# HP OpenView Internet Services

# Custom Probes API Guide

Document Release Date: April 2007
Software Release Date: April 2007

# Legal Notices

## Warranty

## Restricted Rights Legend

## Copyright Notices

## Trademark Notices

## Documentation Updates

This manual's title page contains the following identifying information:

- Software version number, which indicates the software version
- Document release date, which changes each time the document is updated
- Software release date, which indicates the release date of this version of the software

To check for recent updates, or to verify that you are using the most recent edition of a document, go to:

**http://ovweb.external.hp.com/lpe/doc_serv/**

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

# Support

You can visit the HP Software Support web site at:

**www.hp.com/managementsoftware/services**

HP Software online support provides an efficient way to access interactive technical support tools. As a valued support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract.

To find more information about access levels, go to:

**www.hp.com/managementsoftware/access_level**

To register for an HP Passport ID, go to:

**www.managementsoftware.hp.com/passport-registration.html**

# Contents

# 1 Custom Probes

## Introduction

The HP OpenView Internet Services Custom Probe feature is designed to allow seamless integration and measurement logging of user implemented custom probes into the Internet Services Management Server.

‼ The Custom Probes feature is only supported with the English language version of Internet Services at this time.

## What's Included in Custom Probes

The Internet Services custom probe feature includes the following:

- This documentation which describes the APIs and the steps to implementing a custom probe
- The necessary header files and libraries
- A Custom Probes wizard for adding, updating and removing custom probe definitions into the Internet Services Configuration Manager.
- Two fully functional sample probe implementations, with full source code and Visual C++ 6.0 project files and UNIX Makefiles.

It is recommended that you read this documentation before developing your custom probes.

➤ A thorough understanding of Internet Services and the underlying data-models (in the context of probes) is required to implement a custom probe. Also C/C++ programming skills are required.

## Requirements

The Custom Probes SDK requires the following C++ compilers:

### Windows:

Microsoft Visual Studio 6.0, Service Pack 5

Required Options:

| | |
|---|---|
| /GR | enable RTTI |
| /GX | enable exception handling |
| /MD | Multithreaded DLL (use for release version) |
| /MDd | Debug Multithreaded DLL (use for debug version) |

### HP-UX:

HP aC++ Compiler C.03.33 (or higher)

### Linux:

gcc version 2.95.4 (or higher)

### Solaris:

Sun Forte 6 Update 2

Note, C++ compiler/linker are required. This is necessary since the SDK might change and/or include other C++ libraries.

# The Custom Probes Architecture

Figure 1 is a block diagram of the Internet Services architecture and within it is shown how a custom probe integrates.

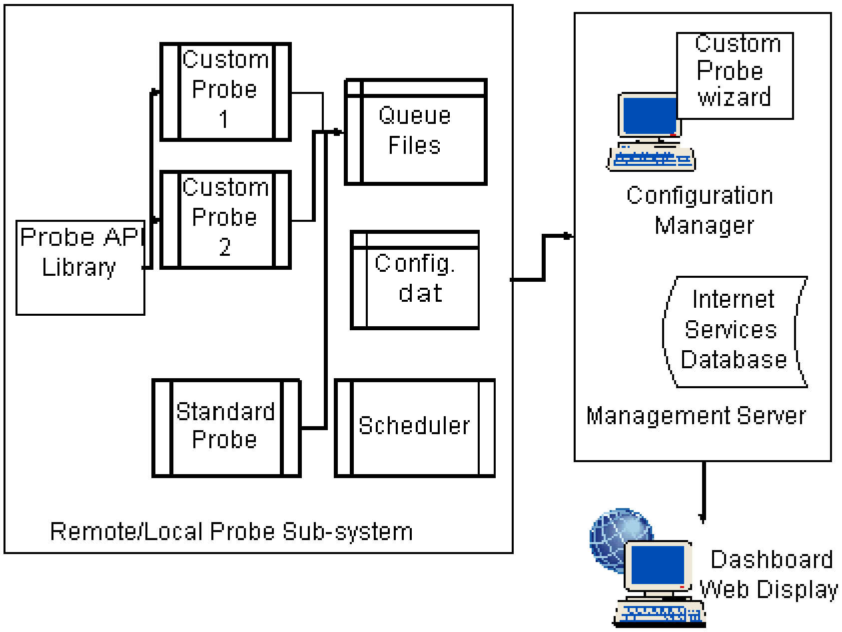


**Figure 1 Custom Probes Architecture**

Please refer to the *Internet Services User's Reference Guide* and Online help for more information on architectural data flow, probes and how Internet Services works.

# API Conventions, Libraries and Files

## Function-naming Conventions

The functions of the Internet Services APIs have consistent names that reflect the operation they perform. See Figure 2. Naming the Internet Services API Functions for an example of how the Internet Services API functions are named.

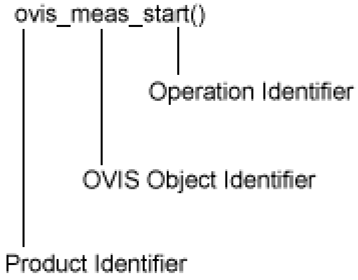


**Figure 2 Naming the OVIS API Functions**

The function names consist of the following parts:

| | |
|---|---|
| Product Identifier: | Identifies the product. In Internet Services, this is always 'ovis'. |
| OVIS Object Identifier: | Identifies the OVIS object on which the operation is to be performed. OVIS objects are shown in Table 1. |

**Table 1 OVIS Objects**

| cmdline | Probe Command Line |
|---|---|
| meas | OVIS Measurement |
| error | OVIS Error Handling |
| trace | OVIS Tracing |
| timer | OVIS Timers |

| | |
|---|---|
| Operation Identifier: | Identifies the operation which the function performs on the OVIS object. |

Note: Unless explicitly mentioned all the parameters passed to the APIs should be considered as input parameters.

## Libraries on the Management Server and Remote Probes Systems

Development of custom probes on various platforms using Internet Services custom probes requires using platform specific libraries.

Platform Specific Libraries include:

**Table 2 Platform Specific Libraries**

| **PLATFORM** | **Library** |
|---|---|
| MS Windows | OvIsApi.dll |
| HPUX | libOvIsApi.sl |
| Solaris | libOvIsApi.so |
| Linux Red Hat | libOvIsApi.so |

## Include and Lib Files

Development of custom probes on various platforms requires using platform specific header files and libraries.

Platform Specific Include Files/Libs:

**Table 3 Platform Specific Header Files/Libs**

| PLATFORM | Header File | Lib File |
|---|---|---|
| MS Windows | OvIsApi.h | OvIsApi.lib |
| HPUX | OvIsApi.h | ----- |
| Solaris | OvIsApi.h | ----- |
| Linux | OvIsApi.h | ----- |

## Makefiles

Sample make files are provided for development of custom probes on various platforms.

Platform Specific Makefiles:

**Table 4 Platform Specific Makefiles**

| PLATFORM | MakeFile |
|---|---|
| MS Windows | ProbeDummy.dsp |
| HPUX | Makefile |
| Solaris | Makefile |
| Linux | Makefile |

## Queue Files

Every call to the ovis_meas_log() function in the probe implementation should generate a queue file in the `<data_dir>\datafiles\probe\queue` folder(queue folder). If your probe makes more than one call to the ovis_meas_log() (probes with multiple transactions), check for multiple queue files in the queue folder.

The queue file subsequently gets uploaded to the Management Server at regular intervals. This is done through a regular HTTP connection. If a proper HTTP connection doesn't exist between the probe machine and the Management Server, the queue files will continue to accrue in the `<data_dir>\datafiles\probe\queue` folder on the probe machine. If left in such a state for a long tie, this could result in enormous disk space consumption on the probe machine, and the probe machine might eventually run out of disk space.

The Management Server will correctly reflect the status of all of the probes and their service targets on the configuration manager GUI and on the dashboard, if the queue files are getting uploaded to the Management Server at regular intervals.

# 2 Implementation Steps

This chapter explains the basic steps to creating and implementing a custom probe.

Please read through these steps and the detailed descriptions of the Custom Probe API calls in Chapter 3 before you begin to develop your custom probe.  Also see Chapter 4 for example source code and sample files that can be helpful in getting started building your custom probe.

# Implementing Custom Probes

## A. Steps to implement a custom probe on MS Windows

**Step 1.**

Define your probe name [type]. This name must match the probe name you enter in the Custom Probe wizard in step 2 below.

A note on probe naming convention:.

You MUST prefix your probe name with a **`C_`** (e.g., `C_PROBE_CUSTOM`). This will guarantee that your probe name will never conflict with any future changes/additions to OVIS probes.

ANYTCP
DHCP
DIAL
DNS
Exchange
FTP
HTTP
HTTPS
HTTP_TRANS
ICMP
IMAP4
LDAP
MAILROUNDTRIP
NNTP
NTP
ODBC
POP3
RADIUS
SAP
Script
SMS
SMTP
SOAP
STREAM_MEDIA
TCP
TFTP
UDP
WAP

This list is subject to change in the future.

