the probe waits for a response.

Probe Command Details - Alphabetical

BRCV {BER Sequence}

Receives TCP data and decodes 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 and syntax.

> **NOTE:**
> Documentation of the BER format is beyond the scope of this manual.

Intermapper- specific BER syntax required information:

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

> **NOTE:**
> Remember that `${}` is the variable format. Do not confuse the sequence start and end characters `{}` with the variable delimiters.

Example

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

BSND {BER sequence}

Encodes local data in BER format and sends. BER stands for Basic Encoding Rules for ASN.1. See LDAP probes for examples and syntax.

> **NOTE:**
> Documentation of BER format is beyond the scope of this manual. See Intermapper-specific BER syntax above for the information on using `BSND`.

Example

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

CHCK "string" #fail

Use the **CHCK** command to determine if the string is not 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 connection timeout of the probe and whether to process Telnet options.

If you 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 - specifies the number of seconds to wait while trying to connect before giving up.

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

SECURE - creates an SSL connection, places the word SECURE at the end of the line.

SECURE:ADH - uses anonymous Diffe-Hellman key exchange.

SECURE:NO_TLS - TLSv1 is disabled 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 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 one of the following conditions:

```
[OKAY | WARN | ALRM | DOWN]
```

The optional message parameter allows you to provide more detail about the condition. The status values for the **DONE** command must be in upper case .

Example

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

This example sets the status of the device to ALRM. The condition of the device (displayed in the device Status window and the Device List window) is set to [HTTP] 500 Response received. to provide you with 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`.

This expression can use any operator or function defined in Probe Calculations. Using this expression, you can 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 regular expressions.

Examples

```
-- Simple Assignment
EVAL $msgstring := $otherstring

--Simple subtraction of numeric values
EVAL $newopen := $fileopen - $prevfileopen
```

```
--Use of function and conditional logic
EVAL $prevtest := defined("test") == 1 ? $test : 0

--Use of regular expression
EVAL $msg_part := ($msg =~ "(.*)| *([^|]+)$")
```

Numerous examples that use the EVAL command can be found in the built-in TCP probes.

> **NOTE:**
> Do not confuse the EVAL command used in TCP probes with the **${eval} macro** available in the output sections of command-line, SNMP, and TCP probes.

EXIT

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

EXPT "string" #fail

Use the **EXPT** command to **EXP**ec**T** or search for the specified string in any number of incoming lines.

- If the string is found, the script goes to the next statement.
- If the string is not found, the script goes to the statement specified in the fail parameter.

> **NOTE:**
> - EXPT is identical to MTCH, except in the following ways:
>   - 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.

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 goes to the statement specified by #fail.

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

FAIL

Specifies the line number to go 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.

> **NOTE:**
> - By default, the script reads in **LINE ON** mode. The incoming data is read until it is terminated by a CR-LF or by a plain LF before 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. If lines are longer than that, they can be treated as separate lines.

Tip - After you match some data in **LINE ON** mode, you should not match any more because your position in the buffer is not restored and you might miss something.

Possible Failures - None

MTCH "string" #fail

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

> **NOTE:**
> - MTCH is identical to EXPT except in the following ways:
>   - 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.

Example

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

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 goes to the statement specified by #fail.

If the idle timeout expires, 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 to a string and placed in the variable.

Example

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

adds 3 to the `"fred"` variable value. If `"fred"` contains 3, the result is 6. If `"fred"` contains golf, the result is 3 (because the conversion from a string to a number yields 0).

If the number is missing, the script adds 0 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 `@exit` label if the `${arg1}` numeric is greater than the `${arg2}` numeric.

> **NOTE:**
> Use a leading pound sign **#** to force Intermapper to treat arguments as numeric values.

Possible Failures None

NBNE #arg1 #arg2 #line

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

Possible Failures None

NEXT

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

> **NOTE:**
> - 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 `@exit` label if the string `${arg1}` is not equal to `${arg2}`.

Possible Failures None

SCAT "variable1" "variable2" #fail

The **SCAT** (**S**tring Con**CAT**enate) 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 65,535 characters, the SCAT command fails and transfers 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 quotation convention.

Example

```
SEND "Greetings!\r\n"
```

transmits the data "Greetings!" followed by a CR-LF.

Possible Failures This command cannot fail. If the data cannot be sent because of a network failure or device failure, the failure appears 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 the following:

```
[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. You can set a variable to a numeric value by enclosing the number in double quotation marks (" "). Subsequent parts of the script 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 are limited 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** command, 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 is 60 seconds.

Parameter 2 - #idlefail - If present, the script goes 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.

Parameter 3 - #discfail - If present, the script goes 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) MTCH "331" else goto -2- #16
06) SEND "PASS ${Password}\r\n"
07) MTCH "230" else goto -3- #20
08) SEND "NOOP\r\n"
09) MTCH "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) 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})"
19)
20) MTCH "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 the script fails if it does not receive responses back within 30 seconds.

```
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 do not want to match an incoming "220-"; the incoming dash indicates there are still more 220 lines to be read but 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 else goto -1- string has no function (except readability) in the script command text. This statement can also be written 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 - sends the FTP USER command. With this command, we send the user ID specified by the user, for example, 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 goes to statement 16. Unlike the **EXPT** command, the **MTCH** command fails immediately if the next line does not 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; meaning, if an unexpected response to the USER command is received. The response is checked to see if it matches 500, which indicates that the command is not 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, in milliseconds, of a device as it is tested by a TCP probe.

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

The following are TCP timers:

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

TCP Script Commands

Intermapper supports the following commands for measuring intervals during a script:

STRT	Starts the probe's custom timer.

TIME *varname*	Sets the variable named ${*varname*} to the milliseconds elapsed since the customtimer started.

The <script-output> Section

Use the optional `<script-output>` section to display the results of custom TCP probes. The data in this section is displayed in the Status window when you click and hold the device. The format of this section is the same as the `<snmp-device-display>`, described in Customized Status Windows.

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.

Accuracy

Intermapper uses the following 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 as soon as it arrives. Therefore, 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. Therefore, the measured time can be affected slightly by the such things as the number of devices probed and other various other tasks, as they can affect how long it takes Intermapper to execute a single round of polling.

Example TCP Probe File

The following is the Fortra-provided probe for the Custom TCP script:

```
<!--
Custom TCP (com.dartware.tcp.custom)
Copyright© Fortra, 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
connection. 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
  MTCH "${OK Response}" else #+2
  DONE OKAY "[Custom] Response was \"${_LINE:50}\"."
  MTCH "${WARN Response}" else #+2
  DONE WARN "[Custom] Response was \"${_LINE:50}\"."
  MTCH "${ALRM Response}" else #+2
  DONE ALRM "[Custom] Response was \"${_LINE:50}\"."
  MTCH "${DOWN Response}" else #+2
  DONE DOWN "[Custom] Response was \"${_LINE:50}\"."

  @IDLE:
    DONE DOWN "[Custom] Did not receive a line of data within
${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} msecs
  \4\Time spent connected to host:\0\ ${_active} msecs
</script-output>
```

Command Line Probes

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

Intermapper allows you to run a command-line probe, a script or program (written in perl, C, C++, or another 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 the following common sections:

- The `<header>` section of a command-line probe specifies the probe type, name, and other properties fundamental to the operation of the probe.
- The `<description>` section specifies the help text that appears in the Set Probe window. Format the description using IMML, Intermapper's Markup language.
- The `<parameters>` section defines the fields presented to the user in the Probe Configuration window.

Sections Specific to Command Line Probes

Each command-line probe also includes the following:

- `<command-line>` - defines the command-line, specifying the path of the executable, the command to execute, and arguments for the command.
- `<command-exit>` - controls how the device's state is set, based on the command results.
- `<command-display>` - 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 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 have to download the Nagios plugins to build and compile them yourself.

If you want to develop your own command-line probes, Fortra recommends that you follow the developer guidelines for Nagios. This results in probes or plugins that work for both Intermapper and Nagios.

For more information on Intermapper and Nagios Plugins, see the Nagios Plugins page.

See Command Line Probe Example for a sample shell script and corresponding probe.

The <tool> Section - Embedding a Companion Script

You can embed script code directly in a probe. This provides a way to deliver a command-line probe and a script that runs in a single probe file, ensuring that the script version matches the probe version. WMI probes provide a number of good examples of companion scripts. For more information, see The <tool> Section.

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 the following information to Intermapper:

- The operating system return code, or *exit code*, is used to indicate the success/failure/severity. This is handled by the `<command-exit>` section of the probe file.
- The script can return additional values, such as measurements discovered during execution, by writing to the script's stdout.

  These values are returned as a comma-separated list enclosed in "\{" ... "}" characters. These values can be handled as variables in the probe's `<command-display>` section. The values are name-value pairs in the following format:

  `<name> := <value>`

- The script can return a **reason string** that explains the device's condition. You can specify the reason string by writing to the script's stdout. This text should follow the closing curly braces (}) of 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.

Installing a Command Line Probe

After you create 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 another executable, create the program and make it runnable. If it is a Perl, Python, or another script type, set the permissions so that it can run from the command-line. If it is written in C, C++, or another non-interpreted language, compile the source and then place the resulting binary in an appropriate directory. For more information, see the path information 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 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 can be set as follows:

```
arg = "-H ${Timeout}"
```

> **NOTE:**
> Depending on the nature of the parameters you are passing, you can pass the parameters through STDIN, as described below.

Sending Data to STDIN

Using the `ps` command on Linux systems, or using the Task Manager or other utility programs on Microsoft Windows systems, you can 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. For a detailed example, see Sending Data to STDIN in The <command-line> Section.

`<command-line>` Section

The `<command-line>` section allows you to specify the information needed to execute the commands for the probe. Use the following variables:

- **path** - specifies the path to the executable script/command.
- **cmd** - specifies the actual script/command.
- **arg** - specifies the arguments to be passed to the script/command.
- **input** - specifies information to pass to STDIN to the script/command.

