HP Performance Agent

For the Windows® Operating System

Software Version: 4.70

Tracking Your Transactions

Document Release Date: September 2007
Software Release Date: September 2007

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices

Trademark Notices

Adobe® is a trademark of Adobe Systems Incorporated.

Intel486 is a U.S. trademark of Intel Corporation.

Java™ is a U.S. trademark of Sun Microsystems, Inc.

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

Netscape™ and Netscape Navigator™ are U.S. trademarks of Netscape Communications Corporation.

Oracle® is a registered U.S. trademark of Oracle Corporation, Redwood City, California.

Oracle Reports™, Oracle7™, and Oracle7 Server™ are trademarks of Oracle Corporation, Redwood City, California.

OSF/Motif® and Open Software Foundation® are trademarks of Open Software Foundation in the U.S. and other countries.

Pentium® is a U.S. registered trademark of Intel Corporation.

SQL*Net® and SQL*Plus® are registered U.S. trademarks of Oracle Corporation, Redwood City, California.

Support

You can visit the HP Software Support web site at:

www.hp.com/go/hpsoftwaresupport

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:

http://h20230.www2.hp.com/new_access_levels.jsp

To register for an HP Passport ID, go to:

http://h20229.www2.hp.com/passport-registration.html

Contents

1 What is Transaction Tracking?

Improving Performance Management

You can improve your ability to manage system performance with the transaction tracking capabilities of HP Performance Agent.

As the number of distributed mission-critical business applications increases, application and system managers need more information to tell them how their distributed information technology (IT) is performing.

- Has your application stopped responding?
- Is the application response time unacceptable?
- Are your service level objectives (SLOs) being met?

The transaction tracking capabilities of Performance Agent allow IT managers to build in end-to-end manageability of their client/server IT environment in business transaction terms. With Performance Agent, you can define what a business transaction is and capture transaction data that makes sense in the context of *your* business.

For applications instrumented with the standardized Application Response Measurement (ARM) API calls, Performance Agent provides extensive transaction tracking and end-to-end management capabilities across multi-vendor platforms.

HP Performance Manager in this document refers only to versions 4.0 and later. The name Performance Manager 3.x is used throughout this document to refer to the product that was formerly known as PerfView.

Benefits of Transaction Tracking

- Provides a **client** view of elapsed time from the beginning to the end of a transaction.
- Provides transaction data
- Helps you manage Service Level Agreements (SLA).

These topics are discussed in more detail in the remainder of this section.

Client View of Transaction Times

Transaction tracking provides you with a **client** view of elapsed time from the beginning to the end of a transaction. When you use transaction tracking in your IT environment, you see the following benefits:

- You can accurately track the number of times each transaction executes.
- You can see how long it takes for a transaction to complete, rather than approximating the time as happens now.
- You can correlate transaction times with system resource utilization.
- You can use your own business deliverable production data in system management applications, such as data used for capacity planning, performance management, accounting, and charge-back.
- You can accomplish application optimization and detailed performance troubleshooting based upon a real unit of work (your transaction), rather than representing actual work with abstract definitions of system and network resources.

Transaction Data

When Application Response Measurement (ARM) API calls have been inserted in an application to mark the beginning and end of each business transaction, you can then use the following resource and performance monitoring tools to monitor transaction data.

- Performance Agent provides the registration functionality needed to log, report, and detect alarms on transaction data. Performance Agent is required for transaction data to be viewed in PerfView , or by exporting the data from Performance Agent log files into files that can be accessed by spreadsheet and other reporting tools.
- PerfView graphs performance data for short-term troubleshooting and for examining trends and long-term analysis.
- PerfView or the HP Operations Manager Message Browser allow you to monitor alarms on service level compliance.

Individual transaction metrics are described in Chapter 5.

Service Level Objectives

Service level objectives (SLOs) are derived from the stated service levels required by business application users. SLOs are typically based on the development of the service level agreement (SLA). From SLOs come the actual metrics that Information Technology resource managers need to collect, monitor, store, and report on to determine if they are meeting the agreed upon service levels for the business application user.

An SLO can be as simple as monitoring the response time of a simple transaction or as complex as tracking system availability.

A Scenario: Real Time Order Processing

Imagine a successful television shopping channel that employs hundreds of telephone operators who take orders from viewers for various types of merchandise. Assume that this enterprise uses a computer program to enter the order information, check merchandise availability, and update the stock inventory. We can use this fictitious enterprise to illustrate how transaction tracking can help an organization meet customer commitments and SLOs.

Based upon the critical tasks, the customer satisfaction factor, the productivity factor, and the maximum response time, resource managers can determine the level of service they want to provide to their customers.

Chapter 6 of this manual contains a pseudocode example of how ARM API calls can be inserted in a sample order processing application so that transaction data can be monitored with Performance Agent.

Requirements for Real Time Order Processing

To meet SLOs in the real time order processing example described above, resource managers must keep track of the length of time required to complete the following critical tasks:

- Enter order information
- Query merchandise availability
- Update stock inventory

The key customer satisfaction factor for customers is how quickly the operators can complete their order.

The key productivity factor for the enterprise is the number of orders that operators can complete each hour.

To meet the customer satisfaction and productivity factors, the response times of the transactions that access the inventory database, adjust the inventory, and write the record back must be monitored for compliance to established SLOs. For example, resource managers may have established an SLO for this application that 90 percent of the transactions must be completed in 5 seconds or less.

Preparing the Order Processing Application

ARM API calls can be inserted into the order processing application to identify transactions for `inventory response` and `update inventory`. Note that the ARM API calls must be inserted by application programmers *prior* to compiling the application. See Chapter 6, Transaction Tracking Examples for an example of an order processing program (written in pseudocode) that includes ARM API calls that define various transactions.

For more information on instrumenting applications with ARM API calls, see the *Application Response Measurement 2.0 API Guide*.

Monitoring Transaction Data

When an application that has been instrumented with ARM API calls is installed and running on your system, you can monitor transaction data with Performance Agent or PerfView.

... with Performance Agent

Using Performance Agent, you can collect and log data for named transactions, monitor trends in your SLOs over time, and generate alarms when SLOs are exceeded. Once these trends have been identified, Information Technology costs can be allocated based on transaction volumes. Performance Agent alarms can be configured to activate a technician's pager, so that problems can be investigated and resolved immediately. For more information, see Chapter 8, "Performance Alarms," in the *HP Performance Agent for Windows NT: User's Guide*.

Performance Agent is required for transaction data to be viewed in PerfView .

... with PerfView

PerfView receives alarms and transaction data from Performance Agent. For example, you can configure Performance Agent so that when an order processing application takes four seconds to check stock, PerfView receives an alarm and sends a warning to the resource manager's console as an alert of potential trouble.

In PerfView, you can select **TRANSACTION** from the Class List window for a data source, then graph transaction metrics for various transactions. For more information, see PerfView online Help.

Guidelines for Using ARM

Instrumenting applications with the ARM API requires some careful planning. In addition, managing the environment that has ARMed applications in it is easier if the features and limitations of ARM data collection are understood. Here is a list of areas that could cause some confusion if they are not fully understood.

1. In order to capture data, the OV Performance Transaction Manager service (`ttd`), OV Performance collector (`scopent`), and OV Performance Measurement Interface service (`midaemon`) must be running. (See page 25 and page 28.)
2. To capture any newly-defined transaction names, re-read the transaction configuration file, `ttdconf.mwc`. (See page 29 and page 30.)
3. Performance Agent, user applications, and the Transaction Manager service (`ttd`) must be restarted to capture any *new* or modified transaction ranges and service level objectives (SLOs). (See page 31)
4. Strings in user-defined metrics are ignored by Performance Agent. Six non-string user-defined metrics are logged. (See page 68.)
5. Using dashes in the transaction name has limitations if you are specifying an alarm condition for that transaction. (See page 47.)
6. Performance Agent will only show the first 60 characters in the application name and transaction name. (See page 24.)
7. Limit the number of unique transaction names that are instrumented. (See page 37.)
8. Do not allow ARM API function calls to affect the execution of an application from an end-user perspective. (See page 26.)
9. Use shared library `libarm32.lib` for linking. (See the ARM Library Information on page 41)