**Step 2.**

The next step is to define your probe's input parameters and output metrics. The Internet Services Configuration Manager on the management server needs to be updated with the new probe definition (this is the SRP file) based on the parameters and metrics for your probe.

After a new SRP file is updated and loaded, on the Management Server you need to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

You can create the SRP file on the Management Server in two ways:

- Manually create an SRP file (on the Management Server) based on the parameters and metrics that your custom probe defines and manually load it into the Configuration Manager.

  On the Management Server run `reload -load <SRP file name>` to load the file.

  See the sample SRP file in SRP File Structure on page 69 to understand the format.

- Run the Custom Probes wizard on the Management Server to step through this process (`<install dir>\SDK\InternetServices\bin\probewizard.exe`). The wizard essentially writes the SRP file for you and automatically imports it into the Configuration Manager. The wizard can also be used to update or remove custom probe definitions that have previously been added. See the command to run the wizard and a screen shot below.

The following optional metric parameters cannot be added when using the Custom Probe wizard, the SRP file must be manually modified to include these parameters:

LABEL - Allows setting a local dependent label for the metric.

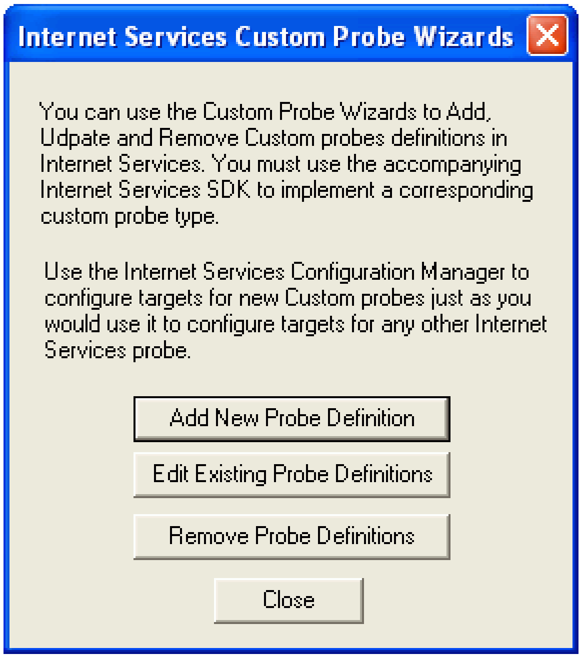
FORMAT - Used to set the display format a metric (e.g., FORMAT: 0.000 will display a number with only 3 digits after the decimal). The value for FORMAT follows the Java formatter convention. See example with the sample SRP file in Chapter 4

COMPOSITE_METRIC and COMPOSITE_ORDER - Used by the OVIS Dashboard to create a stacked bar chart. The COMPOSITE_METRIC specifies the parent metric (usually response time) and the COMPOSITE_ORDER specifies the position fo the metric within the bar chart. See example with the sample SRP file in Chapter 4.

MULTISTEP - This flag indicates whether a metric is part of a graph that shows the steps broken out for a specific metric.

Run the Custom Probes wizard on the Internet Services Management Server as follows:

```
<install dir>\SDK\InternetServices\bin\probewizard.exe
```



You can use the Custom Probe Wizards to Add, Udpate and Remove Custom probes definitions in Internet Services. You must use the accompanying Internet Services SDK to implement a corresponding custom probe type.

Use the Internet Services Configuration Manager to configure targets for new Custom probes just as you would use it to configure targets for any other Internet Services probe.

In the first dialog you can select to Add, Edit or Remove a custom probe definition. When creating a new custom probe definition follow these steps:

- Define the new probe type's Name
- Define the new probe's set of Parameters
- Define the new probe's set of Metrics
- Define the new probe's Executable name. The probe executable should start with **c_probe***.

➤ The probe name as specified in the wizard must be the same as specified in the probe in Step 1.

The probe executable should start with c_probe*.

➤ After a new SRP file is updated and loaded, on the Management Server you need to be sure to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

Note that once you complete implementing your probe and data is being collected, the graphs in the Dashboard will be available for this custom probe without requiring a special Reports Template. But to get reports in the Dashboard **Reports tab**, you must create a Report Template file which requires you to use hp OpenView Reporter A.03.00 and Crystal Decisions Crystal Reports version 8.5 or higher (www.crystaldecisions.com). See Creating Reports for Custom Probes on page 25.

**Step 3.**

Create a new folder on your system to hold the source/header files for your new custom probe. We will refer to this folder henceforth as **`probeCustom`** in this document.

**Step 4.**

Make sure you have the correct versions of these files:

`OvIsApi.h,`

`OvIsApi.lib`

These files are part of Internet Services Custom Probe feature. They should be under the `<install dir>\SDK\InternetServices\include` and `<install dir>\SDK\InternetServices\lib` folders respectively. You can check the version with the `perfstat -v` command.

**Step 5.**

To write a custom probe, in C/C++:

Implement the 'main' function body of your probe in a separate C (.c) or C++ (.cpp) source file. This source file is referred to as **`mainCustom.cpp`** in this document. Create this file in your `probeCustom` folder and add it to your probe project.

The `OvIsApi.h` file needs to be included in the `mainCustom.cpp` implementation file, the probe needs to be linked to the `OvIsApi.lib` file. The most recent release of the `OvIsApi.dll` will be installed in the probe directory by the Internet Services Installer.

You can either copy these two files into your newly created `probeCustom` folder or add the `<install dir>\SDK\InternetServices\include` and `<install dir>\SDK\InternetServices\lib` paths to your project settings to make Developer Studio look for those files there.

If you are using Visual C++ 6.0:

Add the `<install dir>\Sdk\InternetServices\include` path in

`Project->Settings->C/C++->->Preprocessor->Additional include directories`

and the `<install dir>\Sdk\InternetServices\lib` path in

`Project->Settings->Link->Input->Additional library path`

**Step 6.**

If you decide to use the Custom Probe's command line parsing routines, declare the options table, specifying your probe specific command line parameters.

[The options table is declared as an array of string pointers each on of which holds a switch name, that your custom probe could be passed on the command line.]

Note that the following command line switches are reserved by Internet Services and should not be specified in the options table.

```
-customer "customername"
-servicename "servicename"
-serviceid "10;10;10"
-interval 300
-timeout 30
-host "hostname"

-print
-dump
-trace 1
```

These switches are internal to Internet Services and are automatically handled by the Custom Probe's command line parsing routines, when passed on the probe's command line. When passed on the command line, their values should be in the format shown above.

**Step 7.**

If your custom probe is to support tracing and error logging, decide on the probe Error Logging and Tracing scheme for your probe. Your custom probe can either trace and log errors into the default Internet Services trace and error log files, or you may choose to make the probe trace and log errors in your own trace and error log files.

If you decide to use your own trace and error log files, declare string literals for the names of your custom error and trace files.

➤ Steps 1 - 7 ensure that your custom probe now has the appropriate settings and declarations to use the custom probe API to write measurements to the Internet Services Management Server.

**Step 8**

The next step is to implement a timing model for the probe.

A custom probe must implement a timing model by which it self-timouts after a certain time interval. This is necessary since all Internet Services probes (including custom probes) are scheduled by the scheduler to run periodically. If probes do not terminate at regular intervals, the probe system may eventually be rendered unstable due to stray probe processes.

The time interval for timeout is typically passed to the probe through one of the standard input parameters -TIMEOUT. Use the `get_ovis_parameter()` function to retrieve the timeout passed to the probe. Ideally the probe's timing model should terminate the probe in a time interval slightly less than what was specified through the -TIMEOUT parameter. When being scheduled for execution through the scheduler, if the probe does not self-timeout at the -TIMEOUT interval, the OVIS scheduler will force termination of the probe.

Refer to the accompanying `probeExchange` sample probe's source code, for an example of how to implement a timing model in a probe.

**Step 9**

Build your custom probe using your compiler and linker.

➤ See the section on Configuring and Deploying a Custom Probe on page 24 for the final steps to a working probe.

## B. Steps to implement a custom probe on UNIX

Steps to follow to write a Custom Probe on UNIX are similar to that of Windows NT/2000:

**Step 1.**

Define your probe name [type]. This name must match the probe name you enter in the Custom Probe wizard in step 2 below.

A note on probe naming convention:

You MUST prefix your probe name with a **`C_`** (e.g., `C_PROBE_CUSTOM`). This will guarantee that your probe name will never conflict with any future changes/additions to OVIS probes.

The following probe names are reserved and are currently used by standard Internet Services probes and **MUST NOT** be used to name your custom probe.

ANYTCP
DHCP
DIAL
DNS
Exchange
FTP
HTTP
HTTPS
HTTP_TRANS
ICMP
IMAP4
LDAP
MAILROUNDTRIP
NNTP
NTP
ODBC
POP3
RADIUS
SAP
Script
SMS
SMTP
SOAP
STREAM_MEDIA
TCP
TFTP
UDP
WAP

This list is subject to change in the future.