`<command-line>` Section Properties

- Use the `path` property to specify the directories where Intermapper should look for the executable to run as a probe. This is the only path that Intermapper uses. 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 a Linux or macOS system. Microsoft Windows systems use back slashes (\) instead of forward slashes (/) and semicolons (;) instead of colons (:).

> **NOTE:**
> - If no path is specified, Intermapper Settings/Tools is used as the path.
> - On Linux systems, you can 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.

- Use the `cmd` property to specify the executable you want to run. In the example below, this is check_ping. You need to specify the exact name, including extensions (such as .exe or .cmd). You can also specify arguments as part of the `cmd` property.
- Use the `arg` property to specify arguments to the executable. This can be used 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 the following:

```
<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. For more information, see Sending Data to STDIN.

  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 Linux systems, or using the Task Manager or other utility programs on Microsoft Windows systems, you can see the command line arguments. This represents a security vulnerability. Use the `input` variable to pass sensitive data to stdin without 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>
```

`<command-exit>` Section

The `<command-exit>` section allows you to specify which results from the command indicate the five Intermapper device states. The following states are available:

- down
- critical
- alarm
- warning
- okay

For each state, indicate what item Intermapper should examine and what its value should be to result in that state. At the moment, the only thing Intermapper can look at is the exit code, which is indicated with ${EXIT_CODE}. So, in the following example, 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.

`<command-display>` Section

The `<command-display>` section displays variables in the device Status window using the same format as the output section of other probe types. If the plugin returns a non-integer value, use the ${chartable:...} macro to display digits to the right of the decimal point. As with other probe types, you can format the appearance of the output using IMML, Intermapper's markup language.

See Command Line Probe Example for a sample shell script and corresponding probe.

`<tool>` Section

Use the `<tool>` section of a command-line probe to embed a script 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 a subdirectory of Tools is created with the canonical name of the probe and writes the script to that subdirectory, using `scriptname` as the file name.

If a subdirectory of that name already exists, all non-hidden files are deleted before the script is writtern. 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 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 as follows:

```
cmd="python test.py"
```

or, using the ${PYTHON} macro:

```
cmd="${PYTHON}  test.py"
```

and the tool section is as follows:

```
<tool:test.py>

  # Trivial example

  print (“okay”)
  sys.exit(0)

</tool:test.py>
```

When Intermapper starts or reloads probes, a subdirectory of Tools named `com.dartware.cmdline.test` is created if it does not 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>`.

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 path 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© Fortra, LLC. All rights reserved.

1.2  22 Mar 2021 Convert Python2 to Python3 -Jerry
-->

<header>
          type = "cmd-line"
          package = "com.dartware"
          probe_name = "commandline.check_connect"
          human_name = "Check Connect"
          version = "1.2"
          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 = int(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 as 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© Fortra, 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>
```

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© Help/Systems, 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 can use this interpreter to write command-line probe scripts and command-line notifiers. This Python interpreter provides maximum compatibility across systems. In Intermapper 6.4, 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/tut.

As shipped, this Python interpreter requires the use of optimized and stripped mode (-OO), so the interpreter must be invoked as follows:

macOS/ Linux	/usr/local/imdc/core/python3/bin/imdc -OO [script_name]
Windows	c:\Program Files\InterMapper\dwf\core\python3\imdc.exe -OO [script_name]

NOTE:

- Use the ${PYTHON} macro as shown below to determine the platform and expands to the proper path to the interpreter with the `-OO` argument.
- Use the <tool> Section (`<tool: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 the following example. The script is automatically 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
Updated 22 Mar 2021 to change python2 to python3 -Jerry
```

```
-->

<header>
        type="cmd-line"
        package="com.dartware"
        probe_name="python.sample"
        human_name="Python Sample"
        version="1.2"
        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:${EXIT_CODE}=4
        critical:${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/python3/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(5)

print("Pretending we got result %d from device at address %s" %
(result, addr))
sys.exit(result)

</tool:sample.py>
```

Upgrading to Python 3

Starting with version 6.5.2 of Intermapper, the application stack is updated from Python 2.7.1 (used in Intermapper-6.5.1 and earlier versions) to Python 3.8.5. All Python code shipped within the Intermapper combined installer has been converted to Python 3 code. This includes the server-side code of Intermapper DataCenter, the Switches (Layer-2) extension, and the definitions of the Python-coded probes bundled with Intermapper.

When the Intermapper server invokes the code of a registered probe, it honors the command-line content of the probe’s definition. Probes coded with Python logic typically use a line similar to `cmd=${PYTHON} <filename>.py` to invoke the Python interpreter. The ${PYTHON} token is substituted with the path name of the Python interpreter embedded in Intermapper.

For Intermapper 6.5.2, this token references a Python 3.8.5 interpreter rather than a Python 2.7.1 interpreter. This means that the Python code in the probe itself likely needs to be updated from Python 2 to Python 3 because a significant number of Python 2 coding idioms and practices are not supported by Python 3. For information on upgrading Python 2 to Python 3, see the Python platform documentation at https://www.python.org/, more specifically, https://portingguide.readthedocs.io/en/latest/.

Intermapper 6.5.2 delivers Python 3 code as a virtual environment. On Linux and macOS systems, the root directory of the Python tree is typically `/usr/local/imdc/core/python3`. On Microsoft systems, the root directory of the Python tree is typically `C:\Program Files\InterMapper\dwf\core\python3`.

The simplest way to test or troubleshoot existing Python code is to activate the virtual environment to run the code in isolation. For example, on macOS systems, use the following command:

```
$ . /usr/local/imdc/core/python3/venv/bin/activate
(venv) $ python3
```

```
Python 3.8.5 (default, Mar 22 2021, 04:12:07)
[Clang 6.0 (clang-600.0.54)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

For Microsoft Windows systems, run the following command:

```
C:\bin>"C:\Program Files\InterMapper\dwf\core\python3\activate"
(python3) C:\bin>python
Python 3.8.5 (heads/support-6.5.2-dirty:1d13bb49, Mar 22 2021,
12:32:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

By activating the virtual environment in this way, you can ensure that all Python extensions with which the Intermapper Python installation is provisioned is available to your Python code. If you have difficulty re-targeting your Python code that supports a current probe definition to the Python 3 code, contact Technical Support at Fortra.

If you have Python probes that work with Intermapper 6.5.1 and lower, you might need to run 2to3 conversion to upgrade to Python 3. To do that, extract the Python portion of your probe into a .py file, run the 2to3 conversion, and copy the modified content back into your probe.

Nagios Plugins

Intermapper's command-line probes are similar to **Nagios® plugins** (http://www.nagios.org). You can see the standard set of Nagios plugin. Many vendors and individuals have created their own Nagios plugins, many of which are available in the development section. To use these plugins, download them and build and compile them yourself.

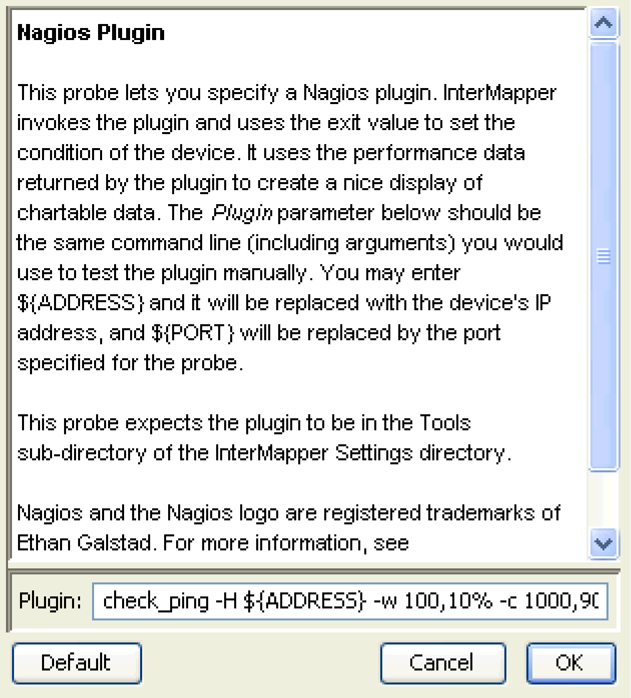
The Nagios Plugin probe allows you to specify a Nagios plugin to run, along with associated parameters. You can use the ${ADDRESS} and ${PORT} macros in the command-line; Intermapper substitutes the device IP address and the specified port. Intermapper invokes the plugin and uses the exit value to set 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, displaying and making performance data returned by the probe chartable. It also displays 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 the device Probe Type to **Nagios Plugin**.
4. Enter the plugin file 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 substitutes the device IP address and the specified port.



Nagios Plugin

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 the performance data returned by the plugin to create a nice display of chartable data. The *Plugin* parameter below should be the same command line (including arguments) you would use to test the plugin manually. You may enter ${ADDRESS} and it will be replaced with the device's IP address, and ${PORT} will be replaced by the port specified for the probe.

This probe expects the 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

Plugin: check_ping -H ${ADDRESS} -w 100,10% -c 1000,9(

Creating Nagios Probes

If you want to develop your own Nagios plugin, follow the developer guidelines for Nagios (found at http://nagiosplug.sourceforge.net/developer-guidelines.html). This results in probes/plugins that work for both Intermapper and Nagios.