2 How Transaction Tracking Works

Technical Overview

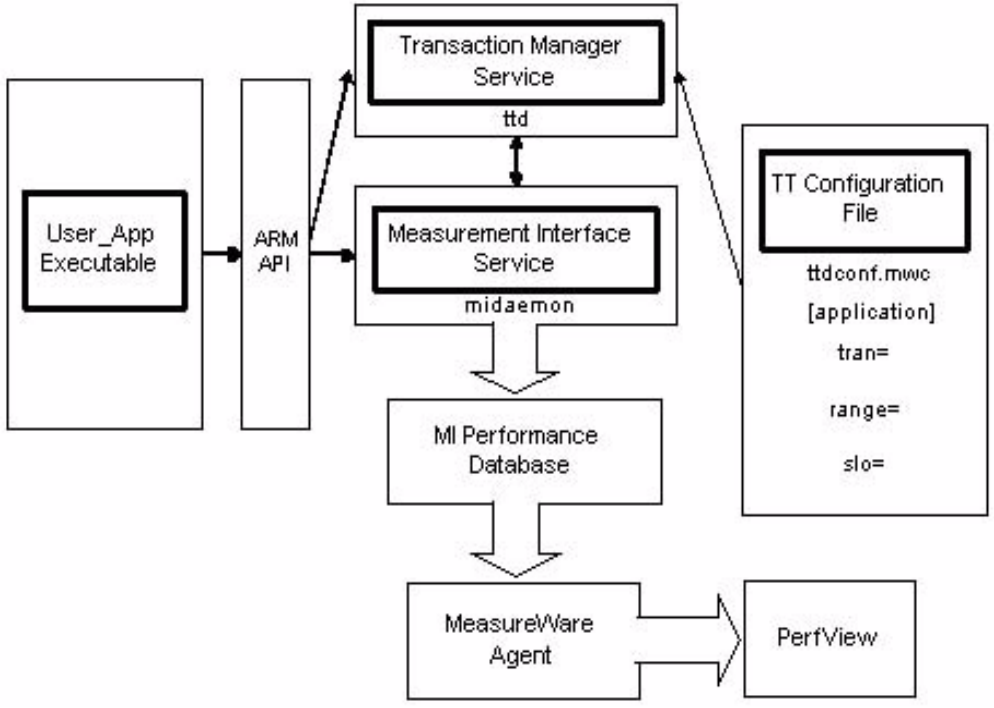
The following components of Performance Agent work together to help you define and track transaction data from applications instrumented with ARM API calls.

- The Measurement Interface service (`midaemon`) monitors and reports transaction data to the MI Performance Database where the information can be accessed and reported by Performance Agent and PerfView.
- The transaction configuration file, `ttdconf.mwc`, allows you to define transactions and identify the information to monitor for each transaction.
- The Transaction Manager service (`ttd`) reads, registers, and synchronizes transaction definitions from the transaction configuration file with the Measurement Interface service (`midaemon`).

These components are shown in the figure on the next page and are described in more detail in the remainder of this chapter.

For backward compatibility, you can compile and run applications that were instrumented with Transaction Tracker API calls, which were provided with previous releases of Performance Agent. However, all new applications *must* be instrumented with ARM API calls, rather than Transaction Tracker API calls.

Figure 1 Technical Overview of Transaction Tracking



Support of ARM 2.0

ARM 2.0 is a superset of the previous version of Application Response Measurement. The features that ARM 2.0 provides are user-defined metrics, transaction correlation, and a logging agent. Performance Agent supports user-defined metrics and transaction correlation but *does not* support the logging agent.

However, you may want to use the logging agent to test the instrumentation in your application. The source code for the logging agent, `logagent.c`, is included in the ARM 2.0 Software Developers Kit (SDK) that is available from the following web site:

http://regions. cmg.org/regions/cmgarmw

For information about using the logging agent, see the *Application Response Measurement 2.0 API Guide*.

The Application Response Measurement 2.0 API Guide uses the term “application-defined metrics” instead of “user-defined metrics.”

Windows Limitations

If you are instrumenting applications using the ARM 2.0 API for use with Performance Agentfor Windows, be aware of the following limitations in the `arm.h` file as provided in the ARM 2.0 Software Developers Kit and shown in the *Application Response Measurement 2.0 API Guide*.

When the int64 and unsigned64 structures are used to hold 64-bit integer values, the "upper" field should hold the lower 32 bits while the "lower" field should hold the upper 32 bits. This format is the opposite of what is defined in the example of the `arm.h` file in the "Examples" section of the *Application Response Measurement 2.0 API Guide*. This change is imposed for Windows NT/2000 because the definitions of the `arm.h` structures, as defined by the CMG ARM Working Group, are based on the byte order on UNIX® systems.

Windows NT/2000 handles this automatically with the use of ARM_INT64 as shown below in the example of code provided by Performance Agent for Windows NT/2000.

The following example of code is taken from the `arm.h` Header File example in the "Appendix: Measurement Agent Information" in the *Application Response Measurement 2.0 API Guide*. It shows the byte order on UNIX® systems.

```
typedef struct int64 {
arm_int32_t   upper;
arm_int32_t   lower;
} int64 ;

typedef struct unsigned64 {
unsigned32   upper;
unsigned32   lower;
} unsigned64 ;
```

The following example of code is the correct code as provided by Performance Agent for Windows NT/2000.

```
#define ARM_INT64_int64;

typedef ARM_INT64 arm_int64_t
typedef unsigned ARM_INT64 arm_unsigned64_t;
```

Support of ARM 2.0 API Calls

The following Application Response Measurement (ARM) API calls are supported in Performance Agent:

`arm_init()`	Names and registers the application and (optionally) the user.
`arm_getid()`	Names and registers a transaction class, provides related transaction information. Defines the context for user-defined metrics.
`arm_start()`	Signals the start of a unique transaction instance.
`arm_update()`	Updates the values of a unique transaction instance.
`arm_stop()`	Signals the end of a unique transaction instance.
`arm_end()`	Signals the end of the application.

See your current *Application Response Measurement 2.0 API Guide* for information on instrumenting applications with ARM API calls as well as complete descriptions of the API calls and their parameters. For commercial applications, check the product documentation to see if the application has been instrumented with ARM API calls.

For important information about required libraries, see ARM Library Information on page 41, in Chapter 3.

arm_complete_transaction Call

In addition to the ARM 2.0 API standard, the HP arm agent supports the `arm_complete_transaction` call. This call, which is an HP-specific extension to the ARM standard, can be used to mark the end of a transaction that has completed when the start of the transaction could not be delimited by an `arm_start` call. The `arm_complete_transaction` call takes as a parameter, the response time of the completed transaction instance.

In addition to signalling the end of a transaction instance, additional information about the transaction can be provided in the optional data buffer.

Sample ARM Instrumented Applications

The following sample programs are included with Performance Agent:

C source code	Executable
\<InstallDir>\arm\examples\armsample1.c	<InstallDir>\bin\armsample1.exe
\<InstallDir>\arm\examples\armsample2.c	<InstallDir>\bin\armsample2.exe
\<InstallDir>\arm\examples\armsample3.c	<InstallDir>\bin\armsample3.exe
\<InstallDir>\arm\examples\armsample4.c	<InstallDir>\bin\armsample4.exe

The <InstallDir> directory name is used throughout this document and stands for the directory in which Performance Agent is installed. The default directory is C:\Program Files\hp OpenView, but you can specify a different installation path for a first-time installation.

armsample1.c uses standard ARM API calls (ARM 1.0) and does not use any advanced functions. It executes ARMed transactions continuously in batch mode. No user interaction is required.

armsample2.c also uses standard ARM API calls (ARM 1.0) and does not use any advanced functions. It executes ARMed transactions interactively.

armsample3.c provides examples of how to use two of the new features, user-defined metrics and transaction correlation, provided by version 2.0 of the ARM API. It is a client/server program where both server and client perform a number of transactions. The user can get correlator information for a transaction by passing the request to the arm_start API call. The user can also pass the user-defined metrics and correlator to a specific transaction via the ARM API.

`armsample4.c` provides samples of how to use one of the new features, user-defined metrics, provided by version 2.0 of the ARM API. For each transaction, the user can pass user-defined metrics using `arm_start`, `arm_update`, and `arm_stop` API calls.

Specifying Application and Transaction Names

Although ARM allows a maximum of 128 characters each for application and transaction names in the `arm_init` and `arm_getid` API calls, Performance Agent shows *only* a maximum of 60 characters. All characters beyond the first 60 will *not* be visible.

Performance Agent applies certain limitations to how application and transaction names are shown in extracted or exported data. These rules also apply to viewing application and transaction names in PerfView.

The application name *always* takes precedence over the transaction name. For example, if you are exporting transaction data that has a 65-character application name and a 40-character transaction name, *only* the application name is shown. However, the last five characters of the application name are not visible.

For another example, if an application name has 29 characters and the transaction name has 35 characters, Performance Agent shows the entire application name but the transaction name appears truncated. A total of 60 characters are shown. Fifty-nine characters are allocated to the application and transaction names and one character is allocated to the underscore (_) that separates the two names. This is how the application name "`WarehouseInventoryApplication`" and the transaction name "`CallFromWestCoastElectronicSupplier`" would appear in Performance Agent or PerfView:

```
warehouseinventoryapplication_CallFromWestCoastElectronicSup
```

The 60-character combination of application name and transaction name must be unique if the data is to be viewed with PerfView.

Transaction Manager Service (ttd)

The Transaction Manager service (`ttd`) reads, registers, and synchronizes transaction definitions from the transaction configuration file, `ttdconf.mwc`, with the Measurement Interface service (`midaemon`). Processing occurs through ARM API calls that come from the application you are monitoring.

The Transaction Manager service (`ttd`) is started when you start the OV Performance Services from the OV Performance Services window. It runs in background mode when dispatched, and errors are written to the error file `status.ttd`.

The Measurement Interface service (`midaemon`) must also be running to process the transactions and to collect performance metrics associated with these transactions. (For more information, see Measurement Interface Service (midaemon) on page 28 in this chapter.)

We strongly recommend that you do not stop the Transaction Manager service (`ttd`).