**Step 2.**

The next step is to define your probe's input parameters and output metrics. The Internet Services Configuration Manager on the management server needs to be updated with the new probe definition (this is the SRP file) based on the parameters, and metrics for your probe.

After a new SRP file is updated and loaded, on the Management Server you need to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

You can create the SRP file on the Management Server in two ways:

- Manually create an SRP file (on the Management Server) based on the parameters, and metrics that your probe defines and manually import it into the Configuration Manager.

  On the Management Server run `repload -load <SRP file name>` to load the file.

  See the sample SRP file in SRP File Structure on page 69 to understand the format.

- Use the Custom Probes wizard to step through this process (`<install dir>\SDK\InternetServices\bin\probewizard.exe`). . The wizard essentially writes the SRP file for you and automatically imports it into the Configuration Manager. The wizard can also be used to update or remove custom probe definitions that have previously been added. . See the command to run the wizard and a screen shot below.

  The following metric parameters cannot be added when using the Custom Probe wizard, the SRP file must be manually modified to include these parameters:

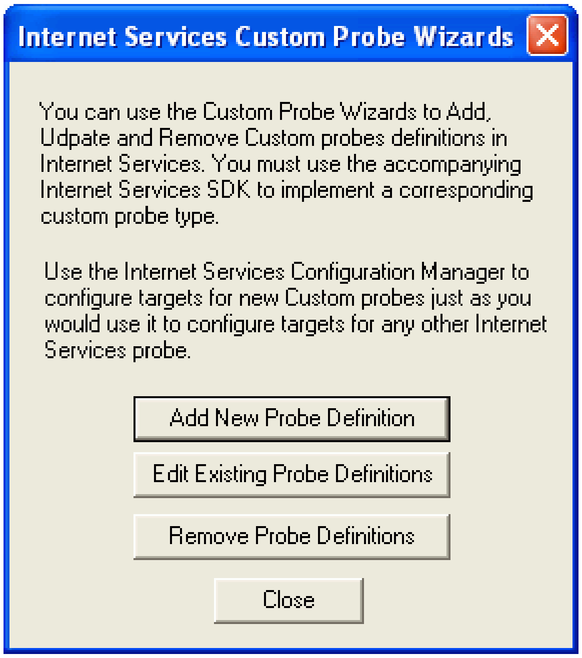
  LABEL - Allows setting a local dependent label for the metric.

  COMPOSITE_METRIC and COMPOSITE_ORDER - Used by the OVIS Dashboard to create a stacked bar chart. The COMPOSITE_METRIC specifies the parent metric (usually response time) and the COMPOSITE_ORDER specifies the position fo the metric within the bar chart. See examples with the sample SRP files in Chapter 4.

  MULTISTEP - This flag indicates whether a metric is part of a graph that shows the steps broken out for a specific metric.

Run the Custom Probes wizard on the Internet Services Management Server as follows:

```
<install dir>\SDK\InternetServices\bin\probewizard.exe
```



You can use the Custom Probe Wizards to Add, Udpate and Remove Custom probes definitions in Internet Services. You must use the accompanying Internet Services SDK to implement a corresponding custom probe type.

Use the Internet Services Configuration Manager to configure targets for new Custom probes just as you would use it to configure targets for any other Internet Services probe.

In the first dialog you can select to Add, Edit or Remove a custom probe definition. In creating a new custom probe definition follow these steps:

- Define the new probe type's Name
- Define the new probe's set of Parameters

- Define the new probe's set of Metrics
- Define the new probe's Executable name. The probe executable should start with **c_probe***.

The probe name as specified in the wizard must be the same as specified in the probe in Step 1.

The probe executable should start with c_probe*.

After a new SRP file is updated and loaded, on the Management Server you need to be sure to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

Note that once you complete implementing your probe and data is being collected, the graphs in the Dashboard will be available for this custom probe without requiring a special Reports Template. But to get reports in the Dashboard **Reports tab**, you must create a Report Template file which requires you to use hp OpenView Reporter A.03.00 and Crystal Decisions Crystal Reports version 8.5 or higher (www.crystaldecisions.com). See Creating Reports for Custom Probes on page 25.

**Step 3.**

Create a new folder on your system to hold the source files and header files for your new custom probe. We will refer to this folder henceforth as **`probeCustom`** in this document.

**Step 4.**

Make sure you have the correct versions of the files:

`OvIsApi.h,`
`libOVisApi.so` or `libOvIsApi.sl` (for Solaris)

These files are part of Custom Probes. They should be under the `opt/OV/VPIS/probes` and `opt/OV/lib` folders respectively. You can use the **what** command to determine the version and compare this to the list of files and versions in the OVIS release notes. For example

```
# what libOvIsApi.sl
libOvIsApi.sl:
      libOvIsApi  A.04.00.00  1/05.01   HP-UX 11.0 - 11.20
```

**Step 5.**

To write a custom probe, in C/C++:

Implement the main function body of your probe in a separate C (.c) or C++ (.cpp) source file. This source file is referred to as **`mainCustom.cpp`** in this document. Create this file in your `probeCustom` folder and add it to your probe project.

The `OvIsApi.h` file needs to be included in the `mainCustom.cpp` implementation file, the probe needs to be linked to the `OvIsApi.so/OvIsApi.sl` file. The most recent release of the `OvIsApi.sl/OvIsApi.sl` files will be installed in the `/opt/OV/lib` directory by the Internet Services Installer.

**Step 6.**

If you decide to use the Custom Probe's command line parsing routines, declare the options table, specifying your probe specific command line parameters.

[The options table is declared as an array of string pointers each on of which holds a switch name, that your custom probe could be passed on the command line.]

Note that the following command line switches are reserved by Internet Services and should not be specified in the options table.

```
-customer "customername"
-servicename "servicename"
-serviceid "10;10;10"
-interval 300
-timeout 30
-host "hostname"

-print
-dump
-trace 1
```

These switches are internal to Internet Services and are automatically handled by the Custom Probe's command line parsing routines, when passed on the probe's command line. When passed on the command line, their values should be of the format as show above.

**Step 7.**

If your custom probe is to support tracing and error logging, decide on the probe Error Logging and Tracing scheme for your probe. Your custom probe can either trace and log errors into the default Internet Services trace and error log files, or you may choose to make the probe trace and log errors in your own trace and error log files.

If you decide to use your own trace and error log files, declare string literals for the names of your custom error and trace files.

➤ Steps 1 - 7 ensure that your custom probe now has the appropriate settings and declarations to use the custom probe API to write measurements to the Internet Services Management Server.

**Step 8**

The next step is to implement a timing model for the probe.

A custom probe must implement a timing model by which it self-timouts after a certain time interval. This is necessary since all Internet Services probes (including custom probes) are scheduled by the scheduler to run periodically. If probes do not terminate at regular intervals, the probe system may eventually be rendered unstable due to stray probe processes.

The time interval for timeout is typically passed to the probe through one of the standard input parameters -TIMEOUT. Use the `get_ovis_parameter()` function to retrieve the timeout passed to the probe. Ideally the probe's timing model should terminate the probe in a time interval slightly less than what was specified through the -TIMEOUT parameter. When being scheduled for execution through the scheduler, if the probe does not self-timeout at the -TIMEOUT interval, the OVIS scheduler will force termination of the probe.

Refer to the accompanying `probeExchange` sample probe's source code, for an example of how to implement a timing model in a probe.

**Step 9.**

Build the Custom Probe. The probe can be built using plain command line commands. See the following for an example of plain command line commands:

```
 #g++ -I/opt/OV/VPIS/probes -c mainDummy.cpp
 #g++ -o probeDummy mainDummy.o -Wl,-rpath -Wl,/opt/OV/lib -L/opt/OV/lib
-lOvIsApi
```

Alternatively create your Makefile to build the probe.

A sample Makefile is shown below.

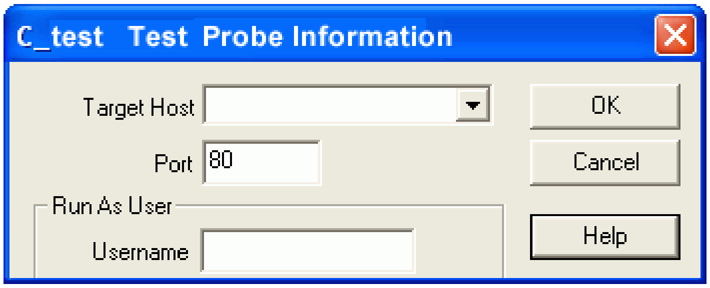
```
# Sample Makefile for a dummy probe using shared custom probe api library

# for RedHat Linux 6.0 or later
#
# Usage:
# make probeDummy

OVIS_PROBE_OBJS = mainDummy.o
OVIS_CUST_LIB_N = OvIsApi
OVIS_CUST_LIB_E = .so

OVIS_SHLIB_PATH = /opt/OV/lib
OVIS_INCLU_PATH = /opt/OV/VPIS/probes

OVIS_LIBS = -l$(OVIS_CUST_LIB_N)
OVIS_LIB_LINK_SW = -Wl,-rpath -Wl,$(OVIS_SHLIB_PATH)
-L$(OVIS_SHLIB_PATH)

OVIS_CFLAGS = -I$(OVIS_INCLU_PATH)
OVIS_CC = g++

probeDummy: $(OVIS_PROBE_OBJS) $(OVIS_SHLIB_PATH)/
lib$(OVIS_CUST_LIB_N)$(OVIS_CUST_LIB_E) Makefile
        $(OVIS_CC) -o $@ $(OVIS_PROBE_OBJS) $(OVIS_LIB_LINK_SW)
$(OVIS_LIBS)

.SUFFIXES : .o .cpp

.cpp.o:
        $(OVIS_CC) $(OVIS_CFLAGS) -c $<

clean:
        rm $(OVIS_PROBE_OBJS)
```

➤ See the section on Configuring and Deploying a Custom Probe on page 24 for the final steps to a working probe.