As described in the Nagios Guidelines, a Nagios plugin returns the following:

- **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 use the following format:

  ```
  SERVICE STATUS: information text
  ```

- Pipes (|) 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 the following format:

  ```
  'label'=value[UOM];[warn];[crit];[min];[max]
  ```

Example Return String

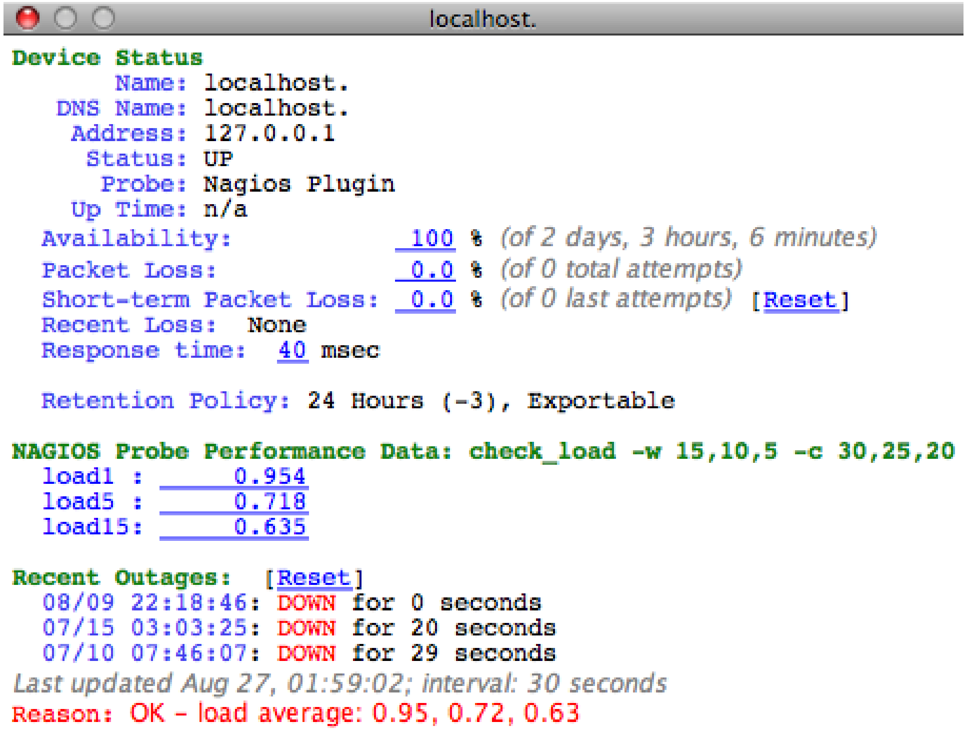
The Nagios check_load string returns the following load averages:

- Average over 1 minute
- Average over 5 minutes
- Average over 15 minutes

  When the plugin is invoked, it returns a response similar to the following:

```
% ./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 response line and uses the ${nagios_output} macro to produce a status window. For example,



```
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, go to http://www.nagios.org. Nagios® and the Nagios logo are registered trademarks of Ethan Galstad.

Intermapper 5.0 Changes

For those familiar with the older Nagios Template probe, the new Nagios Plugin probe contains the following changes in behavior:

- The Nagios Template probe maps 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 takes 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.
- The canonical name of the Nagios probe has not changed. However, any device which used the old Nagios Template probe now automatically uses the Nagios Plugin probe. An Intermapper probe automatically handles a Nagios plugin if it includes 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© Fortra, 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. \${ADDRESS} 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
Galstad. For more information, see \U2\http://www.nagios.org\P0\
</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}\P0\
${nagios_output}
</command-display>
```

NOAA Weather Probe Example

It is now easier than ever to build command-line probes. This example 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 (for example, KLEB, Lebanon Municipal Airport). 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 the XML response to retrieve the temperature. (There is a lot more information in the Weather Service feed; you can extend the program to display more information.) The following are some features of this probe:

- The ${PYTHON} macro provides the path to the built-in python interpreter of Intermapper DataCenter no matter what platform you use. For example, the probe can now use the following:

  ```
  cmd = "${PYTHON} program.py"
  ```

  Intermapper substitutes the proper path to invoke Python, whether on Microsoft Windows, macOS, or Linux systems.

  > **NOTE:**
  > To use this macro, the Intermapper DataCenter (IMDC) must be installed. IMDC is installed automatically with Intermapper 5.2 on Microsoft Windows and OSX systems; Linux systems require a separate installation for IMDC.

- You can include the script directly in the probe file text.This makes it 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 can also save private files in that directory.

- The example probe file uses a few Python libraries. For example, urllib2 makes it easy to make queries from web services. It includes a few straightforward calls to build a URL, issue it, and retrieve the results.
- The probe uses the xml.dom.minidom library to parse XML data returned from the NOAA web service. For more information on this library, see Chapter 9 of Dive into Python.

NOAA Temperature Probe

To use this probe, copy the text below, paste it to a text editor, save it to a text file, and click File > Import> Probe... in Intermapper.

```
<!--
Weather Service Temperature - Retrieve the temperature from the NOAA weather XML
(com.dartware.tool.noaa.txt) Copyright© Fortra, 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 = "1.3"
     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.request, urllib.parse, urllib.error
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 as 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.parse.quote_plus(station))
```

```
# print noaaString;
retcode = 4;
try:
      request = urllib.request.Request(noaaString)
      opener = urllib.request.build_opener()
      request.add_header('User-Agent', userAgent)
      usock= opener.open(request)
#      print buf
except IOError as 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, tempval + '
degrees at ' + locval))
sys.exit(retcode)
</tool:noaa_weather.py>
```

See also

${PYTHON} macro - for the full path the to the Python interpreter.

The <tool> Section - to 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: for information on XML processing in Python, see http://diveintopython.org/xml_processing/.

PowerShell_Probe

A PowerShell probe is a command-line probe with a PowerShell Script attached to it.

The only difference is in the following script arguments:

- Arguments passed to the Microsoft Windows command-line.
- Arguments passed to the script itself.

The things you can do with a PowerShell probe are virtually limitless.

See the PowerShell Probe Example for more information.

PowerShell Probe Examples

PowerShell probes are command-line probes. They launch PowerShell and invoke a command.

The following examples demonstrate two different Intermapper macros:

- **${PSREMOTE}** - for less experienced PowerShell users, this macro handles the connection to the remote machine. It creates a credential object and sets up authentication. It executes the specified command on the remote machine.
- **${PS}** - for experienced PowerShell users, this macro launches PowerShell on the local machine with the specified arguments. It leaves all PowerShell commands up to the developer.

These macros are used in the `<command-line>` section of the probe.

Scripts must be located in the `Intermapper Settings\Tools` folder. If you include the script in the `<tools>` section, it is installed in the Tools folder when you load the probe.

Example 1: Installed Software Probe

This probe lists installed applications, updates, or both on the target device. It launches PowerShell with the arguments supplied in `arg`, uses `${PSREMOTE}` to connect to and authenticate on the remote device, and executes the command specified `input`.

```
<!--
This probe lists installed Applications, Updates, or Both using
PowerShell. Requires PowerShell 2.0 or later and requires that PS
remoting be enabled.

File Name: com.helpsystems.powershell.remote.installedSoftware.txt
(c) 2015 Fortra, Inc.
-->
```

```
 <header>
       type = "cmd-line"
       package = "com.helpsystems"
       probe_name = "ps.remote.InstalledSoftware"
       human_name = "Installed Software"
       version = "1.0"
       address_type = "IP"
       display_name = "PowerShell/Remote/Installed Software"
       visible_in = "Windows"
       flags = "NTCREDENTIALS"
</header>

<description>
\GB\List Installed Software\p\

This probe uses PowerShell to provide a listing of installed
software, installed updates, or both. This probe requires that
\b\PowerShell 2.0\p\ or later be installed, and PowerShell remoting
must be enabled and configured to use this probe. This probe uses
the registry, not WMI objects

Intermapper invokes the included ApplicationList.ps1 companion
script in Intermapper Settings/Tools.

</description>

<parameters>
       "Type[Software,Update,All]"="Software"
       User=""
       "Password*" = ""
    "Authentication
[Default,Basic,Negotiate,NegotiateWithImplicitCredential,Credssp,Di
gest,Kerberos]"="Default"
    "Timeout (sec)"="10"
</parameters>

<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-line>
       path=""
       cmd="${PSREMOTE}"
       arg="-ExecutionPolicy RemoteSigned -NoProfile"
       input = "Invoke-Command -FilePath .\\ApplicationList.ps1 -
ArgumentList '${Type[Software,Update,All]}'"
       timeout = ${Timeout (sec)}
</command-line>

<command-display>
  ${Type[Software,Update,All]} installed on ${address}
  ${^stdout}
</command-display>

<tool:ApplicationList.ps1>
param([string] $filter)

$exitCode = 0
$reason = ''

if ($filter -eq 'All')
{
    $software = Get-ItemProperty
HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninsta
ll\* | Select-Object DisplayName, DisplayVersion, InstallDate,
Publisher | Sort-Object DisplayName
}

elseif ($filter -eq 'Update') #show only windows updates
{
    $software = Get-ItemProperty
HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninsta
ll\* | Select-Object DisplayName, DisplayVersion, InstallDate,
Publisher | where {$_.DisplayName -match $filter} | Sort-Object
DisplayName
}

elseif  ($filter -eq 'Software')
{
    $software = Get-ItemProperty
HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninsta
ll\* | Select-Object DisplayName, DisplayVersion, InstallDate,
Publisher | where {$_.DisplayName -notmatch 'update'} | Sort-Object
DisplayName
}

#get rid of blanks in the input
$cleanedUpList = New-Object System.Collections.ArrayList

foreach($app in $software)
{
    if ($app.DisplayName )
    {
        $cleanedUpList.Add($app) | Out-Null #ArrayList.Add returns
the index of the item added, we don't want this goint to standard
out, confusing Intermapper.
    }
}

#set up the object for return

# Intermapper can't take an array or ArrayList of objects yet, so
convert to a string.
# Also, Powershell will truncate to a default size, unless the table
is formatted using -AutoSize and get around column dropping by
setting the width of the resulting string.
$stdoutString = $cleanedUpList |format-table -AutoSize | out-string
-width 4096

$result = New-Object PSCustomObject -Property @{
    'stdout'=$stdoutString;
    'ExitCode'=$exitCode;
    'reason'=$reason;
}
write-output $result

</tool:ApplicationList.ps1>
```

Example 2: Windows Disk Space Probe

This probe checks the amount of disk space on the target device. It uses ${PS} to launch PowerShell with the arguments supplied in `arg` and executes the command specified in `input`.

Similarly to the first example, this example connects to a remote device, but does not use `${PSREMOTE}` to handle the connection. This example also passes thresholds to the script so that it can return the correct exit code.