If you stop the Transaction Manager service, any ARM-instrumented applications that are running must also be stopped before you restart the service and Performance Agent processes. The Transaction Manager service must be running to capture all `arm_init` and `arm_getid` calls being made on the system. If the Transaction Manager service is stopped and restarted, transactions IDs returned by these calls will be repeated, thus invalidating the ARM metrics.

Use the `mwa` command-line utility to start MWA processes to ensure that the processes are started in the correct order. `mwcmd stop` will not shut down `ttd`. If `ttd` must be shut down for a re-install or ant performance software, use the command `ttd -k`. However, we do not recommend that you stop `ttd`, except when re-installing Performance Agent.

See the *HP Performance Agent for Windows NT/2000: Installation & System Management* manual for information on starting the OV Performance services.

ARM API Call Status Returns

The Transaction Manager service, (ttd) must always be running in order to register transactions. If the service is *not* running, arm_init or arm_getid calls will return a "failed" return code. If the Transaction Manager service is restarted, new arm_getid calls may re-register the same transaction IDs that are already being used by other programs, thus causing invalid data to be recorded.

When the Transaction Manager service is terminated and subsequently restarted, ARM-instrumented applications may start getting a return value of -2 (TT_TTDNOTRUNNING) and an EPIPE errno error on ARM API calls. When your application initially starts, a client connection handle is created on any initial ARM API calls. This client handle allows your application to communicate with the Transaction Manager process. When the Transaction Manager service is terminated, this connection is no longer valid and the next time your application attempts to use an ARM API call, you may get a return value of TT_TTDNOTRUNNING. This error reflects that the *previous* Transaction Manager process is no longer running even though there is another Transaction Manager process running.

To get around this error, you must restart your ARM-instrumented applications if the Transaction Manager service is terminated. First, stop your ARMed applications. Next, restart the Transaction Manager service and then restart your applications. The restart of your application causes the creation of a new client connection handle between your application and the Transaction Manager process.

Some ARM API calls will not return an error if the Measurement Interface service (midaemon) has an error. For example, this would occur if the midaemon has run out of room in its shared memory segment. The performance metric GBL_TT_OVERFLOW_COUNT will be > 0. (Check for a midaemon error in the status.mi file.) If an overflow condition occurs, you may want to stop all OV Performance services as well as all ARMed applications, and then restart the midaemon using the -smdvss option to specify more room in the shared memory segment.

We recommend that your applications be written so that they continue to execute even if ARM errors occur. ARM status should not affect program execution.

The number of active client processes that can register transactions with `ttd` through the `arm_getid` call is limited to 512 open files by default. each client registration request results in `ttd` opening a socket (an open file) for the RPC connection. The socket is closed when the client application terminates. Therefore, this limit affects only a number of active clients that have registered a transaction through the `arm_getid` call. After this limit is reached, `ttd` returns `TT_TTDNOTRUNNING` to a client's `arm_getid` request. The maximum number of open files can be increased to 2048 by using the `_setmaxstdio()` option.

Measurement Interface Service (midaemon)

The Measurement Interface service (`midaemon`) is a low-overhead process that continuously collects system performance information, including transaction data. The Measurement Interface service must be running for Performance Agent to collect transaction data. The Measurement Interface service is started by the OV Performance Collector service, `scopent`, when you start the OV Performance services from the OV Performance Services window.

The Measurement Interface service must be started after the Transaction Manager service (`ttd`) has been started. In general, use the OV Performance Services window, not the Windows NT/2000 Services window, to start and stop OV Performance services to ensure that the processes are started in the correct order.

See the *HP Performance Agent for Windows : Installation and System Management Guide* for information on starting the OV Performance services.

Transaction Configuration File

The transaction configuration file, `ttdconf.mwc`, allows you to configure the following entries:

- the transaction name
- performance distribution ranges
- service level objectives to be met for each transaction
- application name if application-specific transactions are used

The Transaction Manager service (`ttd`) reads `ttdconf.mwc` to determine how to register each transaction.

A typical `ttdconf.mwc` entry looks like this:

```
tran=Personnel range=0.0005, 0.010, 0.015 slo=0.10
```

where `tran` defines the transaction name, `range` defines the performance distribution ranges, and `slo` defines the service level objective.

Customization of `ttdconf.mwc` is optional. The default configuration file that ships with Performance Agent causes *all* transactions instrumented in the application to be monitored.

If you are using a commercial application and don't know which transactions have been instrumented in the application, collect some data using the default configuration file. Then look at the data to see which transactions are available. You can then customize the transaction data collection for that application by modifying `ttdconf.mwc`.

The order of the entries in the `ttdconf.mwc` file is not relevant. Exact matches are sought first. If none are found, the longest match with a trailing asterisk (`*`) is used.

Adding New Applications

If you are adding new ARMed applications to your system that use the default `slo` and `range` values from the `tran=*` line in the `ttdconf.mwc` file, you do not have to do anything to incorporate the new transactions. (See the section, Configuration File Keywords on page 31 for descriptions of `tran`, `range`, and `slo`.) The new transactions will be picked up automatically. The `slo` and `range` values from the `tran=*` line in the `ttdconf.mwc` file will be applied to the new transactions.

Adding New Transactions

If you need to add new transactions to `ttdconf.mwc`, you *must* do the following:

1. Stop all ARMed applications that are currently running, including HP OpenView Service Reporter.
2. To add the new transactions to `ttdconf.mwc`, choose **Transactions** from the Configure menu in the Performance Agent main window.

After you have added the new transactions to the `ttdconf.mwc` file, you *must* do the following steps to activate the additions you made.

1. Choose **Stop/Start** from the Agent menu in the Performance Agent main Window.
2. Select the **Transactions** checkbox
3. Click the **Refresh** button. This action stops and then restarts the `scopent` collector. It then causes `ttdconf.mwc` to be re-read and registers the new transactions, along with their `slo` and `range` values, with the Transaction Manager service (`ttd`) and the Measurement Interface service (`midaemon`). The re-read will not change the `slo` or `range` values for transactions that were in the `ttdconf.mwc` file prior to the addition of the new transactions
4. Click the **Close** button.
5. Restart your ARMed applications.

Changing SLO or Range Values

If you need to change the `slo` or `range` values of existing transactions in the `ttdconf.mwc` file, you must do the following:

1. Stop all ARMed applications including HP OpenView View Service Reporter.
2. Stop the `scopent` collector by choosing OV Performance from the Windows NT/2000 Control Panel to display the OV Performance Services window, then click **Stop Services**.
3. Stop the Transaction Manager service (`ttd`) by choosing Services from the Windows NT Control Panel (the Services applet is under Administrative Tools in the Windows 2000 Control Panel). In the Services window, select **OV Performance Transaction Manager** service and click **Stop**.
4. Change the `slo` or `range` values in the `ttdconf.mwc` file as necessary by choosing **Transactions** from the Configure menu in the Performance Agent main window.
5. Restart the `scopent` collector by choosing **OV Performance** from the Windows NT/2000 Control Panel to display the OV Performance Services window, and click **Start Services**. This action also restarts the Transaction Manager service (`ttd`).
6. Restart your ARMed applications.

Configuration File Keywords

The transaction configuration file, `ttdconf.mwc`, associates transaction names with transaction attributes that are defined by the following keywords.

Table 1 Configuration File Keywords

Keyword	Syntax	Usage
`tran`	**`tran=transaction_name`**	Required
`range`	**`range=sec[,sec,...]`**	Optional
`slo`	**`slo=sec`**	Optional

These keywords are described in more detail below.

tran

Use `tran` to define your transaction name. This name must correspond to a transaction that is defined in the `arm_getid` API call in your instrumented application. You must use the `tran` keyword before you can specify the optional attributes `range` or `slo`. `tran` is the only required keyword in the `ttdconf.mwc` file. A trailing asterisk (`*`) in the transaction name causes a wild card pattern match to be performed when registration requests are made against this entry. Dashes *can* be used in a transaction name. However, spaces *cannot* be used in a transaction name.

The transaction name can contain a maximum of 128 characters. However, only the first 60 characters are visible in Performance Agent. (See Specifying Application and Transaction Names on page 24.)

The default `ttdconf.mwc` file contains several entries. The first entries define transactions used by the `scopent` data collector, which is instrumented with ARM API calls. The file also contains the single entry `tran=*`, which registers *all* transactions in applications instrumented with ARM API calls.

range

Use `range` to specify the transaction performance distribution ranges. Performance distribution ranges allow you to distinguish between transactions that take different lengths of time to complete and to see how many successful transactions of each length occurred.

Each value entered for *sec* represents the upper limit in seconds for the transaction time for the range. The value may be an integer or real number with a maximum of six digits to the right of the decimal point. On the Windows platform, however, the precision is 10 milliseconds (0.01 seconds), so only the first two digits to the right of the decimal point are recognized.

A maximum of 10 ranges are supported for each transaction you define. You can specify up to nine ranges. One range is reserved for an overflow range, which collects data for transactions that take longer than the largest user-defined range. If you specify more than nine ranges, the first nine ranges are used and the others are ignored.

If you specify fewer than nine ranges, the first unspecified range becomes the overflow range. Any remaining unspecified ranges are not used. The unspecified range metrics are reported as 0.000. The first corresponding unspecified count metric becomes the overflow count. Remaining unspecified count metrics are always zero (0).