# Configuring and Deploying a Custom Probe

Once your custom probe is has been fully implemented and its definition added to the Configuration Manager, you can create probes with this probe type using the Internet Services Configuration Manager. In the Configuration Manager follow the same steps as you would for a standard probe to configure customer, service groups, service targets, services level objectives, service level agreements and define the location of the probe system. Be sure to save your configuration.

Note that when you create a probe with this custom probe type you can specify Run As User in the Service Target Information dialog box. This allows the probe to run as a specific user as opposed to the account that runs the OVIS scheduler.



Target Host

Port 80

Username

Also note that the custom probe configuration information can be automatically deployed to the probe system as with a standard probe. See the *Internet Services User's Reference Guide* or the Configuration Manager online help for more information on deploying probes to UNIX and Windows NT/2000 systems.

After you have configured service targets for this custom probe type, you can deploy the custom probe implementation (source code) as follows:

- On Windows systems (local or remote) copy your probe binary into the `<install dir>/probes` folder
- On UNIX systems copy your probe binary into the `/opt/OV/VPIS/probes` directory. See the *Internet Services User's Reference Guide* for more information.

## Updating a Custom Probe

Updating a custom probe involves one of the following scenarios:

1. Updating the probe implementation (source code) but keeping its input (command line) parameters and output metrics the same.
2. Updating the probe implementation (source code) so as to change its input parameters and/or metrics.

In case (1) you just need to redeploy the updated probe implementation to one or more probe locations.

In case (2) you need to update the probe definition using the custom probe wizard to reflect changes in input/output parameters and metrics and redeploy the updated probe implementation to one or more probe locations.

# Creating Reports for Custom Probes

The graphs in the Dashboard will be available for this custom probe without requiring you to create report templates.

If you want to create reports (viewed in the Reports workspace of the Dashboard) for your custom probes you need to use hp OpenView Reporter A.03.60 (or higher) and Crystal Decisions Crystal Reports version 10.0 (or higher) (www.crystaldecisions.com).

Use Crystal Reports to create the custom report and hp OpenView Reporter to configure the report to be viewed in Internet Services. Documentation on setting up reports to be generated and viewed is provided in the Reporter Concepts Guide in *Step 6: Add the Report Definition to Reporter*. Also refer to the Reporter online help topic *Add report definition* for details.

A sample report template (`a_IOps_Dummy.rpt`) for the Dummy Probe, can be found on the Management Server under the `<install dir>\sdk\InternetServices\examples\Report Template Files/` folder.

To integrate this into OVIS do the following:

1. Copy the report template file (`a_IOps_Dummy.rpt`) under the `<install dir>\data\reports\iops` folder on the Management Server.
2. Edit the `reload_C_DUMMY_PROBE.SRP` file, which can be found under the `<install dir>\sdk\InternetServices\examples\SRP Files` folder to add the following section. Make sure you match the PROBENAME with the REPORT name prefixed by IOPS_.

   ```
   REPORT:      IOPS_C_DUMMY_PROBE
           CATEGORY:       190 Internet Services
           ALL_TEMPLATE:   reports\IOps\a_iops_DUMMY.rpt
           DESCRIPTION:    DUMMY - Dummy Service
           MAXTIME:        10
           FAMILY:         "Internet Services"
           END_REPORT:

   GROUPREPORT:  IOPS_C_DUMMY_PROBE
           GROUP:          ALL
           END_GROUPREPORT:
   ```

3. Reload the SRP file into the OVIS Configuration Manager by running the following:

   ```
   reeload -load reeload_C_DUMMY_PROBE.SRP
   ```

After a new SRP file is updated and loaded, on the Management Server you need to be sure to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

4. Let the dummy probe run overnight. Next day the nightly report for the dummy probe should show up under the Reports tab of the Internet Services Dashboard.

To integrate a report for your custom probe do the following:

1. To integrate a custom report template for your custom probe, create an appropriate report template file using Crystal Reports, (similar to `a_IOps_Dummy.rpt`), and put it in the `<install dir>/data/reports/iops/` folder.
2. Use hp OpenView Reporter to add your custom report. Be sure to set the following:

   ```
   REPORT = IOPS_<probe name>
   CATEGORY = 190 Internet Services
   ```

```
HTML_DIRECTORY = webpages\<a_custom report_1>
```

Where `<a_custom report_1>` is the report name in the webpages relative directory. Refer to the Reporter documentation for how to do this.

3. Let your custom probe run overnight. Next day the nightly report for your custom probe should show up under the Reports workspace of the Internet Services Dashboard.

# Troubleshooting Your Custom Probes

How do I verify that measurements have been written by the probe?

Every call to the `ovis_meas_log()` function in the probe implementation should generate a queue file in the `<data_dir>\datafiles\probe\queue` folder (queue folder). If your probe makes more than one call to the `ovis_meas_log()` (probes with multiple transactions), check for multiple queue files in the queue folder.

The queue file subsequently gets uploaded to the Management Server at regular intervals. This is done through a regular HTTP/S connection. If a proper HTTP/S connection doesn't exist between the probe system and the Management Server, the queue files will continue to accrue in the `\<data_dir>\datafiles\probe\queue` folder on the probe system. If left in such a state for a long time, this could result in enormous disk space consumption on the probe system, and the probe system might eventually run out of disk space.

The Management Server will correctly reflect the status of all of the probes and their service targets in the Configuration Manager status display and in the Dashboard, if the queue files are getting uploaded to the Management Server at regular intervals.

# 3 Custom Probes API

## The Application Programming Interface

Internet Services comes with a set of Application Programming Interfaces (APIs) that support development of Custom Probes to probe user specific services and forward measurements back to the Internet Services Management Server.

This chapter describes the Internet Services custom probes API data structures and the API calls. The APIs primarily provide functionality for the following:

- command line parsing
- probe measurement (initializing, starting, logging, stopping the probe measurement process, and getting/setting probe metrics)
- probe tracing
- error logging and data logging to the OVIS Management Server.

Chapter 2 describes the steps to implementing a custom probe. Chapter 4 gives you examples of the makefile, SRP file and sample code.

The documentation assumes you have a good understanding and working knowledge of OVIS and C/C++ programming.

Please read the documentation on all the API calls before using them to develop a custom probe.

### API for Command Line Parsing

An Internet Services probe is typically invoked with a set of command line switches and corresponding values. These command line switches and associated values are the primary input to the probe. The command line parsing APIs provide an easy to use set of functions to parse the command line passed to the probe and later retrieve values of the individual switches, as needed.

For proper functioning of these routines, the command line switches and values passed to the probe must be separated by one or more blank spaces. For example: `probeDummy -host "xyz.com" -availability "80" -print`

The command line routines differentiate between command line switches and command line values in the following way.

A command line switch must be prefixed by a "-". Strings passed on the command line without this prefix are interpreted as command line values. It is recommended that command line values be enclosed in quotes. From the following example, you can see which are command line switches and which are command line values in the table below.

```
>robeDummy -host "xyz.com" -availability "80" -print
```

| Command Line Switch | Command Line Value |
|---|---|
| -host | "xyz.com" |
| -availability | "80" |
| -print | |

It is recommended that you use these command line parsing routines in your probe code to parse the command line. Doing so has several advantages, and it also simplifies your probe code substantially. However if your probe requires command line parsing capabilities that are beyond the scope of these routines, you can implement your own command line parsing code in the probe.

## Data Structures

Opaque List structure to hold command line parameter values:

A generic list structure is used to hold command line parameter values. This structure is an opaque structure, exported through the `ovis_cmdoptions void` pointer in `OvIsApi.h`. Use the Custom Probes APIs to set and retrieve values from it.

```
OVIS_API void * OVIS_CMDOPTIONS
```

Opaque Data structure to hold probe metrics:

An opaque data structure is used to hold probe metrics. Use the Custom Probes APIs to set and retrieve values from this structure.

```
OVIS_API void * OVIS_PARAMETRICS
```

## ovis_cmdline_parse()

**Syntax:**

```
int ovis_cmdline_parse(int argc, char* argv[],
                        int optc, char* optv[],
                      OVIS_CMDOPTIONS cmdoptions)
```

**Description:**

Use this function to parse the command line. The function parses the command line, looks for the command line parameters supported by the probe and stores their respective values into the list pointed by the `cmdoptions` parameter, for later use. Values for individual parameters can later be retrieved by calling the `ovis_cmdline_getpvalue()` function, and passing it the `cmdoptions` list that was populated by the `ovis_cmdline_parse()` function.