```
<!--

Windows Disk Space Probe
This probe uses a PowerShell script to look up the amount of disk
space on the
target device.
(c) 2015 Fortra, Inc.
-->
```

```
 <header>
       type = "cmd-line"
       package = "com.helpsystems"
       probe_name = "ps.wmi.diskspace"
       human_name = "Non-Remoting (WMI) Disk Space Monitor"
       version = "1.0"
       address_type = "IP"
       display_name = "PowerShell/Disk Space"
       visible_in = "Windows"
    flags = "NTCREDENTIALS"
</header>

<description>
       \GB\Windows Disk Space Monitor\p\

       This probe uses Powershell to retrieve the disk space available on
a drive
       on the target host. Specifically, it queries the Size and FreeSpace
       properties of the Win32_LogicalDisk class, computes percentage free
space,
       and compares it against the Warning and Critical parameters you
set.  The
       target host must be running PowerShell with Remoting enabled.

       The Drive parameter may be set to "All" to enumerate all Local hard
drives
       on the host.  It may also be set to a list of comma-separated drive
names
       (including the colon), which will be listed whether or not they are
local
       hard drives.  Zero-sized drives (i.e. an empty cd-rom) will not be
listed.
       The first drive failing the warning or critical criteria will be
the one
       cited in the reason.

       The User parameter may be a local user on the target host, or may
take the
       form of "domain\\user" for a domain login.  Leave it blank if
authentication
       is not required, such as when the target is the localhost.

       Intermapper invokes the WindowsFreeDiskSpace.ps1 companion script
which was
       placed in the Tools folder of the Intermapper Settings folder when
this
       probe was loaded.  It uses the exit value to set the condition of
```

```
the device
       and the performance data returned by the script to create a nice
display of
       chartable data.
</description>

<parameters>
       Drive="C:"
       "Warning (%)"="10"
       "Alarm (%)"="5"
       "Critical (%)"="3"
       "Down (%)"="1"
       User=""
       "Password*" = ""
       "Timeout (sec)"="10"
    "Powershell Version
[notSpecified,2.0,3.0,4.0,5.0]"="notSpecified"
</parameters>

<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-line>
       path=""
       cmd="${PS}"
       arg="-ExecutionPolicy RemoteSigned -NoProfile"
       input = "$c = New-Object System.Management.Automation.PSCredential
-ArgumentList '${User}', (ConvertTo-SecureString -String
'${Password*}' -AsPlainText -Force) ; Invoke-Command -ScriptBlock {
& '.\\WindowsFreeDiskSpace.ps1' -compName '${address}' -cred $c -
drives '${Drive}' -downThr ${Down (%)} -critThr ${Critical (%)} -
alrmThr ${Alarm (%)} -warnThr ${Warning (%)} }"
       timeout = ${Timeout (sec)}
</command-line>

<command-display>

Disk Space Available
    ${^stdout}
</command-display>

<tool:WindowsFreeDiskSpace.ps1>
param([string] $compName,
[System.Management.Automation.PSCredential] $cred, [string] $drives,
[int] $downThr, [int] $critThr, [int] $alrmThr, [int] $warnThr)

Function Update-ExitCode
{
    param([int] $current_status, [int] $new_status)

    if ($new_status -gt $current_status) { return $new_status }
    else { return $current_status }
}

Function Update-Reason
{
    param($disk_name,$disk_threshold)
    return "Disk $disk_name is below $disk_threshold % free."
}

$STATUS = New-Object -TypeName PSObject -Prop(@
{'down'=4;'critical'=3;'alarm'=2;'warning'=1;'ok'=0})

#reason and exit code
[int] $exit_code = $STATUS.ok
[string] $reason = "All disks within acceptable limits"
[string] $debugInfo = ''

$disks = New-Object -TypeName System.Collections.ArrayList

if ($drives -eq "All")
{
    $disks = (Get-WmiObject Win32_LogicalDisk -ComputerName
$compName -Credential $cred -Filter "DriveType='3'" | Select-Object
Size,FreeSpace,DeviceID)
}

else
{
    $diskList = (Get-WmiObject Win32_LogicalDisk -ComputerName
$compName -Credential $cred | Select-Object Size, Freespace,
DeviceId)

    $driveArray = $drives.replace(' ', '').split(',')
    foreach($drive in $driveArray)
    {
        #$debugInfo += "$drive`r`n"

        $found = $false

        foreach($disk in $diskList)
        {
            #$debugInfo += $disk.GetType().FullName

            if ($disk.DeviceID -eq $drive)
            {
                $found = $true
                if ($disk.Size -ne $null)
                {
                    #$debugInfo += " -- adding " + $disk.DeviceID +
"  Size: " + $disk.Size + "  FreeSpace: " + $disk.FreeSpace + "`r`n"
                    $disks.Add($disk)
                }
                else
                {
                    $debugInfo += $disk.DeviceID + "  --- No
information ---`r`n"
                }
            }
        }

        if ($found -ne $true)
        {
            $debugInfo += $drive + "  --- Not found ---`r`n"
        }
    }
}

if ($disks.count -eq 0)
{
    throw "Disks could not be found or parameter error. Check your
probe settings.`r`n" + $debugInfo
}

foreach ($disk in $disks)
{
    #calculate percentage of the disk that is free
    $disk | Add-Member -type NoteProperty -name "PercentFree" -value
([Math]::round($disk.FreeSpace / $disk.Size * 100))
    $disk.Size          = "{0:N1}" -f [Math]::round
(($disk.Size/1GB))
    $disk.FreeSpace     = "{0:N1}" -f [Math]::round
(($disk.FreeSpace/1GB))

    # calculate alerts
    if ($disk.PercentFree -le $downThr)
    {
        # $disk | Add-Member -type NoteProperty -name "exit_code" -
value $STATUS.down
```

```
        $old_code = $exit_code
        $exit_code = $STATUS.down
        if ($old_code -ne $exit_code)
        {
            $reason = Update-Reason $disk.DeviceID $downThr
        }
    }

    elseif ($disk.PercentFree -le $critThr -and $disk.PercentFree -
gt $downThr )
    {

        # $disk | Add-Member -type NoteProperty -name "exit_code" -
value $STATUS.critical
        $old_code = $exit_code
        $exit_code = Update-ExitCode $exit_code $STATUS.critical
        if ($old_code -ne $exit_code)
        {
            $reason = Update-Reason $disk.DeviceID $critThr
        }
    }

    elseif ($disk.PercentFree -le $alrmThr -and $disk.PercentFree -
gt $critThr )
    {
        # $disk | Add-Member -type NoteProperty -name "exit_code" -
value $STATUS.alarm
        $old_code = $exit_code
        $exit_code = Update-ExitCode $exit_code $STATUS.alarm
        if ($old_code -ne $exit_code)
        {
            $reason = Update-Reason $disk.DeviceID $alrmThr
        }
    }

    elseif ($disk.PercentFree -le $warnThr -and $disk.PercentFree -
gt $alrmThr )
    {
        #$disk | Add-Member -type NoteProperty -name "exit_code" -
value $STATUS.warning
        $old_code = $exit_code
        $exit_code = Update-ExitCode $exit_code $STATUS.warning
        if ($old_code -ne $exit_code)
        {
            $reason = Update-Reason $disk.DeviceID $warnThr
        }
    }
    else
    {
        #$disk | Add-Member -type NoteProperty -name "exit_code" -
value $STATUS.ok
        $exit_code = Update-ExitCode $exit_code $STATUS.ok
    }
}

#format the output for the probe to display in the status window
$stdoutString = ($disks | Format-Table
DeviceID,Size,FreeSpace,PercentFree | out-string)
$stdoutString += $debugInfo


#create the return object that the probe will use for display

$result = New-Object PSCustomObject -Property @{
    'stdout'=$stdoutString;
    'ExitCode'=$exit_code;
    'reason'=$reason;
}
write-output $result

</tool:WindowsFreeDiskSpace.ps1>
```

Troubleshooting PowerShell Probes

When you run a PowerShell probe, Intermapper launches PowerShell.exe and executes a command or script. It passes the following input:

- The parameters used to launch PowerShell.
- The command executed after PowerShell is launched.

Intermapper combines these inputs into a single command, which may reference a separate PowerShell script. All scripts (or links to them) must reside in the Intermapper Settings\Tools folder.

Each time a PowerShell probe is chosen, or when its parameters change, two things happen.

1. A connectivity test is run.
2. If the test is successful, the probe runs at the next polling interval.

For the connectivity test, and for each time a PowerShell probe runs, the following entries are created in the Debug log:

- One entry shows the input string sent to `stdin`.
- Another entry shows the variables returned by the probe, enclosed in "\{...}", followed by the string assigned to `stdout`.

Example Debug Log Entries

For each entry, the first two sets of numbers are as follows:

- Time
- IP address of the target device

Connectivity Test Command

```
12:56:14 10.65.49.31 : Remoting Disk Space Monitor:
XCmdLine::SendProbe: stdin: 1090 $global:t = 0 ; try { ;
$ErrorActionPreference = 'Stop' ;   $global:t = 1 ;   Write-Output
"PSRTest: $global:t" ;   $vMaj = $PSVersionTable.PSVersion.Major ;
Write-Output $vMaj ;   $global:t = 2 ;   Write-Output "PSRTest:
$global:t" ;   Test-WSMan 10.65.49.31 ;   $global:t = 3 ;   Write-
Output "PSRTest: $global:t" ;   $cred = New-Object
System.Management.Automation.PSCredential -ArgumentList '\Fred
Flintstone', (ConvertTo-SecureString -String '*************' -
AsPlainText -Force) ;   Connect-WSMan 10.65.49.31  -Authentication
Default -Credential $cred ;   $maxConnections = Get-ChildItem -Path
WSMan:/10.65.49.31/Service/MaxConnections ;   Disconnect-WSMan
10.65.49.31 ; Write-Output $maxConnections ;   $global:t = 4 ;
Write-Output "PSRTest: $global:t" ;   $sess = New-PSSession
10.65.49.31  -Authentication Default -Credential $cred ;   $result =
New-Object PSCustomObject -Property @{ 'State'=$sess.State;
'Availability'=$sess.Availability } ;   Remove-PSSession -Id
$sess.Id ; Write-Output $result ; } catch { ;   throw "Exception in
PSRTest: $global:t $_.Exception.Message" ; }
```

Connectivity Test Response

```
12:56:19 10.65.49.31 : Remoting Disk Space Monitor:
XCmdLine::PollProbeForPS -- Reason: \{ reason:='PowerShell Remoting
Test succeeded; Your probe will run next probe cycle.'} ***
PowerShell Tests *** Running with PowerShell version 3.0.
Test-WSMan succeeded: received expected response.
Connect-WSMan succeeded: MaxConnections = 300.
```

```
New-PSSession succeeded: State = Opened, Availability = Available.