Ranges must be defined in ascending order (see examples later on in this chapter).

slo

Use `slo` to specify the service level objective (SLO) in seconds that you want to use to monitor your performance service level agreement (SLA).

As with the `range` keyword, the `slo` value can be an integer or real number, with a maximum of six digits to the right of the decimal point. On the Windows platform, this allows for a precision of ten milliseconds (0.01 seconds) so only the first two digits to the right of the decimal point are recognized.

Configuration File Format

The `ttdconf.mwc` file can contain two types of entries: general transactions and application-specific transactions.

General transactions are defined at the beginning of `ttdconf.mwc` before any application is defined. These transactions will be associated with all the applications that are created.

The default `ttdconf.mwc` file contains one general transaction entry and entries for the `scopent` collector that is instrumented with ARM API calls.

```
tran=* range=0.5,1,2,3,5,10,30,120,300 slo=5.0
```

If you want to restrict management to specific applications and transactions specified in the `ttdconf.mwc` file, remove the above entry.

Optionally, each application can have its own set of transaction names. These transactions will be associated *only* with that application. The application name you specify must correspond to an application name defined in the `arm_init` API call in your instrumented application. Each group of application-specific entries must begin with the name of the application enclosed in brackets. For example:

```
[AccountRec]
tran=acctOne range=0.01, 0.03, 0.05
```

The application name can contain a maximum of 128 characters. However, only the first 60 characters are visible in Performance Agent.

If there are application-specific transactions that have the same name as a "general" transaction, the transaction listed under the application will be used.

For example:

```
tran=abc range=0.01, 0.03, 0.05 slo=0.10
tran=xyz range=0.02, 0.04, 0.06 slo=0.08
tran=t* range=0.01, 0.02, 0.03

[AccountRec]
tran=acctOne range=0.04, 0.06, 0.08
tran=acctTwo range=0.1, 0.2
tran=t* range=0.3, 0.5

[AccountPay]

[GenLedg]
tran=genLedgOne range=0.01
```

In the above example, the first three transactions apply to all of the three applications specified.

The application `[AccountRec]` has the following transactions: `acctOne`, `acctTwo`,`abc`, `xyz`, and `t*`. One of the entries in the general transaction set also has a wild card transaction named "`t*`". In this case, the "`t*`" transaction name for the `AccountRec` application will be used; the one in the general transaction set is ignored.

The application [`AccountPay`] uses only transactions from the general transactions set.

The application [`GenLedg`] has transactions `genLedgOne`, `abc`, `xyz`, and `t*`.

The ordering of transaction names makes no difference within the application.

For additional information about transaction and application names, see Specifying Application and Transaction Names on page 24, earlier in this chapter.

Configuration File Examples

Example 1

```
tran=* range=0.5,1,2,3,5,10,30,12,30 slo=5.0
```

The "*" entry is used as the default if none of the entries match a registered transaction name. These defaults can be changed on each system by modifying the "*" entry. If the "*" entry is missing, a default set of registration parameters are used that match the initial parameters assigned to the "*" entry.

Example 2

```
[MANufactr]
tran=MFG01 range=1,2,3,4,5,10 slo=3.0
tran=MFG02 range=1,2.2,3.3,4.0,5.5,10 slo=4.5
tran=MFG03
tran=MFG04 range=1,2.2,3.3,4.0,5.5,10
```

Transactions for the `MANufctr` application, `MFG01`, `MFG0`, and `MFG04`, each use their own unique parameters. The `MFG03` transaction does not need to track time distributions or service level objectives so it does not specify these parameters.

Example 3

```
[Financial]
tran=FIN01
tran=FIN02 range=0.1,0.5,1,2,3,4,5,10,20 slo=1.0
tran=FIN03 range=0.1,0.5,1,2,3,4,5,10,20 slo=2.0
```

Transactions for the for the `Financial` application, `FIN02` and `FIN03`, each use their own unique parameters. The `FIN01` transaction does not need to track time distributions or service level objectives so it does not specify these parameters.

Example 4

```
[PERSONL]
tran=PERS*  range=0.1,0.5,1,2,3,4,5,10,20 slo=1.0
tran=PERS03 range=0.1,0.2,0.5,1,2,3,4,5,10,20 slo=0.8
```

The PERS03 transaction for the PERSONL application uses its own unique parameters while the remainder of the transactions use a default set of parameters unique to the PERSONL application.

Example 5

```
[ACCOUNTS]
tran=ACCT_*    slo=1.0
tran=ACCT_REC range=0.5,1,2,3,4,5,10,20 slo=2.0
tran=ACCT_PAY range=0.5,1,2,3,4,5,10,20 slo=2.0
```

Transactions for the ACCOUNTS application, ACCT_REC and ACCT_PAY, each use their own unique parameters while the remainder of the transactions use a default set of parameters unique to the ACCOUNTS application. Only the accounts receivable and payable transactions need to track time distributions. The order of transaction names makes no difference within the application.

For more configuration file examples, see Chapter 6, Transaction Tracking Examples.

Overhead Considerations for Using ARM

The current version of Performance Agent contains modifications to its measurement interface that supports additional data required for ARM 2.0. These modifications can result in increased overhead for performance management. You should be aware of overhead considerations when planning ARM instrumentation for your applications.

Guidelines

Here are some guidelines to follow when instrumenting your applications with the ARM API:

- The total number of separate transaction IDs should be limited to not more than 4,000. Generally, it is cheaper to have multiple instances of the same transaction than it is to have single instances of different transactions. Register *only* those transactions that will be actively monitored.
- Although the overhead for the `arm_start` and `arm_stop` API calls is very small, it can increase when there is a large volume of transaction instances. More than a few hundred `arm_start` and `arm_stop` calls per second on most systems can significantly impact overall performance.
- Request ARM correlators *only* when using ARM 2.0 functionality. (For more information about ARM correlators, see the "Advanced Topics" section in the *Application Response Measurement 2.0 API Guide*.) The overhead for producing, moving, and monitoring correlator information is significantly higher than for monitoring transactions that are not instrumented to use the ARM 2.0 correlator functionality.
- Larger string sizes (applications registering lengthy transaction names, application names, and user-defined string metrics) will impose additional overhead.

Disk I/O Overhead

The performance management software does not impose a large disk overhead on the system. Performance Agent's collector, `scopent`, generates disk log files, but their size is not significantly impacted by ARM 2.0.

The `scopent logtran` log file is used to store ARM data. For more information, see Chapter 2, "Managing Data Collection," in the *HP Performance Agent for Windows NT/2000: User's Manual*.

CPU Overhead

A program instrumented with ARM calls will generally not run slower because of the ARM calls. This assumes that the rate of `arm_getid` calls is lower than one call per second, and the rate of `arm_start` and `arm_stop` calls is lower than a few thousand per second. More frequent calls to the ARM API should be avoided.

Most of the additional CPU overhead for supporting ARM is incurred inside of the performance tool programs and services themselves. Measurement Interface service (`midaemon`) CPU overhead rises slightly but not more than two percent more than it was with ARM 1.0. In addition, `scopent` CPU overhead will be slightly higher on a system with applications instrumented with ARM 2.0 calls. Only those applications that are instrumented with ARM 2.0 calls that make extensive use of correlators and/or user-defined metrics will have a significant performance impact on the Measurement Interface service and `scopent`.

A `midaemon` overflow condition can occur when usage exceeds the available default shared memory. This results in:

- No return codes from the ARM calls once the overflow condition occurs.
- Display of incorrect metrics, including blank process names.
- Errors being logged in `status.mi` (for example, "out of space").

Memory Overhead

Programs that are making ARM API calls will not have a significant impact in their memory virtual set size, except for the space used to pass ARM 2.0 correlator and user-defined metric information. These buffers, which are explained in the *Application Response Measurement 2.0 API Guide*, should not be a significant portion of a process's memory requirements.

There is additional virtual set size overhead in the performance tools to support ARM 2.0. The Measurement Interface service (`midaemon`) creates a shared memory segment where ARM data is kept internally for use by Performance Agent. The size of this shared memory segment has grown, relative to the size on releases with ARM 1.0, to accommodate the potential for use by ARM 2.0. By default on most systems, this shared memory segment is approximately two megabytes in size. This segment is not all resident in physical memory unless it is required. Therefore, this should not be a significant impact on most systems that are not already memory-constrained. The memory overhead of the Measurement Interface service can be tuned using special Performance Agent startup parameters.

3 Getting Started

Putting It All Together

This chapter gives you the information you need to begin tracking transactions and monitoring your service level objectives. For detailed reference information, see Chapter 2, How Transaction Tracking Works. For an example of how an application can be instrumented with ARM API calls and examples of how the transaction configuration file, `ttdconf.mwc`, can be customized, see Chapter 6, Transaction Tracking Examples.

ARM Library Information

Performance Agent installs the ARM shared library, `libarm32.dll`, into the `\%windir%\system32\` directory. The following additional libraries are installed into the `\InstallDir\lib\` directory:

`libarm32.lib`	ARM import library used to compile with applications being instrumented with ARM API calls.
`libarm32d.dll`	Debug version of the ARM library.
`libarm32d.lib`	Debug version of the ARM import library.
`libarmnop.dll`	No-operation (NOP) ARM library. When running an application instrumented with ARM API calls on a system where Performance Agent is *not* present, copy this library to the `\%windir%\system32\` directory and rename it as `libarm32.dll`.
`libarm32.dll`	Backup copy of the ARM shared library.