**Parameters:**

[Input]
*argc*: Specifies the count of the arguments passed on the command line
*argv[]*: Array of string pointers wherein each element points to a parameter passed on the command line.

*optc*: Specifies the count of the switches supported by the probe.

*optv[]*: List of string pointers wherein each element points to a switch supported by the probe.

[Output]
*cmdoptions*: Pointer to a structure of type OVIS_CMDOPTIONS.

**Return Value:**

An Integer indicating whether the initialization was successful or not. Non-zero if successful, zero if failed.

## ovis_cmdline_getpvalue()

**Syntax:**

char* ovis_cmdline_getpvalue(OVIS_CMDOPTIONS cmdoptions, const char* param)

**Description:**

This function should always be called after a call to the `ovis_cmdline_parse()` is made. Use this function to retrieve parameter values for different command line parameter that your probe supports.

**Parameters:**

*cmdoptions*: Pointer to the list of type `ovis_list` that holds the command line parameters. This list is populated by a call to the `ovis_cmdline_parse()` function.

*param*: Specifies the name of the parameter whose value is to be returned.

**Return Value:**

A string pointer pointing to the value of the parameter requested. NULL if the parameter was not passed to the probe.

## ovis_is_print()

**Description:**

This function is used to check if the probe was invoked with a `-print` command line option. If yes, the probe should handle the switch and print out its output on the stdout.

**Parameters:**

None.

**Return Value:**

Non-zero if the probe was invoked with the `-print` command line option, else zero.

## ovis_is_dump()

**Description:**

This function is used to check if the probe was invoked with a -`dump` command line option. If yes, the probe should handle the switch and dump out its output in a dump file. The recommended dump file format is `hostname.PROTOCOL`.

For example:

```
>probeX -host "xyz.com" -availability "80" -dump
```

should generate a dump file named `xyz.com.X`. This follows the recommended dump file format of hostname.PROTOCOL that all OVIS probes follow.

**Parameters:**

None.

**Return Value:**

Non-zero if the probe was invoked with the -`dump` command line option, else zero.

## ovis_is_trace()

**Description:**

This function is used to check if the probe was invoked with a `-trace` command line option. If yes, the probe should use the Custom Probes trace supporting APIs for tracing in the trace file.

**Parameters:**

None.

**Return Value:**

Non-zero if the probe was invoked with the `-trace` *`1`* command line option, else zero.

## API for Initializing, Starting, Logging and Stopping Measurements

These APIs provide a set of functions that are used to log probe metrics to the Internet Services Management Server.

An Internet Services probe gathers measurement metrics by probing the appropriate host/ web service and then logs the measurements to the Internet Services Management Server.

Data logging has to be first initialized and then finally ended. In between the initialization and end, a probe logs data one or multiple times based on whether it is a single transaction probe or a multiple transaction probe.

The API can be used to implement either a single transaction probe in which only one set of metrics are written to the Internet Services Management Server at a time, or multiple transaction probe, where a probe writes more than one set of metrics to the Internet Services Management Server during a single run.

In a single transaction probe the process of logging probe metrics logically involves starting the logging process, logging the data, and stopping the logging process. A multiple transaction probe iterates this logical sequence multiple times.

Each data log results in the creation of a temporary queue file on the probe system, which is later uploaded at a scheduled time to the local/remote Internet Services Management Server.

Appropriate memory allocations are done by `ovis_meas_init()` function. Default values are then assigned to the probe metrics at the start of the log process by the `ovis_meas_start()` function. The `ovis_set_long()`, `ovis_set_double()`, and `ovis_set_string()` functions are later used to actually set the proper values to the probe metrics. The `ovis_log_data()` then logs data into the temporary queue file and completes the logging process. The `ovis_meas_end()` function deallocates memory allocations done by the `ovis_meas_init()` function.

## ovis_meas_init()

**Syntax:**

int ovis_meas_init(const char* probename, OVIS_PARAMETRICS *meas)

**Description:**

Call this function once to initialize the probe, and the Internet Services data structure prior to calling the `ovis_meas_start()` function.

**Parameters:**

[Input]

*probename*: Specifies the name [type] of the probe.

[Output]

*meas*: Pointer to a pointer to the opaque OVIS_PARAMETRICS data structure.

**Return Value:**

An Integer indicating whether the initialization was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_start()

**Syntax:**

int ovis_meas_start(OVIS_PARAMETRICS meas)

**Description:**

This function initializes the probe metrics with default values. The function should be called each time before making a call to any of the Custom Probe timer APIs and the `ovis_meas_log()` function to log the probe metrics.

**Parameters:**

[Output]

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure to hold measurement metrics.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_log()

**Syntax:**

int ovis_meas_log(OVIS_PARAMETRICS meas)

**Description:**

This function logs measurement data contained in the OVIS_PARAMETRICS data structure (pointed to by the parameter *meas*) to the Internet Services Management Server. The function should be called after each successful completion of an `ovis_meas_set*()` function where the measurement metrics are stored into the OVIS_PARAMETRICS data structure.

The `ovis_meas_log()` call results in the creation of a temporary queue file on the probe system which is then uploaded to the Management Server at a scheduled time.

This call should be followed either by a call to the `ovis_meas_end()` function to indicate the end of data logging, or another call to the `ovis_meas_start()` function to restart another iteration of the logging function for multiple transaction probes.

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure that holds measurement metrics.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_end()

**Syntax:**

int ovis_meas_end(OVIS_PARAMETRICS meas)

**Description:**

This function is called to indicate the end of the probe session. The function should be called only once to end the process of measuring/logging of the probe metrics. The function also stops all active metric measurement timers, frees and resets them to zero.

WARNING: No further OVIS API function calls should be made. The result of calling any of the Custom Probe API functions after making a call to `ovis_meas_end()` is undefined and will cause unspecified results.

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure that holds measurement metrics.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## API for Getting/Setting Probe Metrics

These APIs provide a set of functions that can be used to individually get or set probe metrics.

Internet Services incorporates support for a standard set of well-defined default performance metrics and up to 8 user-defined metrics (typically set by the probe developer). These metrics are listed in the table below.

Listed in Table 5 below, are the performance metrics (standard and user defined) that should typically be set by the probe developer.

**Note:** All the OVIS_METRIC_* metrics need to be explicitly set. If you use the command line parsing API `ovis_cmdline_parse()` to parse the probe's command line, all of the OVIS_PARA_* metrics in the table will be automatically set by the API. If you don't use the command line parsing API to parse the probe's command line, it is your responsibility to set all of the OVIS_PARA_*/OVIS_METRIC_* metrics explicitly.

A call to the `ovis_meas_start()` function assigns default values to all of these metrics if values haven't been set for one or more of them. See Table 5 for default values.

**Note:** For most services, the metric OVIS_METRIC_TARGET and the parameter OVIS_PARA_HOST remain the same, however for some services the OVIS_METRIC_TARGET may be required to be different from the OVIS_PARA_HOST.

The metric OVIS_METRIC_TARGET is assigned the same value as the HOST by default through the call to the `ovis_meas_start()` function. The value assigned is the one that was passed on the command line. You can later call the `ovis_meas_set_string()` API with the OVIS_METRIC_TARGET or OVIS_PARA_HOST id to set the HOST or TARGET to a different value. In some cases, you might explicitly decide to set the HOST to some other value (by calling `ovis_meas_set_string()` API) after a call to `ovis_meas_start()`. It is then up to you to also update the TARGET accordingly.