The PowerShell Remoting Test succeeded. Your probe will run next
probe cycle.
```

Sending a Command

```
12:56:44 10.65.49.31 : Remoting Disk Space Monitor:
XCmdLine::SendProbe: stdin: 409 $cred = New-Object
System.Management.Automation.PSCredential -ArgumentList '\Fred
Flintstone', (ConvertTo-SecureString -String '*************' -
AsPlainText -Force) ; $sess = New-PSSession 10.65.49.31  -
Authentication Default -Credential $cred ; try { Invoke-Command -
Session $sess -FilePath .\WindowsFreeDiskSpace.ps1 -ArgumentList
localhost, 'C:, D:, L:', 1, 3, 5, 10 } finally { Remove-PSSession -
Id $sess.Id }
```

Command Response

```
12:56:49 10.65.49.31 : Remoting Disk Space Monitor:
XCmdLine::PollProbeForPS -- Reason: \{
PSComputerName:='10.65.49.31',RunspaceId:='a8f1f138-7781-4e77-a185-
18aa6db978c9',PSShowComputerName:='true',reason:='Disk C: is below 5
% free.'} DeviceId Size  Freespace PercentFree
-------- ----  --------- -----------
C:       918.0 43.0                5
D:       13.0  2.0                12
L:       932.0 917.0              98
```

Installing and Modifying Probes

Use custom probes to enhance Intermapper's capabilities. These probes are created for special purposes or for certain devices.

To install a custom probe:

1. Download the probe and decompress the file if necessary.
2. From the File menu, do one of the following:
    - Select **Import** > **Probe** command.
    - Click the **Plus** icon on the right in the Set Probe window. From the dialog, select the probe file you want to import and click **Open**. The probe is installed and

copied to the Intermapper Settings/Probes directory. The probe is available in 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 select **Set Probe**. The Select Probe window is displayed.
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, do one of following before activating the changes:

- Re-import the probe as described above. You can do this regardless of the file location of the probe.
- Manually reload probes. If you make a change to the probe file located in the Intermapper Settings/Probes directory, click **Reload Probes** (on the 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.

Before you can view or modify a built-in probe, you need to unzip the archive.

Resolving 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 following:

- the probe **version number**
- the probe **last-modified date**

If you are developing or modifying a built-in probe, update the version number to match the Intermapper version.

Sharing Probes

Intermapper has a user-base that contributed hundreds of probes, many of which have been adopted as built-in probes. Fortra encourages you to check out the library of user-contributed probes and to contribute useful probes themselves.

If you create a probe you find useful, contribute your new probe to Fortra by emailing support@intermapper.com. Fortra will then post it to the Contributions page mentioned above.

Using Contributed Probes

You can use any of the probes created by Fortra. You can also use probes contributed by our other customers. To access these probes, go to the following URL:

> http://intermapper.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 identify 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 a SQLite database.

You can also gain a lot of information by measuring response times of a device as it is tested. A number of different timers are available for viewing and charting.

Errors With Custom Probes

When working with custom probes, you might see unexpected results.

"Undefined variable" in Debug Log

When processing a probe, Intermapper does not evaluate an expression if it detects an undefined variable. A variable is undefined if it is not in the symbol table. This can happen because of one of the following:

- 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 adds 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 (which might be a legitimate case if a particular variable is undefined) you can use the defined() function in the expression:

```
 warning: defined("oid") && ($oid> 0) "Warning condition string"
```

A device shows a "Reason: No SNMP Response." at the bottom of the status window

There are several reasons that Intermapper might retrieve SNMP information from a device. For example,

- The device does not speak SNMP.
- You did not enter the proper SNMP read-only community string.

For more information, see *About SNMP* in the Troubleshooting section of the User Guide.

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 that use the following format:

```
12:57:00 router.example.net.: SNMP error status [[query = 28]]
noSuchName (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 the OID. In this example, the proper OID should have a 2 in place of the 7.

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 in one of the device variables.

Open the Debug window and look for entries that use the following 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 is 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 does not 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`

> **NOTE:** When using probe groups, you cannot perform an SNMPWalk on the entire probe group, only on individual probes.
> **To perform an SNMPWalk on a probe**
>
> 1. In a map's List view, expand the probe group to view the individual probes.
> 2. Right-click the probe you want to perform the SNMPWalk and select **SNMPWalk**. The SNMPWalk window appears.
> 3. Complete the dialog as appropriate and click **OK**.

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
```

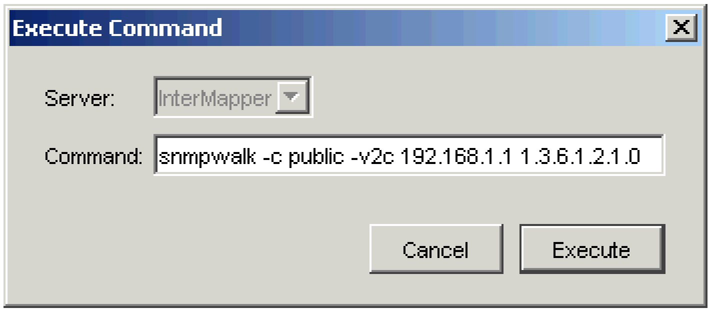
Example

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)



Server: InterMapper

Command: snmpwalk -c public -v2c 192.168.1.1 1.3.6.1.2.1.0

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 : Debug*yyyymmddhhmm*.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.

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:
support@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://www.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.

Telnet Command-Line Help

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 results of the SNMPWalk to the debug log, Intermapper can write the results to a SQLite file. To use this feature, use the ***-o*** option.

When you use the -o option, SNMPWalk stores its output into a SQLite database. To use this feature, specify the name of the database file following the -o option.

Example

To create a SQLite3 database called foo and to store the SNMPWalk results there, use the following server command:

```
snmpwalk -v1 -c public -o foo switch 1.3
```

This writes the output to a SQLite database file called foo located in Intermapper Settings/Temporary directory. The database file called foo might contain multiple SNMPWalks. The file is created if it does not already exist.

When the -o option is used, the following 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 ...
```

SNMPWALK Schema

The following 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 receives a unique identifier that identifies it (id). The other columns are as follows:

- 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.
- pktTimeout

  The packet timeout.
- pktRetries

  The packet retry count.
- maxOids

The maximum number of OIDs.

- toEnd

  A Boolean flag indicating that the walk should proceed to the end (in other words, it does not limit by startOid).
- timeStarted

  The UTC timestamp of when the walk started.
- timeFinished

  The UTC timestamp of when the walk completed.
- oidCount

  The number of OID rows walked.
- stopReason

  The reason the walk stopped when 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 identifier 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 SQLite Data

On macOS 10.4 systems, you can use the following 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 Mozilla Firefox add-on called SQLite Manager that opens and displays SQLite database files. This makes it a cross-platform tool, requiring only Mozilla Firefox 3.5 or higher, plus a small download.

To install SQLite Manager:

1. Start **Mozilla Firefox** 3.5 or higher and click **Tools** > **Add-ons** - **Get Add-ons**.
2. Type **SQLite** in the **Search** field and press **Return** on your keyboard.
3. Double-click **SQLite Manager** to install it. Restart **Mozilla Firefox** when instructed.

To use SQLite Manager:

1. Click **Tools** > **SQLite Manager** to open SQLite Manager.
2. Click **Database** > **Connect Database** (within the window) to open a saved SQLite database.
3. Click the results table to view it.

Reference

You can use the Intermapper Developer Guide reference to access the following documentation:

- **Intermapper HTTP API**- automates Intermapper operations, data import/export, and more.
- **Retrieving Collected Data** - accesses the Intermapper Database to retrieve collected data.
- **Customizing Web Pages** - customizes any Intermapper web page to suit your needs.
- **Command-line Options** - uses the command-line interface to automate or streamline maintenance and monitoring operations.
- **Intermapper Service Management for Linux Systems** - describes how to manage the Intermapper services for Linux systems.

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 straightforward URL syntax to make these requests.

The following features are provided in the HTTP API:

- **File Import/Export** - provides access to files of the Intermapper Settings folder.
- **Table Import/Export** - provides access to the tables in the same formats that are currently available in the Remote Access Import and Export commands.
- **Acknowledgements** - sends Basic Acknowledgements to an Intermapper server using HTTP. This allows you to acknowledge downed 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.

> **NOTE:**
> To use the HTTP API interface, you must connect as an Intermapper user that is a member of the Intermapper Adminstrators group, as specified in the Users pane of the Server Settings window.

The following features are provided in the HTTP API:

Importing & Exporting Files - documents how to access the Intermapper Settings directory's file system.

Importing & Exporting Tables - documents how to import or export map data directly.

Acknowledging Devices - documents how to acknowledge downed devices or interfaces using the HTTP API.

Scripting Examples - Provides examples of the cloned scripts and much more.

Importing and Exporting Files

Most of the files in the Intermapper Settings folder can be accessed from the HTTP API. These include the following:

- Custom Icons
- Fonts
- Maps
- MIB Files
- Probes
- Sounds
- Web Pages

These folder contents can 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 treated 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.