Setting Up Transaction Tracking

Follow the procedures below to set up transaction tracking for your application. These steps are described in more detail in the remainder of this section.

1. Define SLOs by determining what key transactions you want to monitor and the response level you expect (*optional*).
2. To monitor transactions in Performance Agent and PerfView, make sure that the Performance Agent `parm.mwc` file has transaction logging turned on. Then start or restart Performance Agent to read the updated `parm` file
3. Run the application that has been instrumented with ARM API calls, which are described in the *Application Response Measurement 2.0 API Guide*.
4. Use Performance Agent or PerfView to look at the collected transaction data. If the data isn't visible in PerfView, close the data source and then reconnect to it.
5. Customize the `ttdconf.mwc` file to modify the way transaction data for the application is collected (*optional*).
6. If you need to add transactions to `ttdconf.mwc`, see Adding New Applications on page 30. If you need to change the `slo` or `range` values of existing transactions in `ttdconf.mwc`, see the Changing SLO or Range Values on page 31.

Defining Service Level Objectives

Your first step in implementing transaction tracking is to determine the key transactions that are required to meet customer expectations and what level of transaction responsiveness is required. The level of responsiveness that is required becomes your service level objective (SLO). You define the service level objective in the transaction configuration file, `ttdconf.mwc`.

Defining service level objectives can be as simple as reviewing your Information Technology department's service level agreement (SLA) to see which transactions you need to monitor to meet your SLA. If you don't have an SLA, you may want to implement one. However, creating an SLA is not required in order to track transactions.

Modifying the Parm File

If necessary, modify the `parm.mwc` file to add transactions to the list of items to be logged for use by Performance Agent and PerfView. Include the `transaction` option as shown in the following example:

```
log global application process transaction device=disk
```

The default for the `log transaction` parameter is `no resource` and `no correlator`. To turn on resource data collection or correlator data collection, specify `log transaction=correlator` or `log transaction=resource`. Both can be logged by specifying `log transaction=resource, correlator`

Before you can collect transaction data, the modified `parm.mwc` file must be activated as follows:

1. Open the OV Performance Services window from the Windows Control Panel or the Performance Agent program.
2. Select the **Data Collection** check box.
3. Click the **Refresh** button.

If the Performance Agent is stopped, you must click **Start Services** button.

Collecting Transaction Data

Start up your application. Transaction data for your application is collected and synchronized as the application runs. The data is stored in the MI Performance Database where it can be used by Performance Agent and PerfView. See Monitoring Performance Data on page 46 for information on using each of these tools to view transaction data for your application.

Error Handling

Due to performance considerations, not all problematic ARM API calls return errors in real time. Some examples of when errors are *not* returned as expected are:

- calling `arm_start` with a bad `id` parameter such as an uninitialized variable
- calling `arm_stop` without a previously successful `arm_start` call

To debug these situations when instrumenting applications with ARM API calls, run the application long enough to generate and collect a sufficient amount of transaction data. Then export data from the file by choosing the **Export** command from the Logfiles menu in the Performance Agent main window. Examine the data to see if all transactions were logged as expected. (For more information about exporting data, see Performance Agent online Help or Chapter 3 in your *HP Performance Agent for Windows User's Manual*.)

Limits on Unique Transactions

Depending on your particular system resources, a limit may exist on the number of unique transactions allowed in your application. This limit is normally several thousand unique `arm_getid` calls.

This situation arises when the Measurement Performance Database used by the Measurement Interface service (`midaemon`) is full. If this happens, data for subsequent new transactions won't be logged and an appropriate warning message is written to the `status.scope` file. Transactions that have already been registered will continue to be logged and reported.

The `GBL_TT_OVERFLOW_COUNT` metric reports the number of new transactions that could not be measured.

This situation can be remedied by stopping and restarting the `midaemon` process in the Windows Command Prompt by using the `–smdvss` option to specify a larger shared memory segment size. The current shared memory segment size can be checked using the `midaemon –sizes` option.

Customizing the Configuration File (optional)

After viewing the transaction data from your application, you may want to customize the Transaction configuration file, `ttdconf.mwc`, to change the way transaction data for the application is collected. This is optional because the default configuration file works with all transactions defined in the application. If you do decide to customize `ttdconf.mwc`, complete this task on the same systems where you run your application. See the Transaction Configuration File on page 29 for information on the configuration file keywords – `tran`, `range`, and `slo`. Some examples of how each keyword is used are shown below:

`tran=`	Example:	`tran=answerid`
		`tran=answerid*`
		`tran*`

`range=`	Example:	`range=2.5,4.2,5.0,10.009`
`slo=`	Example:	`slo=4.2`

Customize `ttdconf.mwc` to include all of your transactions and each associated attribute. Note that the use of the `range` or `slo` keyword must be preceded by the `tran` keyword. An example of a `ttdconf.mwc` file is shown below:

```
tran=*
tran=my_first_transaction slo=5.5


[answerid]
tran=answerid range=2.5,4.2,5.0,8.0 slo=4.2


[orderid]
tran=orderid range=1.0,1.5,2.0,2.5,3.0,3.5,4.0
```

To add transactions to `ttdconf.mwc`, see the Adding New Transactions on page 30.

To change `slo` or `range` values to `ttdconf.mwc`, see the Changing SLO or Range Values on page 31.

Monitoring Performance Data

You can use Performance Agent and PerfView to monitor transaction data.

... with Performance Agent

By collecting and logging data for long periods of time, Performance Agent gives you the ability to analyze your system's performance over time and to perform detailed trend analysis. Data from Performance Agent can be viewed with PerfView or exported for use with a variety of other performance monitoring, accounting, modeling, and planning tools.

You can export transaction data for use with spreadsheets and analysis programs using the **Export** command from the Logfiles menu in the Performance Agent main window. For more information about exporting data, see the Performance Agent online Help or Chapter 3 of the *HP Performance Agent for Windows User's Manual*.

... with PerfView

PerfView imports Performance Agent data and gives you the ability to translate that data into a customized graphical or numerical format. Using PerfView, you can perform analysis of historical trends of transaction data and you can perform more accurate forecasting.

You can select **TRANSACTION** from the Class List window for a data source in PerfView, then graph transaction metrics for various transactions. For more information, see PerfView online Help, which is accessible from the PerfView Help menu. If you don't see the transactions you expect in PerfView, close the current data source and then reconnect to it.

Alarms

You can detect alarms in transaction data with Performance Agent and PerfView.

... with Performance Agent

In order to detect alarms with Performance Agent, you must define alarm conditions in its alarm definition file `alarmdef.mwc`. You can set up Performance Agent to notify you of an alarm condition in various ways, such as sending mail or initiating a call to your pager.

To pass a syntax check for the `alarmdef.mwc` file, you must have data logged for that application name and transaction name in the log files, or have the names registered in the transaction configuration file, `ttdconf.mwc`.

There is a limitation when you define an alarm condition on a transaction name that has a dash (-) in its name. To get around this limitation, use the ALIAS statement in the `alarmdef.mwc` file to redefine the transaction name.

For more information about alarms, see Performance Agent Online Help or Chapter 8 in the *HP Performance Agent for Windows User's Manual*.

... with PerfView

The PerfView Monitor option displays alarms generated by Performance Agent. You can view alarm information from all monitored systems and see graphs of metrics that characterize the performance of your systems or applications. Because PerfView is designed for multivendor installations, you can receive alarms from a variety of computer systems. When you receive an alarm, you can analyze the details surrounding it by using the color graphs and the tabular backup data provided by PerfView. For more information, see PerfView Online Help.

4 Transaction Tracking Messages

Transaction Tracking Messages

When an application instrumented with ARM API calls is running, any error messages you see will probably be from the Transaction Manager service (`ttd`) When a `ttd` error message occurs, see the `status.ttd` file for more information.

If a Measurement Interface service (`midaemon`) error message occurs, see either the `midaemon.err` or `status.mi` file for more information.

The error values returned from failed ARM API calls are given on the next page and can be utilized by an application developer when instrumenting an application with ARM API calls.

Return Values for Failed ARM API Calls

The following values are returned from failed ARM API calls:

Table 2 Return Values for Failed ARM API Calls

ARM API call	Return value *	Meaning
`arm_init()`	`ARM_APPNAMETOOLONG`	Application name is longer than `ARM_MAXNAMELEN`
	`ARM_USERNAMETOOLONG`	User name is longer than `ARM_MAXNAMELEN`
	`ARM_NOMEMAVAIL`	Insufficient memory available to complete the operation
`arm_getid()`	`ARM_INVID`	Invalid argument
	`ARM_TTDNOTRUNNING`	The Transaction Manager service (`ttd`) is not running, or the client application cannot connect to `ttd`
	`ARM_NAMENOTFOUND`	Transaction name not found or matched using the wild card facility in the transaction configuration file
	`ARM_BADOSVERS`	ARM library is not compatible with the operating system
`arm_start()`	`< 0`	An error occurred
`arm_stop()`	`< 0`	An error occurred
`arm_end()`	`< 0`	An error occurred

* The return values are defined in `c:\<InstallDir>\include\arm.h`

5 Transaction Metrics

This chapter describes all of the transaction metrics that are available if you are running applications that have been instrumented with ARM 2.0 API calls.