### Table of Metric/Parameter Identifier.

**Table 5 Measurement Metric Identifiers**

| Metric/Parameter Identifier | Data Type | Description | Default Value |
|---|---|---|---|
| OVIS_PARA_CUSTOMER | String | customer name | "Unspecified" |
| OVIS_PARA_SERVICENAME | String | service name | "Unspecified" |
| OVIS_PARA_HOST | String | target host name (see the note above) | "Unspecified" |
| OVIS_PARA_INTERVAL | Long | interval in seconds | 300 |
| OVIS_METRIC_AVAILABILITY | Long | availability could be 0 (for unavailable) or 1 (for available) | 0 |

**Table 5 Measurement Metric Identifiers**

| | | | |
|---|---|---|---|
| OVIS_METRIC_SETUPTIME | double | DNS + network connection setuptime in seconds. This time represents the time it took to establish connection with the server before sending the first protocol request. | 0 |
| OVIS_METRIC_RESPONSETIME | double | total response time in seconds | 0 |
| OVIS_METRIC_TRANSFERTPUT | double | transfertput KBytes/Second | 0 |
| OVIS_METRIC_1 | double | user defined metric 1 | 0 |
| OVIS_METRIC_2 | double | user defined metric 2 | 0 |
| OVIS_METRIC_3 | double | user defined metric 3 | 0 |
| OVIS_METRIC_4 | double | user defined metric 4 | 0 |
| OVIS_METRIC_5 | double | user defined metric 5 | 0 |
| OVIS_METRIC_6 | double | user defined metric 6 | 0 |
| OVIS_METRIC_7 | double | user defined metric 7 | 0 |
| OVIS_METRIC_8 | double | user defined metric 8 | 0 |
| OVIS_METRIC_TIME | Long | Time at measurement instance | 0 |
| OVIS_METRIC_TIMEZONE | Long | Timezone of the probe system | 0 |
| OVIS_METRIC_PROBESYSTEM | String | Probe system name | "Unknown" |
| OVIS_METRIC_PROBENAME | String | Probe Name | "Unknown" |

**Table 5 Measurement Metric Identifiers**

| | | | |
|---|---|---|---|
| OVIS_METRIC_TRANSID | long | transaction id should be -1 for single transaction probe and indicates the transaction number for a multiple transaction probe. | -1 |
| OVIS_METRIC_IPADDR | string | Target IP address | "Unresolved" |
| OVIS_METRIC_TARGET | string | Probe target (see the Note above) | "Unspecified" |
| OVIS_PARA_SERVICEID | string | serviceid - format serviceId;serviceTargetId;probeId; | 0;0;0 |

## ovis_meas_set_long()

**Syntax:**

```
int ovis_meas_set_long(OVIS_PARAMETRICS meas,
                       int meas_parametric_id,
                       long value)
```

**Description:**

This function is called to set the value of a metric/parameter (long type) as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_set_long(parametrics, OVIS_METRIC_AVAILABILITY,
lAvailability);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5).

*value*: Value (long) of the parameter to be set as specified by the *meas_parametric_id*.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_set_double()

**Syntax:**

```
int ovis_meas_set_double(OVIS_PARAMETRICS meas,
                         int meas_parametric_id,
                         double value)
```

**Description:**

This function is called to set the value of a metric/parameter (double type) as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_set_double(parametrics, OVIS_METRIC_SETUPTIME, fSetupTime);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5).

*value*: Value of the parameter to be set as specified by the *meas_para_id*.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_set_string()

**Syntax:**

```
int ovis_meas_set_string(OVIS_PARAMETRICS meas,
                          int meas_parametric_id,
                          char *value)
```

**Description:**

This function is called to set the value for a metric (string type) as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_set_string(parametrics, OVIS_METRIC_TARGET, szTarget);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5).

*value*: Value of the parameter to be set as specified by the *meas_para_id*.

**Return Value:**

An Integer indicating whether the function was successful or not. Non-zero if successful, zero if failed.

## ovis_meas_get_long()

**Syntax:**

long ovis_meas_get_long(OVIS_PARAMETRICS meas, int meas_parametric_id)

**Description:**

This function is called to get the value of a metric as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_get_long(meas, OVIS_METRIC_AVAILABILITY);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5).

**Return Value:**

A long type metric as specified by the meas_parametric_id. NULL if no value has been previously set.

## ovis_meas_get_double()

**Syntax:**

double ovis_meas_get_double(OVIS_PARAMETRICS meas, int meas_parametric_id)

**Description:**

This function is called to get the value of a metric as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_get_double(meas, OVIS_METRIC_SETUPTIME);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5)..

**Return Value:**

A double value of the parameter as specified by the meas_parametric_id. NULL if no value has been previously set.

## ovis_meas_get_string()

**Syntax:**

char *ovis_meas_get_string(OVIS_PARAMETRICS meas, int meas_parametric_id)

**Description:**

This function is called to get the value of a metric as specified by the *meas_parametric_id* parameter. For example:

```
ovis_meas_get_string(meas, OVIS_METRIC_TARGET);
```

**Parameters:**

*meas*: Pointer to the OVIS_PARAMETRICS opaque structure.

*meas_parametric_id*: Parameter/metric ID (see the metric identifier in table 5).

**Return Value:**

A pointer (char *) to the value of the parameter as specified by the meas_parametric_id. NULL if no value has been previously set.

# API for Tracing

The tracing APIs provide a set of functions that can be used to trace various probe conditions into a trace file. A typical OVIS probe writes trace statements into a trace file indicating the various states that it goes through while being executed. Trace logs are extremely helpful in troubleshooting and debugging probe executions.

The degree of importance and detail of a trace statement is determined by a trace level. Certain trace statements with fine granular details about probe execution may not be necessary at all times and can unneccessarily clutter the trace file.

The trace level is determined by a setting on the Internet Services Management Server. Each trace statement is traced by the probe with a specific trace level in mind. For example, a setting of trace level 5 on the Management Server makes the probe trace only those statements that have a level 5 or lower. The higher the trace level, the more granular and detailed the trace information. Table 6 lists the various trace levels. Based on the information you need, you can decide on the appropriate trace statements and level.

## Table of Trace Levels

**Table 6 Trace Levels**

| | | |
|---|---|---|
| OVIS_TRACE_LEVEL_OFF | trace level 0 | OFF |
| OVIS_TRACE_LEVEL_1 | trace level 1 | Minimum |
| OVIS_TRACE_LEVEL_2 | trace level 2 | |
| OVIS_TRACE_LEVEL_3 | trace level 3 | |
| OVIS_TRACE_LEVEL_4 | trace level 4 | |
| OVIS_TRACE_LEVEL_5 | trace level 5 | High |
| OVIS_TRACE_LEVEL_6 | trace level 6 | |
| OVIS_TRACE_LEVEL_7 | trace level 7 | |
| OVIS_TRACE_LEVEL_8 | trace level 8 | |
| OVIS_TRACE_LEVEL_9 | trace level 9 | Maximum |

## ovis_trace_init()

**Syntax:**

```
int ovis_trace_init(int trace_level,
                    const char* prog_name,
                     char* trace_file)
```

**Description:**

This function initializes tracing with a default trace level (5) and a default trace file (trace.log). Internet Services needs to be initialized before calling any of the tracing APIs that can be used to trace various probe conditions to the Internet Services trace file.

**Parameters:**

*Itrace_level*: Specifies the initial trace level. The trace level can be changed using the `ovis_trace_set_level()` API.

*prog_name*: Specifies the name of the executable module that is using the Trace engine. Typically it is the probe executable name.

*trace_file*: Specifies the trace file name. Should be set to NULL to use the default trace file. The default trace file is located under the `<data_dir>\log\trace.log` folder. If the *trace_file* parameter passed to `ovis_trace_init()` is not NULL, it should contain the fully qualified path of the custom trace file.

**Return Value:**

An integer indicating whether the initialization succeeded or not. Non-zero if initialization succeeded, zero if failed.

## ovis_trace_set_level()

**Syntax:**

int ovis_trace_set_level(int trace_level)

**Description:**

This function sets the existing trace level to a new value.

**Parameters:**

*Itrace_level*: Specifies the new trace level.

**Return Value:**

This API function returns the previous trace level.

## ovis_trace()

**Syntax:**

int ovis_trace(const char* format, ...)

**Description:**

This function logs a trace statement into the Internet Services trace file. The format of the trace statement can be specified by the user through the *format* string. The API takes a variable number of parameters based on the format string.

If the number of parameters don't match with the format statement, the API fails to log the statement into the trace file and returns a zero.

**Parameters:**

*format*: Format of the trace statement.

One or more trace parameters.

**Return Value:**

An integer indicating whether the trace was written to the trace file or not. Non-zero if successful, zero if failed.

## ovis_trace_l()

**Syntax:**

int ovis_trace_l(int trace_level, const char* format, ...)

**Description:**

This function logs a trace statement into the trace file. Just as the `ovis_trace()` function, the format of the trace statement can be specified by the user through the *format* string. In addition `ovis_trace_l()` takes one more parameter, namely the *trace_level*. The function only logs the trace statement if the current trace level happens to be greater than or equal to the trace level as specified by the *trace_level* parameter.

Use this function to conditionally log traces in the trace file.

**Parameters:**

*trace_level*: Minimum Trace Level at which the trace statement should be written.

*format*: Format of the trace statement.

**Return Value:**

An integer indicating whether the trace was written to the trace file or not. Non-zero if successful, zero if failed.

## API for Error Reporting

The error reporting API provides a set of functions that can be used to log various error conditions into an error log file.  A typical probe writes error logs into a log file indicating error conditions encountered while executing.  Error logs are extremely helpful in troubleshooting and debugging probe executions.

The error logs can be either written into the standard OVIS error log file, or a custom log file, or simply printed on stdout.  The destination of an error log is determined by a flag passed to the error logging API.

Table 7 lists the various possible error destinations.

### Table of Error Destinations:

**Table 7 Error Destinations**

| **Destination ID** | **Destination** |
|---|---|
| OVIS_ERR_DST_OVISLOG | log errors to OVIS error log file |
| OVIS_ERR_DST_CUSTOMLOG | log error to user defined error log file |
| OVIS_ERR_DST_STDERR | log error to stderr |

## ovis_error_init()

**Syntax:**

int ovis_error_init(int dst, const char* prog_name, char* error_file)

**Description:**

This function initializes the error handling. This is necessary before making calls to the subsequent error logging APIs that can be used to log various probe error conditions to the OVIS error log file (`<data_dir>\log\probe\error.log`).