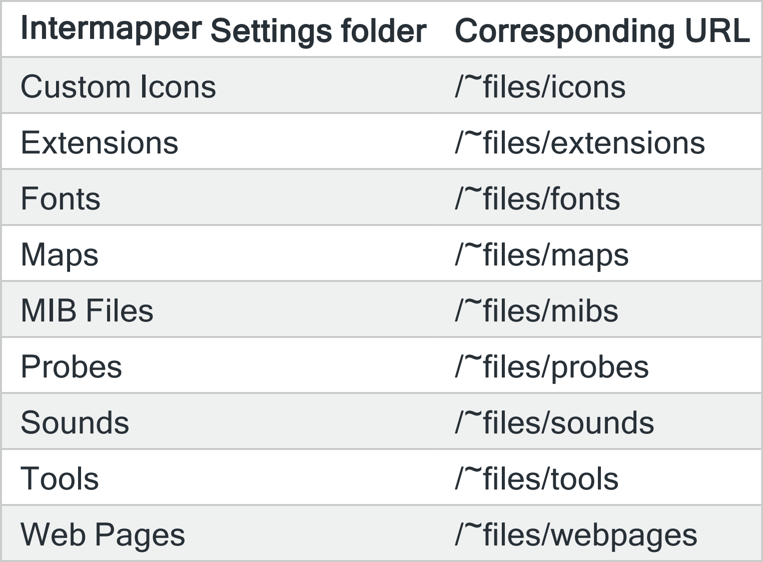
All URLs below are relative to a URL composed of the Intermapper Server address and the webport, as defined in the server settings. For example, in the following discussion, a URL of `/~files` implies the full URL (either http or https):

```
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, which 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 using an HTTP connection as described above. The URL should use the following format:

```
http://imserver:port/~files/icons/Folder/Filename.type
```

If the file type is a valid image file (jpeg or png), it is available for immediate use.

Sample curl commands (command-line) to use this facility might look like the following (and they 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 unsigned certificates.

Maps

You can import maps or map data using the HTTP API.

A sample curl command line to import a map file should use the following format:

```
$ curl --user admin:Pa55w0rd --data-binary @/path/to/local/map_file
http://imserver:port/~files/maps/map_file
```

Importing and Exporting Tables

Table-Based Import/Export

The table-based functions of the Intermapper HTTP API match same the capabilities as clicking File > Import > Data file and File > Export > Data file in Intermapper Remote Access. These import and export a number of tables of information about the monitored devices. These tables include the following information:

- Devices
- Interfaces
- Vertices
- Maps
- Notifiers
- Users
- Schema

For more information on these tables, see the *Advanced Data Import/Export* section of the Intermapper User Guide. The URLs for importing and exporting use the following format:

```
http://imserver:port/~export/tablename.format? (options)
```

The following are supported formats:

- .tab - saves as a tab-delimited text file.
- .csv - saves as a comma-delimited text file.
- .xml - saves as an XML format file.
- .html- displays as HTML directly in the browser.

The primary option is fields=. The list of valid fields are listed in the schema export. For example, the following 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 in the browser. Other examples include the following:

```
http://imserver:port/~export/devices.tab
```

provides a list of all devices on active maps as a tab-delimited file. The following URL:

```
http://imserver:port/~export/devices.tab?fields=id,name,macadress,a
ddress
```

provides a list of all devices on active maps, but only includes the ID, Name, MACAddress, and Address fields.

Importing Table-Based Data

An external program can import table information with an HTTP POST operation by including the table data as the payload. For example,

```
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 by clicking File > Import > Data File in Intermapper Remote Access.

A sample curl command line to import map data should use the following format:

```
$  curl --user admin:Pa55w0rd --data-binary @/path/to/import/file
http://imserver:port/~import/file
```

Acknowledging with HTTP

You can perform a Basic Acknowledgment of a device by issuing a POST to a URL of the following format:

```
http://imserver:port/mapid/device/deviceIMID/*acknowledge.cgi?messa
ge=URL+encoded+string
```

where:

- **mapid** is the map identifer of the corresponding map.
- **deviceIMID** is the IMID of the device you want to acknowledge.
- **message** requires a URL-encoded text string.

You can find these values by doing one of the following:

- Look in the web interface at the Status window for a device, remove the trailing !device.html text at the end, and replace the text with *acknowledge.cgi?message=.
- Review the device table using the HTTP API and obtain the MapId and IMID.

Example

The following 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?messa
ge=URL+encoded+text+string -d "dummy post data"
```

> **NOTE:**
> - This command responds with a web page, which makes sense when acknowledging through a browser, but less so when using `curl`. As a result, HTML code is sent to `curl`, which sends it to `stdio`. The returned code can be ignored, logged, or parsed as needed.
> - The curl parameter `-d "dummy post data"` forces `curl` to send the command using the HTML `POST` method, rather than the `GET` method.

Example

The following curl command retrieves the full list of devices, 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 the following 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

The Intermapper Clone facility included in the Intermapper server distribution is implemented as scripts that use the HTTP API. It is supplied in both shell script (for Linux, macOS, and Cygwin systems) and as VBScript (for Microsoft Windows systems). The Clone facility is located in one of the following locations, depending on your operating system:

- /usr/local/share/intermapper/CloneIM
- C:\ProgramData\Intermapper\InterMapper Settings\Scripts\CloneIM.vbs

These scripts provide examples of practical use of the HTTP API. For more information, see the documentation for the applicable HTTP API.

The scripts use the following URLs to copy the specified data from a remote Intermapper server to the corresponding location in the local Intermapper server (or, optionally, an alternative directory). For more information, see the "Invoking Intermapper Clone" subsection of the Reference section of the *Intermapper User Guide*, which can be viewed through the online help.

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

Retrieving Collected Data from Intermapper Reports Server

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 user interface is the easiest way to obtain 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 example perl scripts. The perl scripts require DBI and DBD::pg.

These scripts are packaged, zipped, and placed in the Fortra Downloads server, and are available in the following location:

http://download.intermapper.com/sql/sql_examples.tar.gz

Feel free to share your own with Fortra.

If you want to create your own queries to retrieve data, see Creating SQL Queries.

Intermapper Database Schemas

The most up-to-date schema for the Intermapper Database is available in the following locations:

- https://[Your Intermapper Database Server URL]:8182/~imdatabase/schemaddl.html
- http://download.intermapper.com/schema/imdatabaseschema.sql

Creating SQL Queries

You can create your own SQL queries to retrieve data from the Intermapper database. The `datasample` tables contain the 5-minute, hourly, and daily samples derived from the original data values. The recommended approach for retrieving data is to obtain it from these tables. For more information, see Intermapper Database Schemas.

You can also query individual data values, but this is much slower than querying the `datasample` tables. In Intermapper 5.4 and earlier, individual data values were stored in the `datapoints` table. They are now stored in the `datastore` table. Existing queries on the `datapoints` table must be rewritten to use the `datastore` table instead. This only applies if you have written queries in this or a related construct:

```
SELECT FROM datapoint WHERE dataset_id = 5 AND data_time BETWEEN a
AND b
```

To retrieve data from the `datastore` table, use the `load_data()` function, described below.

Using the load_data() Function

Use this function only if you have an existing query on the `datapoint` table, or if you need the individual raw values. Most of the time, you should query the `datasample` tables as described above, since they are faster and easier to access.

The load_data() function uses the following syntax:

```
load_data([dataset id],[datatime start],[datatime end])
```

For example, to retrieve data for `dataset_id = 5` between `data_times a` and `b`, use the following syntax:

```
SELECT data_time, data_value FROM load_data(5, a, b)
```

The explicit column list is not required, but it is recommended. If you use `SELECT *` rather than an explicit column list, the function returns a single column of the built-in composite-value type, containing both values. You can still reference the values from this composite data type, but you cannot treat it as you would a regular PostgreSQL column.

The `load_data()` function acts as a table source, and accepts the built-in PostgreSQL `infinity` and `-infinity` timestamps.

```
SELECT data_time, data_value
FROM load_data(1, '2011-11-09 00:00:00', 'infinity')
ORDER BY data_time
```

You can also use `UNION` to combine sources.

```
SELECT 5 as dataset_id, data_time, data_value
FROM load_data(5, '2011-11-09 00:00:00', 'infinity')
UNION SELECT 6 as dataset_id, data_time,data_value
```