A subset of transaction metrics will have values if transaction correlation is turned off in Performance Agent.

Global Metrics

GBL_TT_OVERFLOW_COUNT

The number of new transactions that could not be measured because the Measurement Processing Daemon's (`midaemon`) Measurement Performance Database is full. If this happens, the default Measurement Performance Database size is not large enough to hold all of the registered transactions on this system. This can be remedied by stopping and restarting the `midaemon` process using the `-smdvss` option to specify a larger Measurement Performance Database size. The current Measurement Performance Database size can be checked using the `midaemon -sizes` option.

Per Transaction Metrics

TT_ABORT

The number of aborted transactions during the last interval.

TT_ABORT_WALL_TIME_PER_TRAN

The average transaction time in seconds for aborted transactions during the last interval.

TT_APP_NAME

The registered ARM application name. The maximum length of the application name is 128 characters. However, only the first 60 characters are visible in Performance Agent. For an explanation, see Specifying Application and Transaction Names on page 24.

TT_APP_TRAN_NAME

The combined ARM application name and transaction name. The maximum length of the combined names is 128 characters. Only 60 characters are visible in Performance Agent. The application name *always* takes precedence over the transaction name. For an explanation , see Specifying Application and Transaction Names on page 24.

TT_CLIENT_ADDRESS

In a client/server environment, this is the IP address of the client system. This client system has requested work be done by the server on its behalf. Note that in some instances, the client process may be on the same system as the server process.

TT_CLIENT_ADDRESS_FORMAT

The format of the IP address described by metric `TT_CLIENT_ADDRESS`. The formats are defined in the *Application Response Measurement 2.0 API Guide*.

TT_CLIENT_TRAN_ID

In a client/server environment, this is the transaction ID that will be found on the client system. This transaction ID corresponds to the transaction that requested work be done on the server on its behalf. This helps trace a specific transaction that may involve several systems.

TT_COUNT

The number of transactions that completed during the last interval for this transaction name.

TT_FAILED

The number of failed transactions during the last interval for this transaction name.

TT_INFO

The registered ARM transaction information for this transaction.

INTERVAL

The number of seconds between data collection points for this transaction.

TT_NAME

The registered transaction name. The maximum length of the transaction name is 128 characters. However, only the first 60 characters are visible. For an explanation, see Specifying Application and Transaction Names on page 24.

TT_NUM_BINS

The number of distribution ranges (also called bins).

TT_SLO_COUNT

The number of completed transactions that violated the defined SLO by exceeding the SLO threshold time during the interval.

TT_SLO_PERCENT

The percentage of transactions that violate service level objectives.

TT_SLO_THRESHOLD

The upper range (transaction time) of the SLO threshold value. This value is used to count the number of transactions that exceed this user-supplied transaction time value.

TT_TERM_TRAN_1_HR_RATE

The number of transactions completed per hour.

TT_TRAN_1_MIN_RATE

The number of transactions completed per minute.

TT_TRAN_ID

The registered ARM Transaction ID for this transaction as returned by `arm_getid`. A unique transaction id is returned for a unique application (returned by `arm_init`), transaction name, and meta data buffer contents.

TT_UNAME

The registered ARM Transaction User Name for this transaction name.

TT_USER_MEASUREMENT_NAME(_6)

The name of the user-defined transactional measurement.

TT_USER_MEASUREMENT_MIN(_6)

If the measurement type is a counter, this metric returns the lowest measured counter value over the life of the transaction. The counter value is the difference observed from a counter between the start and the stop (or last update) of a transaction.

If the measurement type is a numeric or a gauge, this metric returns the lowest value passed on any ARM call over the life of the transaction.

TT_USER_MEASUREMENT_MAX(_6)

If the measurement type is a counter, this metric returns the highest measured counter value over the life of the transaction. The counter value is the difference observed from a counter between the start and stop (or last update) of a transaction.

If the measurement type is a numeric or a gauge, this metric returns the highest value passed on any ARM call over the life of the transaction.

TT_USER_MEASUREMENT_AVG(_6)

If the measurement type is a counter, this metric returns the average user-defined metric counter differences of the transaction during the last interval. The counter value is the difference observed from a counter between the start and the stop (or last update) of a transaction.

If the measurement type is a numeric or a gauge, this metric returns the average of the values passed on any ARM call for the transaction during the last interval.

TT_USER_MEASUREMENT_COUNT(_6)

This returns the total number of times the associated user-defined metric (UDM) was sampled during the last interval.

TT_WALL_TIME_PER_TRAN

The average transaction time in seconds for this transaction during the last interval.

TTBIN_TRANS_COUNT_1 ... 10

These metrics log the number of completed transactions in this distribution range during the last interval.

TTBIN_UPPER_RANGE_1 ... 10

The upper range (transaction time) for this bin.

6 Transaction Tracking Examples

This chapter contains a pseudocode example of how an application might be directly instrumented with ARM API calls, so that the transactions defined in the application can be monitored with Performance Agent. This pseudocode example corresponds with the real time order processing scenario described in Chapter 1.

Several examples of transaction configuration files are also included in this chapter, including one that corresponds with the real time order processing scenario.

See the next chapter, Advanced Features and your current *Application Response Measurement 2.0 API Guide* for information on instrumenting applications with ARM advanced functions. These functions include user-defined metrics (which are called application-defined metrics in ARM) and transaction correlation.

Pseudocode for Real Time Order Processing

This pseudocode example includes the ARM API calls used to define transactions for the real time order processing scenario introduced in Chapter 1. This routine would be processed *each time* an operator answered the phone to handle a customer order. The lines containing the ARM API calls are highlighted with bold text in this example.

```
routine answer calls()
{
*****************************************************
•Register the transactions if first time in       *
*****************************************************
if (transactions not registered)
   {
appl_id = arm_init("Order Processing Application","*", 0,0,0)
answer_phone_id = arm_getid(appl_id,"answer_phone","1st tran",0,0,0)
if (answer_phone_id < 0)
REGISTER OF ANSWER_PHONE FAILED - TAKE APPROPRIATE ACTION
order_id = arm_getid(appl_id,"order","2nd tran",0,0,0)
if (order_id < 0)
REGISTER OF ORDER FAILED - TAKE APPROPRIATE ACTION
check_id = arm_getid(appl_id,"check_db","3rd tran",0,0,0)
if (check_id < 0)
REGISTER OF CHECK DB FAILED - TAKE APPROPRIATE ACTION
update_id = arm_getid(appl_id,"update","4th tran",0,0,0)
if (update_id < 0)
REGISTER OF UPDATE FAILED - TAKE APPROPRIATE ACTION
} if transactions not registered
*****************************************************
•Main transaction processing loop
*****************************************************
while (answering calls)
{
if (answer_phone_handle = arm_start(answer_phone_id,0,0,0) < -1)
TRANSACTION START FOR ANSWER_PHONE NOT REGISTERED
*****************************************************

•At this point the answer_phone transaction has    *
•started.  If the customer does not want to order, *
•end the call; otherwise, proceed with order.      *
*****************************************************
if (don't want to order)
arm_stop(answer_phone_handle,ARM_FAILED,0,0,0)
GOOD-BYE - call complete
else
      {
*****************************************************
```

```
•They want to place an order - start an order now  *
*****************************************************
if (order_handle = arm_start(order_id,0,0,0) < -1)
TRANSACTION START FOR ORDER FAILED
take order information: name, address, item, etc.
*****************************************************
•Order is complete - end the order transaction    *
*****************************************************
if (arm_stop(order_handle,ARM_GOOD,0,0,0) < -1)
TRANSACTION END FOR ORDER FAILED
*****************************************************
•order taken - query database for availability     *
*****************************************************
if (query_handle = arm_start(query_id,0,0,0) < -1)
TRANSACTION QUERY DB FOR ORDER NOT REGISTERED
query the database for availability
*****************************************************
•database query complete - end query transaction  *
*****************************************************
if (arm_stop(query_handle,ARM_GOOD,0,0,0) < -1)
TRANSACTION END FOR QUERY DB FAILED
*****************************************************
•If the item is in stock, process order, and        *
•update inventory.                                  *
*****************************************************
if (item in stock)
if (update_handle = arm_start(update_id,0,0,0) < -1)
TRANSACTION START FOR UPDATE NOT REGISTERED
update stock
*****************************************************
•update complete - end the update transaction     *
*****************************************************
if (arm_stop(update_handle,ARM_GOOD,0,0,0) < -1)
TRANSACTION END FOR ORDER FAILED
*****************************************************
•Order complete - end the call transaction          *
*****************************************************
if (arm_stop(answer_phone_handle,ARM_GOOD,0,0,0) < -1)
TRANSACTION END FOR ANSWER_PHONE FAILED
} placing the order
GOOD-BYE - call complete
sleep("waiting for next phone call...zzz...")
} while answering calls
arm_end(appl_id, 0,0,0)
} routine answer calls
```

Configuration Files Examples

This section contains some examples of `ttdconf.mwc` files. For more information on `ttdconf.mwc` and configuration file keywords, see the Transaction Configuration File on page 29.