The *dst* parameter can be used to specify more than one destination by using a combination of one or more of the three predefined flags. For example: specifying the *dst* as

```
ovis_error_init( OVIS_ERR_DST_OVISLOG | OVIS_ERR_DST_STDERR,
"program_name")
```

Will make Internet Services log errors at two places (OVIS Error log file and stderr) simultaneously.

**Parameters:**

*dst*: Specifies the destination for error messages. Error messages can be sent to one or more of three different destinations, as specified by this parameter.

1. OVIS Error log file.
2. Stderr
3. User specified error log file.

*prog_name*: Specifies the name of the probe that reported the error.

*error_file*: Specifies the user specified error log file. Ignored if *Dst* does not contain OVIS_ERR_DST_CUSTOMLOG.

**Return Value:**

An integer indicating whether the initialization succeeded or not. Non-zero if initialization succeeded, zero if failed.

## ovis_err_set_output_dst())

**Syntax:**

int ovis_err_set_output_dst(int dst)

**Description:**

This function sets a new destination for error message logs. Error messages can be directed to any of one or more (by using logical OR conditions) of the three destinations, as specified by the *Dst* flag.

**Parameters:**

*dst*: Specifies the new destination for error messages.
Error messages can be sent to one or more of three different destinations, as specified by this parameter.

1. OVIS Error log file.
2. Stderr.
3. User specified error log file.

**Return Value:**

An integer returns the previous error destination.

## ovis_error_out()

**Syntax:**

```
int ovis_error_out(int error_code,
                    char severity,
                    int sys_errno,
                    const char* source_file,
                    int line_no,
                    const char* format,
                      ...)
```

**Description:**

This function outputs an error message indicating the error code, severity of the error, the source file name and the source line number, as to where the error occurred. Additionally a custom error message can be outputted through the *format* parameter.

**Parameters:**

*error_code*: Specifies the error code. Error codes are user defined.

*severity*: Specifies the severity of the error as follows:

| | |
|---|---|
| OVIS_ERR_SEV_WARNING | for warning. |
| OVIS_ERR_SEV_ERROR | for error. |

*sys_errno*: Use this to pass any error code that the might have been returned by the system as a result of a system call failure. This will provide for additional diagnostics and help in troubleshooting the probe.

*source_file*: Specifies the source file name in which the error occurred.

*line_no*: Specifies the exact source code line number within the source file.

*format*: Format of error message string.

**Return Value:**

An integer indicating whether the error message was logged successfully or not. Non-zero indicates success, zero indicates failure.

# API for Time Keeping

The time keeping APIs provide a set of functions to perform various timing measurements. Most Internet Services probes report one or more timing metric. Having a set of time keeping APIs makes it easier to make timing measurements in probes.

Timers are initialized by the `ovis_timer_start()` function, A unique timer ID is returned by this function. This ID can be later used to stop the timer at a desired instance of time and later to retrieve the measured time interval.

The time keeping APIs allow for the initialization of up to 256 concurrent timers. The accuracy and resolution of the timers are OS dependent and are the same as the OS's own time accuracy and resolution.

## ovis_timer_start()

**Description:**

This function initializes a new Timer. The timer acts like a stopwatch that can be used to measure timing related probe metrics.

**Parameters:**

None

**Return Value:**

A non-zero integer if the function is successful, zero if failed.

The return value is the ID of the newly initialized timer.

## ovis_timer_stop()

**Syntax:**

int ovis_timer_stop(int timer_id)

**Description:**

This function stops an existing timer. Each timer has a unique TimerID associated with it. The Time Keeping APIs can be used to initialize concurrent timers for the purpose of measuring timing metrics.

**Parameters:**

*timer_id*: ID of the timer that is to be stopped.

**Return Value:**

An integer, non-zero if Timer stop succeeded, else zero.

## ovis_timer_elapsed()

**Syntax:**

int ovis_timer_elapsed(int timer_id)

**Description:**

This function returns the elapsed time for an existing timer, since it was started. Each timer has a unique *timer_id* associated with it (returned by `ovis_timer_start()`). The `ovis_timer_elapsed()` function should be passed the appropriate timer_id.

**Parameters:**

*timer_id*: ID of the timer that's elapsed time is to be returned.

**Return Value:**

Elapsed time in milliseconds. An integer, non-zero if successful, -1 if failed.

## Typical Implementation Steps and the API

Coding for a typical custom probe follows the following logical sequence:

1. Parse the Command Line
2. Probe the intended Service
3. Make performance measurements
4. Log measurements to the Internet Services Management Server
5. Quit

These logical steps can be implemented using the Custom Probe API as follows

### Parse the Command Line

```
ovis_parse_cmdline()
```

### Make performance measurements

```
ovis_meas_init()
ovis_meas_start()
ovis_timer_start()
ovis_timer_stop()
ovis_timer_elapsed()
```

### Log measurements to the Internet Services Management Server/print measurements out to stdout

```
ovis_meas_get_long()
ovis_meas_get_double()
ovis_meas_get_string()
ovis_meas_set_long()
ovis_meas_set_double()
ovis_meas_set_string()
ovis_meas_log()
```

### Quit

```
ovis_meas_end()
```

### In addition, the following APIs can be used for error handling and tracing.

```
ovis_error_init()
ovis_error_set_output_dst()
ovis_error_out()
ovis_trace_init()
ovis_trace_set_level()
ovis_trace()
ovis_trace_l()
```

Chapter 2 provides detailed implementation steps. Chapter 4 provides a working sample custom probe implemented using the custom probe API.

# 4 Examples

This chapter includes the following examples:

- Sample Probes
- Sample Code
- Sample Makefile
- Typical SRP File

# Sample Probes

Two fully functional sample probe implementations are provided with the Custom Probes, with full source code and Visual C++ 6.0 project files/UNIX Makefiles. The sample code in the next section is based on the Dummy probe.

1. Dummy probe
   1. `ProbeDummy.dsp` - on the Management Server under the `<install dir>\Sdk\InternetServices\examples\probeDummy` folder
   2. UNIX Makefiles - on the Management Server under the `<install dir>\Sdk\InternetServices\examples\probeDummy` folder
2. Exchange probe
   1. `ProbeExchange.dsp` - on the Management Server under the `<install dir>\Sdk\InternetServices\examples\probeExchange` folder
   2. Not Available on UNIX.

To build the probes, on Windows, simply load the project files (`probeDummy.dsp` and `probeExchange.dsp`) into MS Visual Studio 6.0 (or higher) and build the projects.

For UNIX, copy the files in the `<install dir>\Sdk\InternetServices\examples\probeDummy` folder and the `<install dir>\Sdk\InternetServices\include\OvisApi.h` file to a UNIX system and run `make -f Makefile.<platform>`.

Once built, to integrate the sample probes into an existing install of Internet Services, please refer to the `readme.txt` files under each of the sample folders.