```
FROM load_data(6, '2011-11-09 00:00:00', 'infinity')
UNION SELECT 14 as dataset_id, data_time, data_value
FROM load_data(14,'2011-11-09 00:00:00', 'infinity')
ORDER BY dataset_id, data_time
```

Customizing Web Pages

Intermapper comes with a set of default web page layouts and uses them to generate web pages. Read 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 folder. See Web Pages Folder for more information.

When a web request is received, Intermapper locates a corresponding file (called a target file) to use as the response. The target file is formatted according to information specified the template file. The resulting file is returned in a web browser.

Intermapper uses the following elements to control the appearance of the web pages returned from its web server:

- **Target files** - contains the main text of the various pages sent by the server.
- **Template files** - controls the overall format of the web pages.
- **Directives** - commands within files to control the formatting of the web pages.
- **Quoted links** - makes it easy to create links to other pages.
- **Macros** - elements that you can insert in your templates and target files to show blocks of useful Intermapper information.
- **Web Pages Folder** - controls which web pages are available to administrators and guests.
- **Mime Types** - associates templates or target files with specific MIME types.

Tip - The target and template files are text files. You can edit them with any text editor. On certain platforms, you must have the correct permissions to edit them.

Reloading Changed Web Page Files

Changes to these files are not applied until Intermapper reloads them.

To force Intermapper to reload the Web Page files:

1. From the **Edit** menu, click **Server Settings**.
2. From the **Server Configuration** category, click **Web Server**. The Web Server settings panel is displayed.
3. Stop and 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 can contain HTML markup if desired.

In addition to the page's text, the target file can contain the following elements:

- **Directives** - commands that describe or modify how a page is displayed.
- **Quoted Links** - provides a quick way to create a link to another page using its name, rather than specifying its full URL. If a string is in double quotation marks (" ") and the text matches the title of another Intermapper web page, a link is created.
- **Macros** - Intermapper variables that are replaced with text or formatted HTML in the final web page. You can place the macro with a static string, a device 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
remaining
text is displayed in the "body" of the page. The text may also
contain
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 specifies the title of the page to be displayed.
- The text between the double quotation marks (" ") is placed in the `<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} is replaced by the current date when the page is displayed.

Quoted Links

The text `"This is a test page"` is displayed as a link to its own page, since it is a string in quotation marks that matches the `#title` of a web page (its own). Note, too, that the text body can be a link to a page with a title of body. It is not an error if no such page exists and Intermapper displays the quoted string in place. For more information, see Quoted Links.

What Happens When a Target File Is Read?

As the target file is read, Intermapper processes the directives, expands the macros, and creates the tags for any quoted links it encounters. The web server does not insert white space or paragraph marks (such as <p>) when it encounters carriage returns.

Built-In Target Files

Intermapper provides the following built-in target files. These file names all begin with an exclamation point (!) and are required because Intermapper refers to them explicitly.

- **!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.

> **NOTE:**
> The !network.html, !device.html, !link.html, and !chart.html files are targets intended to display information about a specific network, device, link, or chart. The macros that display lists of maps, networks, devices, and charts create links to these targets. The easiest way to create custom versions of these targets is to edit them directly.

Template Files

To allow web pages to have the same look and feel, Intermapper uses template files to control page formatting. 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

The following is a simple template file that can 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, everything from the target file that is not a directive.
- **${imageref:logo.gif}** - replaced with a tag that refers to the logo.gif file in the ~GuestImages folder on the Intermapper server.
- **${include:footer.incl}** - replaced with the contents of the file called 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 pound sign (#) in the first column.

Summary of Directives

Intermapper includes the following directives that can change how pages are formatted:

#template	`#template "othertemplate.html"` A target file can specify a template file with the `#template` directive. The `#template` directive is optional. If no template file is present, Intermapper uses the file called `!template.html` as the page's template.

#title	`#title "This is a Test Page"` The text in quotation marks (" ") becomes the title of the page - it enclosed in `<title>...</title>` tags in the generated page. The `#title` directive also provides a destination for quoted links on other pages. Every target file must have a `#title` directive to give it a name. You can include a macro within the quoted text of this directive to insert the device name or other information into the title of the web page.
#alt_title	`#alt_title "Test Page"` The optional `#alt_title` directive provides a way to provide a page an alternate name that can be used with a quoted link.
#filename	`#filename "otherpage.html"` The optional `#filename` directive causes Intermapper to treat the file as if named with the quoted string. For example, a target file called xindex.html can include a directive of `#filename "!index.html"`. This causes the target file to be used in place of the file called !index.html if its version number is higher. This can be useful for debugging, or for creating alternate versions of pages.

#version

```
#version "2.1"
```

The optional `#version` directive determines which file is used when there are two or more instances of the same filename (as a result of using `#filename` directives).

The optional `#version` directive is used to break connections between several files having the same name to determine which should be used. This can be used in the following circumstances:

- 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 can 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,

  User files are not overwritten.
- Through the use of the `#filename` directive, you can have two files on disk that have similar or the same name. For example, if one file is called foo.html and another file has a different name, you can use the `#filename` directive to set the 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 the `#version` directive is not present in the file, the default version of 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. 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 open in a new window called 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 you click a map device or link.

Quoted Links

You can create a link to another page by entering the page's title in double quotation marks (" "). For example, `"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 can have the same `#title` or `#alt_title`. When this happens, Intermapper selects one of the target files. However, you cannot predict which one is selected.

Preventing a Quoted String From Becoming a Link

If you place a string in quotation marks and the string does not match another page's `#title` or `#alt_title`, Intermapper displays the quoted string as-is.

You might want text to appear in quotation marks, even when the text matches another page's `#title` or `#alt_title`. (Remember, you can create quoted links only to pages that 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:

Insert backslashes (\) in front of 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 use or require other-information that follow the colon (:). The entire macro is replaced by the appropriate text when the page is generated.

Macros can fall into one of the following categories:

- The Include Macro
- Macros that generate "content" on an Intermapper web page
- Macros that describe Intermapper and its environment
- Macros to place images onto a page
- Macros that control the interval between page refreshes
- Macros related to links and URLs

Include Macro

Your template files and target files might include other files.

${include:file-to-be-included.html} the named file is inserted into the web page. The file must be in the same folder.

Macros That Generate 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 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.

${chartlist}	outputs a sorted list of charts from the current context, one per line with each line preceded by a `<LI>` tag. You are required to supply your own `<UL>` or `<OL>` tags. Each chart title is a hyperlink to the related chart web page. Within an administrator context, **${chartlist}** generates a list of all charts. In a per-map context, **${chartlist}** generates a list of charts from the current map.
${chartname}	outputs the title of the chart related to the current web page. If you are not on a chart-related page, the output is "". This is similar to `${mapname}`.
${currentlinkoutages}	outputs a table of current interface outages. The table's column names are **Date**, **Time**, **Interface**, and **Duration**.
${currentoutages}	shows the list of current outages (devices or links that are currently in warning, alarm, or are down in the named map).
${errorstatus_orig}	outputs the original errors status report. Differences from `${errorstatus}` include the following: • `${errorstatus_orig}` does not show device alarms. • `${errorstatus}` first outputs interfaces in error, then interfaces with high utilization. `${errorstatus}`lists interfaces in random order.
${errorstatus}	shows only the devices and links that are in warning or alarm states, or down for the named map.
${fullstatus}	shows a list of all the devices and links for the map named in the URL.
${include:file-to-be-included.html}	inserts the specified file into the web page.

${maplist}	shows an HTML unnumbered list (`<UL>`) of the maps available.
${maplistwithcharts}	shows an HTML unnumbered list of the maps available, with sub-lists of the charts for each map.
${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.
${previousoutages}	shows the list of devices listed as outages but have since returned to normal.

Miscellaneous Macros That Describe Intermapper and Its Environment

${abouthtml}	Shows the About page with the current version of Intermapper.
${date}	The current date.
${deviceaddress}	The IP or AppleTalk address of the particular device. For anything that is not a device, an empty string is returned.
${deviceid}	Outputs the device identifier of the device related to the current page, in the `"gMMMM-rNN"` format. If the current page is not device-related, output "".
${devicelist_kml}	Generates a device list in KML format for use by Google Earth.
${devicelist}	Outputs a table showing the device list for the current context. The table's columns are **Status**, **Name**, **Condition**, **Date**, **Time**, **Probe**, and **Port**. Within an administrator context, **${devicelist}** generates a list of all devices. In a per-map context, **${devicelist}** generates a list of devices from the current map.
${devicename}	The DNS name or AppleTalk NBP name of the device. This is an empty string for anything that is not a device.

${httplocaladdress}	Outputs the IP address of the web server side of the connection. If the Intermapper server is multi-homed, this is the local side IP address of the current TCP connection. Use caution with this address; URLs produced using this address might break in NAT situations.
${httpremoteaddress}	The IP address of the remote browser.
${httpuserid}	The name used for authentication.
${ifadmin: ADMIN : NONADMIN }	Outputs ADMIN if the user has admin privileges. Otherwise, it outputs NONADMIN.
${imagesuffix}	Set to .png if the web client can display .png images or .jpeg images, or other supported image types.
${intermapperaddress}	The IP address of this Intermapper server.
${mapname}	The current map name.
${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 characters.
${statshtml}	Shows Intermapper's statistics: uptime, memory usage, and so on.
${telnetserverurl}	The `telnet:` URL that connects to this Intermapper Telnet server.
${time}	The arithmetic Linux time in seconds, counted from 00:00:00 UTC on 1 January 1970.
${timestamp}	The human-readable textual representation of the time.
${version}	The version of this copy of Intermapper.
${webserverurl}	The `http:` URL that connects to this Intermapper server.

Macros to Place Images On a Page

${imageref:IMAGEFILE [,tags]}	Creates an `<img ... >` tag to place an image on the page.
	For example,
	`${imageref: photo, class='grade4'}`
	outputs
	`<IMG SRC="/images/photo.gif" class='grade4'>`
	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 where the name matches the IMAGEFILE parameter. For example, if you have two files called photo.gif and photo.png, photo.gif is found first.

${imagesuffix}	Set to .png if the web client can display .png images,.jpeg images, or other supported image types.
${intermapperlogo}	Creates an `<img... >` tag that includes the Made with Intermapper logo image.

*chart	Displays a strip chart that generally has a suffix of `${chart}.${imagesuffix}` to send the desired format for the client's browser. This uses the width parameters for the chart, in pixels and the other parameters of the URL.
	Usage:
	`<IMG SRC="*chart.${imagesuffix}?${clientwidth}&${httpparams}">`
	or
	`<IMG SRC="*chart.${imagesuffix}?width=300&${httpparams}">`

*imagemap.html	Displays an HTML imagemap that corresponds to the map image. When you click in the image, it follows the links in the (automatically-generated) image map. Usage: `${include:/${httpdocument}/document/main/*imagemap.html}`

***map**	Displays an image of the devices, networks, and links (the foreground) of the selected map against a transparent background. The objects in this image match the *imagemap.html, below. Takes an option of a timestamp to provide for auto-refresh. Usage: `<img src="/${httpdocument}/document/main/*map.${imagesuffix}?${timestamp}">`
***mapbg**	Displays the background image of the selected map. This provides the customer-selected background to the map as an image. It takes an option of a timestamp to provide for auto-refresh. Usage: `<img src="/${httpdocument}/document/main/*mapbg?${timestamp}">`
***popuptext.html**	Displays the contents of the current device, network, or interface Status Window (formerly called pop-up windows) as HTML. Usage: `${include: *popuptext.html}`, generally enclosed in `<pre>...</pre>` tags.

NOTE:
The web pages combine `*map` with the `*mapbg` and `*imagemap` to create a `<div>` that superimposes all three items into a single visual unit. See `Intermapper Settings/Web Pages/PerMapHTML/map.html` for an example.

Macros That Control the Interval Between Page Refreshes

Intermapper's web server can automatically refresh a web page at a desired interval. Include these tags on your page to take advantage of this facility.

${htmlrefreshmetaoptions}	The option list that a web client can choose from. The current `${htmlrefreshmetatag}` value is selected. Note that your HTML template should supply the `<form><select>...</select></form>` surrounding this `${htmlrefreshmetaoptions}` macro.

${htmlrefreshmetatag}	Either an empty string or the previous refresh choice from the web client. (Inserts a `<meta http-equiv="refresh"...>` tag on the resulting page.)
${jsrefreshoptions}	The option list that a web client can select from, generated with JavaScript. The current `${htmlrefreshmetatag}` value is selected. Note that your HTML template should supply the `<form><select>...</select></form>`

Macros Related to Links and URLs

These macros all return a fully-escaped string, meaning that a space character is replaced with a `%20`, a question mark (?) with `%3F`, and so on.

The following 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
```

${anchor: value}
${attr}
	Sets the current anchor value.
	