Example 1 (for Order Processing Pseudocode Example)

```
# The "*" entry below is used as the default if none of the
# entries match a registered transaction name.


tran=* range=0.5,1,1.5,2,3,4,5,6,7 slo=1
tran=answer_phone* range=0.5,1,1.5,2,3,4,5,6,7 slo=5
tran=order* range=0.5,1,1.5,2,3,4,5,6,7 slo=5
tran=query_db* range=0.5,1,1.5,2,3,4,5,6,7 slo=5
```

Example 2

```
# The "*" entry below is used as the default if none of the
# entries match a registered transaction name.
tran=* range=1,2,3,4,5,6,7,8 slo=5
# The entry below is for the only transaction being tracked in
# this application.  The "*" has been inserted at the end of
the
# tran name to catch any possible numbered transactions.  For
# example "First_Transaction1", "First_Transaction2", etc.
tran=First_Transaction* range=1,2.2,3.3,4.0,5.5,10 slo=5.5
```

Example 3

```
# The "*" entry below is used as the default if none of the
# entries match a registered transaction name.
tran=*
```

```
tran=Transaction_One range=1,10,20,30,40,50,60 slo=30
```

Example 4

```
tran=FactoryStor* range=0,0.10,0.15,slo=3

# The entries below show the use of an application name.
# Transactions are grouped under the application name.  This
# example also shows the use of less than 10 ranges and
optional
# use of "slo."


[Inventory]
tran=In_Stock range=0.01,0.04,0.08
tran=Out_Stock range=0.01,0.05
tran=Returns range=0.1,0.3,0.7


{Pers]

tran=Acct range=0.5,0.10,slo=5

tran=Time_Cards range=0.010,0.020
```

7 Advanced Features

This chapter describes how Performance Agent uses the following ARM 2.0 features:

- data types
- user-defined metrics
- `scopent` ARM instrumentation
- correlators

How Data Types Are Used in Performance Agent

The following table describes how data types are used in Performance Agent. It is a supplement to "Data Type Definitions" in the "Advanced Topics" section of the *Application Response Measurement 2.0 API Guide*.

`ARM_Counter32`	Data is logged as a 32-bit integer.
`ARM_Counter64`	Data is logged as a 32-bit integer with type casting.
`ARM_CntrDivr32`	Makes the calculation and logs the result as a 32-bit integer.
`ARM_Gauge32`	Data is logged as a 32-bit integer.
`ARM_Gauge64`	Data is logged as a 32-bit integer with type casting.
`ARM_GaugeDivr32`	Makes the calculation and logs the result as a 32-bit integer.
`ARM_NumericID32`	Data is logged as a 32-bit integer.
`ARM_NumericID64`	Data is logged as a 32 bit integer with type casting.
`ARM_String8`	Ignored.
`ARM_String32`	Ignored.

Performance Agent does not log string data. Because Performance Agent logs data every five minutes, and what is logged is the summary of the activity for that interval, it cannot summarize the strings provided by the application.

Performance Agent logs the Minimum, Maximum, and Average for the first six usable user-defined metrics.

For example, if your ARM-instrumented application passes a Counter32, String8, NumericID32, Gauge32, Gauge64, Counter64, NumericID64, String32, and GaugeDivr32, Performance Agent logs the Min, Max, and Average over the five-minute interval for the Counter32, NumericID32, Gauge32, Gauge64, Counter64 and NumericID64 as 32-bit integers. The

String8 and String32 are ignored because strings cannot be summarized in Performance Agent. The GaugeDivr32 is also ignored because only the first six usable user-defined metrics are logged. (For more examples, see the next section, User-Defined Metrics).

User-Defined Metrics

This section is a supplement to “Application-Defined Metrics” under “Advanced Topics” in the *Application Response Measurement 2.0 API Guide*. It contains some examples about how Performance Agent handles user-defined metrics (referred to as application-defined metrics in ARM). The examples show what will be logged if your program passes the following data types.

Table 3 Examples of User Defined Metrics

What your program passes in...	What will be logged
1) String8 Counter32 Gauge32 CntrDivr32	Counter32 Gauge32 CntrDivr32
2) String32 NumericID32 NumericID64	NumericID32 NumericID64
3) NumericID32 String8 NumericID64 Gauge32 String32 GaugeDivr64	NumericID32 NumericID64 Gauge64 GaugeDivr64
4) String8 String32	(nothing)
5) Counter32 Counter64 CntrDivr32 Gauge32 Gauge64 NumericID32 NumericID64	Counter32 Counter64 CntrDivr32 Gauge32 Gauge64 NumericID32

Note that no strings are logged because Performance Agent cannot summarize strings.

In example 1, the counter, gauge and counter divisor are logged; the string is not logged.

In example 2, the string is not logged so only the numerics are logged.

In example 3, only the numerics and gauges are logged.

In example 4, nothing is logged.

In example 5, because only the first six user-defined metrics are logged, NumericID64 is not logged.

Scopent ARM Instrumentation

The `scopent` data collector has been instrumented with ARM API calls. When Performance Agent starts, `scopent` automatically starts logging three transactions, `Scope_Get_Process_Metrics`, `Scope_Get_Global_Metrics`, and `Scope_Log_Headers`. All three transactions are in the HP Performance Tools application.

`scopent` transactions are logged by default. The default `parm.mwc` file contains a commented entry, `scopetransactions=off`. If you *do not* want `Scope_Get_Process_Metrics` and `Scope_Get_Global_Metrics` to be logged, uncomment the entry `scopetransactions=off`.

Transaction data is logged every five minutes so you will find that five `Scope_Get_Process_Metrics` transactions (one transaction per minute) have completed during each interval. This transaction is instrumented around the processing of process data. If there are 200 processes on your system, the transaction should take longer than if there are only 30 processes on your system.

The `Scope_Get_Global_Metrics` transaction is instrumented around the gathering of all five-minute data, including global data. This includes global, application, disk, transaction, and other data types, so you should see one completed transaction with this name during each five-minute interval..

The `Scope_Log_Headers` transaction will always be logged. It is not affected by the `scopetransactions=off` entry in the `parm.mwc` file.

To stop the logging of processes and global transactions data, remove or comment out the entries for the `scopent` transactions in the `ttd.conf` file.

Special Considerations When Using a Correlator

The correlator is always in network byte order (see the "Advanced Topics" section of the *Application Response Measurement 2.0 API Guide*). This means that when you want to copy the correlator from one buffer to another, or any other time that you reference the correlator's length field, you must convert that length field to host byte order. You can use the `network-to-host` library call to do this. The following code from the `armsample3.c` sample application is an example of how to make the conversion.

```
buf_ptr->correlator.length = client_correlator.length;
data_len = (ntohs (client_correlator.length) -
                  sizeof(client_correlator.length));
for (i = 0; i < data_len; i++)
    buf_ptr->correlator.agent_data[i] =
client_correlator.agent_data[i];
```

A Appendix

Overview

This appendix discusses:

- C compiler option examples by platform
- using Java wrappers

C Compiler Option Examples

The `arm.h` include file is located in `/<InstallDir>/include/`. The following example shows a compile command for a C program using the ARM API.

```
c1.exe .\armsample1.c /Fe. \armsample1.exe /I
\<InstallDir>\include /link\<InstallDir>\lib\libarm32.lib

c1.exe .\armsample2.c /Fe. \armsample2.exe /I
\<InstallDir>\include /link\<InstallDir>\lib\libarm32.lib

c1.exe .\armsample3.c /Fe. \armsample3.exe /I
\<InstallDir>\include /link\<InstallDir>\lib\libarm32.lib

c1.exe .\armsample4.c /Fe. \armsample4.exe /I
\<InstallDir>\include /link\<InstallDir>\lib\libarm32.lib
```

Using the Java Wrappers

The Java Native Interface (JNI) wrappers are functions created for your convenience to allow the Java applications to call the ARM API. These wrappers (`armapi.jar` and `armjava.dll`) are included with the ARM sample programs located in the `\<InstallDir>\examples\arm\` directory. InstallDir is the directory in which Performance Agent is installed.

Examples

Examples of the Java wrappers are located in the `\<InstallDir>examples\arm\` directory. This location also contains a README file, which explains the function of each wrapper.

Setting Up an Application (arm_init)

To set up a new application, make a new instance of ARMApplication and pass the name and the description for this API. Each application needs to be identified by a unique name. The ARMApplication class uses the C – function `arm_init`.

Syntax:

```
ARMApplication myApplication =
new ARMApplication("name","description");
```

Setting Up a Transaction (arm_getid)

To set up a new transaction, you can choose whether or not you want to use user-defined metrics (UDMs). The Java wrappers use the C – function `arm_getid`.

Setting Up a Transaction With UDMs

If you want to use UDMs, you must first define a new ARMTranDescription. ARMTranDescription builds the Data Buffer for `arm_getid`. (See also the `jprimeudm.java` example.)