# Sample Code (Windows/UNIX)

This section shows a skeletal C++ sample probe implementation using the Custom Probe APIs. The sample code is based on the Dummy probe provided with the custom probes feature.

```
/* mainCustom.cpp */

#include "OvIsApi.h"

#define probe_name "C_CUSTOM_PROBE"

/* Options table for command line parsing */
const char *optv[] = {
"parameter1",
"parameter2",
"parameter3"
};

int main(int argc, char* argv[])

{

  /* Structure to hold probe metrics */
  OVIS_PARAMETRICS parametrics;

  /* List to hold command line parameters */
  OVIS_CMDOPTIONS cmdoptions;

  int i_TraceLevel = 0;

  int Timer_SetupTime, Timer_ResponseTime = 0; /* Timer ids */
  long lElapsedTime = 0;
  int i = 0;

  long lAvailability = 0;
  double fSetupTime = 0;
  double fResponseTime = 0;
  double fTransferTput = 0;
  double dwSleepTime = 0;
  int optc = sizeof( optv ) / sizeof( optv[0] );

  /* Parse the command line */
  ovis_parse_cmdline(argc, argv, optc, optv, cmdoptions);

  /* Error and trace initialization */
  ovis_error_init(OVIS_ERR_DST_OVISLOG, "probeCustom", 0);

  if(ovis_is_trace())

  {
    if(ovis_get_paramvalue("trace", cmdoptions))
      i_TraceLevel = atoi(ovis_get_paramvalue("trace", cmdoptions));

     ovis_trace_init(i_TraceLevel, "probeCustom", TraceFile);

  }

  /* Initialize measurement structure */
  ovis_meas_init(probe_name, &parametrics);

  /* Start the measurement process */
  ovis_meas_start(parametrics);
```

```
 Timer_SetupTime = ovis_timer_start();
 Timer_ResponseTime = ovis_timer_start();
  /* Setup code here */
……
…….
…….
   ovis_timer_stop(Timer_SetupTime);
  /* Probe transaction code here */
……
……
……
 ovis_timer_stop(Timer_ResponseTime);


 /* Compute metric Values */
 /* Set lAvailability */
/* Set  fSetupTime */
/* Set fResponsTime */
 ovis_meas_set_long(parametrics, OVIS_METRIC_AVAILABILITY,  lAvailability);
 ovis_meas_set_double(parametrics, OVIS_METRIC_SETUPTIME,  fSetupTime);
 ovis_meas_set_double(parametrics, OVIS_METRIC_RESPONSETIME,  fResponseTime);
 ovis_meas_set_double(parametrics, OVIS_METRIC_TRANSFERTPUT,  fTransferTput);
 /* Log Metrics to the Management Server */
 ovis_meas_log(parametrics);
 /* Re-Start data logging */
 ovis_meas_start(parametrics);
 Timer_SetupTime = ovis_timer_start();
 Timer_ResponseTime = ovis_timer_start();




/* Setup code here */
……
…….
…….
   ovis_timer_stop(Timer_SetupTime);
  /* Probe transaction code here */
……
……
……
 ovis_timer_stop(Timer_ResponseTime);
```

```
  /* Re-compute metric Values */
  /* Set lAvailability */
  /* Set  fSetupTime */
  /* Set fResponsTime */

  ovis_meas_set_long(parametrics, OVIS_METRIC_AVAILABILITY,  lAvailability);
  ovis_meas_set_double(parametrics, OVIS_METRIC_SETUPTIME,  fSetupTime);
  ovis_meas_set_double(parametrics, OVIS_METRIC_RESPONSETIME,  fResponseTime);
  ovis_meas_set_double(parametrics, OVIS_METRIC_TRANSFERTPUT,  fTransferTput);

  /* Log Metrics to the Management Server */
  ovis_meas_log(parametircs);

  /* End of probe measurements */
  ovis_meas_end(parametrics);

  return 0;

}
```

# Sample Makefile

A sample Makefile is shown below.

```
# Sample Makefile for a dummy probe using shared custom probe API library

# for RedHat Linux 6.0 or later
#
# Usage:
# make probeDummy

OVIS_PROBE_OBJS = mainDummy.o
OVIS_CUST_LIB_N = OvIsApi
OVIS_CUST_LIB_E = .so

OVIS_SHLIB_PATH = /opt/OV/lib
OVIS_INCLU_PATH = /opt/OV/VPIS/probes

OVIS_LIBS = -l$(OVIS_CUST_LIB_N)
OVIS_LIB_LINK_SW = -Wl,-rpath -Wl,$(OVIS_SHLIB_PATH) -L$(OVIS_SHLIB_PATH)

OVIS_CFLAGS = -I$(OVIS_INCLU_PATH)
OVIS_CC = g++

probeDummy: $(OVIS_PROBE_OBJS) $(OVIS_SHLIB_PATH)/
lib$(OVIS_CUST_LIB_N)$(OVIS_CUST_LIB_E) Makefile
	$(OVIS_CC) -o $@ $(OVIS_PROBE_OBJS) $(OVIS_LIB_LINK_SW) $(OVIS_LIBS)

.SUFFIXES : .o .cpp

.cpp.o:
	$(OVIS_CC) $(OVIS_CFLAGS) -c $<

clean:
	rm $(OVIS_PROBE_OBJS)
```

# SRP File Structure

A typical SRP file has the following structure.

Note that the Probe Metrics parameters LABEL, COMPOSITE_METRIC, COMPOSITE_ORDER, MULTISTEP are optional and are not included in the SRP file generated with the Custom Probe wizard. They can only be added manually into the SRP file directly.

LABEL - Allows setting a locale dependent label for the metric.

FORMAT - Used to set the display format a metric (e.g., FORMAT: 0.000 will display a number with only 3 digits after the decimal). The value for FORMAT follows the Java formatter convention.

COMPOSITE_METRIC and COMPOSITE_ORDER - Used by the OVIS Dashboard to create a stacked bar chart. The COMPOSITE_METRIC specifies the parent metric (usually response time) and the COMPOSITE_ORDER specifies the position fo the metric within the bar chart.

MULTISTEP - This flag indicates whether a metric is part of a graph that shows the steps broken out for a specific metric.

On the Management Server run `repload -load <SRP file name>` to load the SRP file you updated or created.

After a new SRP file is updated and loaded, on the Management Server you need to run `ovc -restart ovtomcatA` and exit the current Dashboard session.

```
PROBENAME: C_PROBE_CUSTOM
  DESCRIPTION: CUSTOM - Custom Probe
  PROBEMETRICLIST: IOPS_CUSTOM
  IDENTIFIER: URL
  INSTANCEID: URL
  DEFAULT_TARGET: /
  DEFAULT_PORT: 80
  PROBE:          probeCustom
  TRANSPORT:HTTP
  PARAMETER1: username
  PARAMETER2: password
  END_PROBENAME:

PROBEMETRICS: IOPS_CUSTOM

  METRIC: AVAILABILITY
  LABEL:  Availability
  UNITS:  Percent
  FORMAT: ###,0,100
  DEFAULT_CONDITION: >
  DEFAULT_SERVICE_LEVEL:90.000
  DEFAULT_WARNING: 90.000
  DEFAULT_BASELINE: 80.000
  DEFAULT_DURATION: 600
  DEFAULT_MESSAGE: CUSTOM Service for <TARGET> is unavailable

  METRIC: RESPONSE_TIME
  LABEL: Response Time
  UNITS: Seconds
```

```
   DEFAULT_CONDITION: <
   DEFAULT_SERVICE_LEVEL:2.0
   DEFAULT_WARNING:2.0
   DEFAULT_MINOR:4.0
   DEFAULT_MAJOR:6.0
   DEFAULT_CRITICAL:10.0
   DEFAULT_BASELINE:80.000
   DEFAULT_DURATION: 600
   DEFAULT_MESSAGE: CUSTOM Service RESPONSE_TIME is slow (<VALUE> vs
<THRESHOLD>) on <TARGET>

   METRIC: SETUP_TIME
   LABEL: Setup Time
   UNITS: Seconds
   DEFAULT_CONDITION: <
   DEFAULT_WARNING: 3.000
   DEFAULT_BASELINE: 80.000
   DEFAULT_DURATION: 600
   DEFAULT_MESSAGE: CUSTOM Service SETUP_TIME is slow  (<VALUE> vs
<THRESHOLD>) on <TARGET>

   METRIC:    DNS_SETUP_TIME
   LABEL: DNS Setup Time
   STDMETRIC:  M1
   UNITS:  Seconds
   COMPOSITE_METRIC: RESPONSE_TIME
   COMPOSITE_ORDER: 1

   METRIC:    CONNECT_TIME
   LABEL: Connect Time
   FORMAT: 0.000
   STDMETRIC:  M2
   UNITS:  Seconds
   COMPOSITE_METRIC: RESPONSE_TIME
   COMPOSITE_ORDER: 2

   METRIC:    SERVER_RESP_TIME
   LABEL: Server Response Time
   STDMETRIC:  M3
   UNITS:  Seconds
   COMPOSITE_METRIC: RESPONSE_TIME
   COMPOSITE_ORDER: 3

   METRIC:    TRANSFER_TIME
   LABEL: Transfer Time
   STDMETRIC:  M4
   UNITS:  Seconds
   COMPOSITE_METRIC: RESPONSE_TIME
   COMPOSITE_ORDER: 4

   END_PROBEMETRICS:

METRICLIST: IOPS_PROBE_DATA
   SOURCE:    IOPS
   CLASS:    IOPS_PROBE_DATA
   RETAINDAYS:    30
   END_METRICLIST:
```

```
METRICS: IOPS_PROBE_DATA
   METRIC: CUSTOMER_NAME
   METRIC: SERVICE_NAME
   METRIC: AVAILABILITY
   METRIC: SETUP_TIME
   METRIC: RESPONSE_TIME
   END_METRICS:

REPORT: IOPS_C_PROBE_CUSTOM
   CATEGORY: 190 Internet Services
   ALL_TEMPLATE: reports\IOps\a_IOps_Custom.rpt
   HTML_DIRECTORY: webpages\a_iops_custom
   DESCRIPTION: CUSTOM Report
   MAXTIME: 10
   FAMILY: "Internet Services"
   END_REPORT:

GROUPREPORT:  IOPS_C_PROBE_CUSTOM
   GROUP: ALL
   END_GROUPREPORT:
```

The above definition would create a stacked bar chart for RESPONSE_TIME with the following metrics broken out: DNS_SETUP_TIME, CONNECT_TIME, SERVER_RESP_TIME, TRANSFER_TIME.

In the above definition, the AVAILABILITY metric is always formatted as 3 digits and with a max range of 0-100 (###,0,100). The CONNECT_TIME metric is formatted with 3 digits after the dot (e.g., 10.123).

The MULTISTEP flag indicates whether a metric is part of a graph that shows the steps broken out for a specific metric.