	Outputs the current anchor parameter value as a set using `${anchor: value}`. If no anchor value is set, the output is "".
	
	These two macros are closely related. `${anchor}` sets the value and `${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". The ${attr} macro is used to place the attribute string in each link. Afterward, ${anchor:} sets the anchor to an empty string.

${httpclass}	The second level directory of the page requested (device, chart, link, document, or network). For example, device.
${httpdocument}	The top level directory of the page requested. Also an alias for `${mapname}`. For example, Map1.

${httpinstance}	The third level directory of the page requested. For example, 192.168.0.1%3ASNMP.
${httpinstancepath}	A concatenation of `${httpdocument}`, `${httpclass}`, `${httpinstance}` separated by forward slashes (/). For example, /Map1/device/192.168.0.1%3ASNMP.
${httpmethod}	The fourth-level part of the page requested. For example, !device.html.
${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 does not exist)
${httpparams_ endchart}** **${httpparams_ nextchart} **${httpparams_ prevchart}** **${httpparams_ startchart}** **${httpparams_ timescale}**	`${httpparams_endchart}` - Replaces the value of the endtime parameter with the last name of the chart. (For scrolling to the end of the chart.) `${httpparams_nextchart}` - Replaces the value of the endtime parameter with a new time value that effectively scrolls the chart one page into the future. `${httpparams_prevchart}` - Replaces the value of the endtime parameter with a new time value that effectively scrolls the chart one page into the past. `${httpparams_startchart}` - Replaces the value of the endtime parameter with the start time of the chart. (For scrolling to the beginning of the chart). `${httpparams_timescale: VALUE}` - Replaces the value of the timescale parameter with the specified value. 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.

${httpparams}	Outputs all the HTTP parameters from the originating GET request in their original format. If there were no parameters attached to the original request, output "". Example, given the following URL: `http://www.example.com/TestMap/?color=red&style=bold` `${httpparams}` outputs color=red&style=bold.
${httppath}	The full path to the requested file. For example, /Map1/device/192.168.0.1%3ASNMP/!device.html.
${webpageurl}	The full URL of the requested web page. For example, the full URL as shown above.

Web Pages Folder

Web target files and template files are in the Web Pages folder in the Intermapper Settings folder. Except for the folders described below, the Intermapper web server serves only files that are located in the top level of the Web Pages folder.

Overriding the Built-In Pages

Intermapper ships a single zip archive called BuiltinWebPages.zip.

To customize pages, you need to create a directory structure that matches the structure within BuiltinWebPages.zip file. Any files placed in those folder override pages of the same name in the zip archive.

Contents of BuiltinWebPages.zip

The BuiltinWebPages.zip file contains the following folders:

- AdminHTML

  This folder contains HTML templates for pages that show the overall status of the Intermapper program. People with access to these pages can also view all the separate map pages. You can access these files from the default web URL, or by using a URL in the following format:

  ```
  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 the following URL format:

```
http://intermapper.domainname.com/~error/filename.html
```

- GuestImages

  This folder contains images used by the Intermapper web server. These images can be placed in a target or template file using the `${imageref: ... }` macro.
- PerMapHTML

  This folder contains HTML templates that are used to display information about a map. To view a specific document's information, use the following URL format:

  ```
  http://intermapper.domainname.com/docname
  ```

The zip archive also contains some supporting files, including JavaScript files, located in the root of the folder.

How 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 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 can 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 the following:

```
http://intermapper.domain.com/~admin/!index.html.
```

Similarly, a request for the following:

```
http://intermapper.domain.com/docname
```

is treated like a request for the following:

```
http://intermapper.domain.com/docname/document/main/!index.html.
```

MIME Types

You can associate a template or target file suffix with MIME-type information that you want to send with the file. You can create this association by placing a file called mimetypes at the top level of the Web Pages folder.

The following 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
```

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:

  `http://.../!chart.html?height=`*`xxx`*`&width=`*`yyy`*

- Enter different time scale:

  `http://.../!chart.html?timescale=`*`XXXXX`* where the time value is in minutes.

Command Line Options for Intermapper

You can call Intermapper and Intermapper Remote Access from a command-line, and control a significant number of functions. This can be useful for automating map updates or for testing purposes.

For more information on the use of the command-line for scripting Intermapper and Intermapper Remote Access, see *Command-line Options for Intermapper* and *Command-line Options for Intermapper Remote Access*in the User Guide's *Reference* section as well as *Intermapper HTTP API* in this manual.

Intermapper Service Management for Linux Systems

Intermapper includes three services: Intermapper Server, Intermapper DataCenter, and Intermapper Flows. On each Microsoft Windows and macOS system, Intermapper Control Center (IMCC) provides a user interface client for starting and stopping Intermapper Server and Intermapper Flows. In addition, on Microsoft Windows systems, these services can be controlled from the Microsoft Windows Services application. On macOS, Intermapper Data Center (IMDC) can be started and stopped using the following commands respectively:

- `/usr/local/imdc/sbin/imdc start`
- `/usr/local/imdc/sbin/imdc stop`

On Linux systems, the three Intermapper services are controlled through systemd, which is the Linux service manager facility. For information necessary to start and stop the Intermapper services on a Linux host, using the systemd command line client (systemctl), see the *Intermapper Installation Guide* and the *Intermapper User Guide*.

This section provides additional information on the system administration of a Linux Intermapper host of the Linux service configuration.

Service Definitions

Under systemd, each Linux host service is described by a service unit file. The service unit files for the Intermapper services are in the `/usr/local/share/intermapper/units` directory. There is a file for each service (intermapperd.service, imdc.service, and imflows.service. The files are copied into the `/etc/systemd/system` systemd directory by the Intermapper installation process which also starts and enables the Intermapper service, the DataCenter service, and (optionally) the Flows service.

For information on how to read and write service definition files, see the systemd.unit(5) and systemd.service(5) Linux manual pages. Make sure you consult the version of the documentation that matches your Linux version as there can be version-specific information.

Although service definition files are written using a stanza-like format, they are case-sensitive, which is the normal Linux convention. Do not include an alias line in the service definition file that matches the service file name.

Some versions of systemd support linking service definition files into the systemd configuration area rather than copying them there, but this is not uniformly supported across target Linux versions for Intermapper. Fortra does not recommend using the systemctl link command for the Intermapper service definition files (even if your Linux system has a more evolved version of systemd) as this can cause confusion to Fortra Technical Support.

systemd Command Line Interface

The primary interface for systemd is the systemctl command line interface. For more information on the systemctl command line interface, see the systemctl(1) Linux manual page. The following commands are used by systemctl where <service> is the name of the service. For example, intermapperd.service.

- `systemctl start <service>` starts a service
- `systemctl stop <service>` stops a service
- `systemctl status <service>` reports the status of a service
- `systemctl restart <service>` stops and starts a service
- `systemctl enable <service>` enables a service on reboot
- `systemctl disable <service>` inhibits a service from restarting on reboot
- `systemctl reload <service>` requests a running service to reload its configuration
- `systemctl daemon-reload` recaches changes to the systemd configuration
- `systemctl reset-failed` clears a service failure statuses
- `systemctl list-units` displays a service status (append "-a" to include all services)

You must run these commands as a Linux super-user (root). If you try to change the systemd configuration as a non-root user, then, depending on your administration policy tool (if any), you are asked to authenticate as an appropriately authorized user before the request can be performed.

Intermapper Service Definitions

Two of the Intermapper services (Intermapper Data Center and Intermapper Flows) are configured as forking services and the Type field of their definition is set to forking. This means that they take advantage of the systemd provision to retain control of the services even if their primary processes fork and delegate their primary service function to the child process (as was required by the traditional Linux service management). This does not result in any significant loss of service management functionality and each of these services records its presence in a pid-file (a plain text file containing theprocess-id) as reflected by the PIDFile clause in its service definition.

By contrast, the Intermapper server has a service definition file that uses the default service type, which, depending on the systemd version, is either `simple` or `exec`. The `--no-daemonize` option is passed on the execution command line specified in the service definition ExecStart clause. This is consistent with earlier versions of Intermapper, but restarting the service automatically on failure (formerly delegated to an Intermapper-supplied shell script) is now handled by systemd itself through use of the `Restart`, `RestartSec`, `StartLimitInterval`, and `StartLimitBurst` clauses of the service definition file.

The Linux Intermapper host system administrator can configure the Intermapper services by editing the service definition files using the generic systemd documentation. Doing this should preserve the original content of the service unit files that are in `/usr/local/share/intermapper/units` as a reference in case you need to recover the standard configuration. Use the systemd commands described in the previous section to install and test the local configuration changes.

Index

Contacting Fortra

Please contact Fortra for questions or to receive information about Intermapper. You can contact us to receive technical bulletins, updates, program fixes, and other information via electronic mail, Internet, or fax.

Fortra Portal

For additional resources, or to contact Technical Support, visit the Fortra Community Portal at https://community.fortra.com.

For support issues, please provide the following:

- Check this guide's table of contents and index for information that addresses your concern.
- Gather and organize as much information as possible about the problem including job/error logs, screen shots or anything else to document the issue.