Syntax:

```
ARMTranDescription myDescription =
new ARMTranDescription("transactionName","details");
```

If you do not want to use details, you can use another constructor:

Syntax:

```
ARMTranDescription myDescription =
new ARMTranDescription("transactionName");
```

Adding the Metrics

Metric 1-6:

Syntax:

```
myDescription.addMetric(metricPosition, metricType,
metricDescription);
```

Parameters:

`metricPosition: 1-6`

```
metricType: ARMConstants.ARM_Counter32
ARMConstants.ARM_Counter64 ARMConstants.ARM_CntrDivr32
ARMConstants.ARM_Gauge32 ARMConstants.ARM_Gauge64
ARMConstants.ARM_GaugeDivr32 ARMConstants.ARM_NumericID32
ARMConstants.ARM_NumericID64 ARMConstants.ARM_String8s
```

Metric 7:

Syntax:

```
myDescription.addStringMetric("description");
```

Then you can create the Transaction:

Syntax:

```
myApplication.createTransaction(myDescription);
```

Setting the Metric Data

Metric 1-6:

Syntax:

```
myTransaction.setMetricData(metricPosition, metric);
```

Examples for "Metric"

```
ARMGauge32Metric metric = new ARMGauge32Metric(start);
ARMCounter32Metric metric = new ARMCounter32Metric(start);
ARMCntrDivr32Metric metric = new ARMCntrDivr32Metric(start,
1000);
```

Metric 7:

Syntax:

```
myTransaction.setStringMetricData(text);
```

Setting Up a Transaction Without UDMs

When you set up a transaction without UDMs, you can immediately create the new transaction. You can choose whether or not to specify details.

With Details

Syntax:

```
ARMTransaction myTransaction =
myApplication.createTransaction(“Transactionname”,”detail
s”;
```

Without Details

Syntax:

```
ARMTransaction myTransaction =
myApplication.createTransaction(“Transactionname”);
```

Setting Up a Transaction Instance

To set up a new transaction instance, make a new instance of ARMTransactionInstance with the method `createTransactionInstance()` of ARMTransaction.

Syntax:

```
ARMTransactionInstance myTranInstance =
myTransaction.createTransactionInstance();
```

Starting a Transaction Instance (arm_start)

To start a transaction instance, you can choose whether or not to use correlators. The following methods call the C – function `arm_start` with the relevant parameters.

Starting the Transaction Instance Using Correlators

When you use correlators, you must distinguish between getting and delivering a correlator. If your transaction instance wants to request a correlator, the call is as follows (see also the `jcorrelators.java` example):

Syntax:

```
int status = myTranInstance.startTranWithCorrelator();
```

If you already have a correlator from a previous transaction and you want to deliver it to your transaction, the syntax is as follows:

Syntax:

```
int status =
myTranInstance.startTranWithCorrelator(parent);
```

Parameter:

`parent` is the delivered correlator. In the previous transaction, you can get the transaction instance correlator with the method `getCorrelator()`.

Syntax

```
int status = startTran(parent);
```

Parameter

`parent` is the delivered correlator. In the previous transaction, you can get the transaction instance correlator with the method `getCorrelator()`.

Syntax:

```
ARMTranCorrelator parent =
myTranInstance.getCorrelator();
```

Starting the Transaction Instance Without Using Correlators

When you do not use correlators, you can start your transaction instance as follows:

Syntax:

```
int status = myTranInstance.startTran();
```

`startTran` returns a unique handle to status, which is used for the update and stop.

Updating Transaction Instance Data

You can update the UDMs of your transaction instance any number of times between the start and stop. This part of the wrappers calls the C – function `arm_update` with the relevant parameters.

Updating Transaction Instance Data With UDMs

When you update the data of your transaction instance with UDMs, first, you must set the new data for the metric. For example,

`metric.setData(value)  for ARM_Counter32 ARM_Counter64, ARM_Gauge32, ARM_Gauge64, ARM_NumericID32, ARM_NumericID64`

`metric.setData(value,value) for ARM_CntrDivr32 and , ARM_GaugeDivr32`

`metric.setData(string)  for ARM_String8 and ARM_String32`

Then you can set the metric data to new (like the examples in the section Setting the Metric Data on page 75) and call the update:

Syntax:

```
myTranInstance.updateTranInstance();
```

Updating Transaction Instance Data Without UDMs

When you update the data of your transaction instance without UDMs, you just call the update. This sends a “heartbeat” indicating that the transaction instance is still running.

Syntax:

```
myTranInstance.updateTranInstance();
```

Providing a Larger Opaque Application Private Buffer

If you want to use the second buffer format, you must pass the byte array to the update method. (See the *Application Response Measurement 2.0 API Guide*.)

Syntax:

```
myTranInstance.updateTranInstance(byteArray);
```

Stopping the Transaction Instance (arm_stop)

To stop the transaction instance, you can choose whether or not to stop it with or without a metric update.

Stopping the Transaction Instance With a Metric Update

To stop the transaction instance with a metric update, call the method `stopTranInstanceWithMetricUpdate`.

Syntax:

```
myTranInstance.stopTranInstanceWithMetricUpdate
(transactionCompletionCode);
```

Parameter:

The transaction Completion Code can be:

`ARMConstants. ARM_GOOD.`	Use this value when the operation ran normally and as expected.
`ARMConstants.ARM_ABORT.`	Use this value when there is a fundamental failure in the system.
`ARMConstants.ARM_FAILED.`	Use this value in applications where the transaction worked properly, but no result was generated.

These methods use the C – function `arm_stop` with the requested parameters.

Stopping the Transaction Instance Without a Metric Update

To stop the transaction instance without a metric update, you can use the method `stopTranInstance`.

Syntax:

```
myTranInstance.stopTranInstance(transactionCompletionCode
);
```

Using Complete Transaction

The Java wrappers can use the `arm_complete_transaction` call. This call can be used to mark the end of a transaction that has lasted for a specified number of nanoseconds. This enables the real time integration of transaction response times measured outside of the ARM agent.

In addition to signaling the end of a transaction instance, additional information about the transaction (UDMs) can be provided in the optional data buffer.

(See also the jcomplete.java example.)

Using Complete Transaction With UDMs:

Syntax:

```
myTranInstance.completeTranWithUserData(status,responseTi
me;
```

Parameters:

`status`	• `ARMConstants. ARM_GOOD` Use this value when the operation ran normally and as expected. • `ARMConstants.ARM_ABORT` Use this value when there was a fundamental failure in the system. • `ARMConstants.ARM_FAILED` Use this value in applications where the transaction worked properly, but no result was generated.
`responseTime`	This is the response time of the transaction in nanoseconds.

Using Complete Transaction Without UDMs:

Syntax:

```
myTranInstance.completeTran(status,responseTime);
```

Further Documentation

For further information about the Java classes, see the doc folder in the `\<InstallDir>\examples\arm\` directory, which includes html-documentation for every Java class. Start with `index.htm`.

Glossary

alarm

A signal that an alarm event has occurred. The signal can be either a notification or an automatically triggered action. The event can be a pre-defined condition that has been met or was exceeded.

client

A system that requests service from a server. In the context of diskless clusters, a client uses the server's disks and has none of its own. In the context of NFS, a client mounts file systems that physically reside on another system (the Network File System Server).

export

A Performance Agent function that copies log file data from the performance application to an external file format for use by other programs (such as spreadsheets and word processors).

extract

A Performance Agent program that lets you extract data from raw or extracted log files, summarizes it, and writes the data to extracted log files. It also lets you export data for use in spreadsheets and analysis programs.

interval

Specific time periods during which performance data is gathered.

log files

Performance Agent performance measurement data files that are either raw or extracted. Raw log files contain summarized measurements of system data. Extracted log files contain a user-defined subset of data that was extracted from a raw log file.

measurement interface

A set of proprietary library calls used by the performance applications to obtain performance data.

Measurement Interface service (midaemon)

The service that monitors system and application performance and creates counters from event traces that are read and displayed by performance applications.

metric

A specific system measurement that helps you characterize performance.

MI shared memory segment

The interface between the kernel and the performance collectors. The Measurement Interface service (`midaemon`) translates trace data into this shared memory segment where it can be accessed by Performance Agent and PerfView. (Also known as the MI Performance Database.)

performance distribution range

An amount of time that you define with the `range= keyword` in the transaction configuration file, `ttdconf.mwc`.

resource manager

A company's Information Technology (IT) manager who monitors service levels between IT and other business sections of a company.

scopent

The Performance Agent program that collects performance data and writes (logs) the raw measurement data to log files for later analysis or archiving.

service level agreement (SLA)

A document prepared for each mission- and business-critical application that explicitly defines the service levels that IT is expected to deliver to users. It specifies what the user group can expect from the IT community in terms of system response, quantities of work, and system availability.

service level objective (SLO)

Objectives that identify what the IT staff must do to support the terms of the SLA, how it will monitor the provisions, and what it will do when an exception occurs. SLAs are not required for a company to implement SLOs.

transaction

Some amount of work performed by a computer system on behalf of a user. The boundaries of this work are defined by the user.

transaction configuration file (ttdconf.mwc)

The configuration file in which you define the attributes of a transaction, including transaction name, performance distribution ranges, and service level objectives. *See also* **Transaction Manager service (ttd)**.

Transaction Manager service (ttd)

The service that reads, registers, and synchronizes transaction definitions from the transaction configuration file, `ttdconf.mwc`, with the Measurement Interface service (`midaemon`).

Index

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V