Application Response Measurement

2.0 API Guide

October 2005

Legal Notices

Warranty

Hewlett-Packard makes no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices

Trademark Notices

Adobe® is a trademark of Adobe Systems Incorporated.

HP-UX Release 10.20 and later and HP-UX Release 11.00 and later (in both 32 and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

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.

Support

Please visit the HP OpenView support web site at:

http://www.hp.com/managementsoftware/support

This web site provides contact information and details about the products, services, and support that HP OpenView offers.

HP OpenView online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valuable support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit enhancement requests online
- Download software patches
- Submit and track progress on support cases
- Manage a support contract
- Look up HP support contacts
- Review information about available services
- Enter 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 log in. Many also require a support contract.

To find more information about access levels, go to:

http://www.hp.com/managementsoftware/access_level

To register for an HP Passport ID, go to:

http://www.managementsoftware.hp.com/passport-registration.html

Contents

1 Application Response Measurement API

Introduction

The applications that are used to run businesses have changed dramatically over the past few years. In the early 1980s, large applications generally executed on large computers, and were accessed from "dumb" terminals. Non-networked applications executing on personal computers were just beginning to be widely used. Since then, these two application models have moved steadily towards each other, fusing together to form distributed (networked) applications.

The most common programming model for distributed applications is the client/server model. In a client/server application, the application is split into two or more parts. One part is the user or **client** part, and this part generally executes on a personal computer or workstation. The **server** parts execute on computers that provide functions for the client part, that is, they serve the client application. The client and server can run on the same system, but generally they are on different systems. The client part of an application may invoke one or more functions on one or more servers, and it may do a significant amount of processing itself – combining, manipulating, or analyzing the data provided by the servers.

An example of a client/server application might be processing a sales order by retrieving inventory information from one database, sales information from another database, and pricing information from a third. The client part of the application determines if there is sufficient inventory to accept the order, calculates the price based on current market conditions, factors in price discounts for this particular customer, and then invokes more server functions to complete processing of the order.

By contrast, host-centric applications contain all the application logic in one computer system, and users connect through "dumb" terminals to use the application. Examples of the protocols used by these applications are 3270, Telnet, and X-Windows. The response time as seen by a user for a transaction

can generally be broken down into two components: the time to process the transaction on the host, and the time for the input message and the output response. Processing time at the terminal is usually trivial.

Measuring Service Levels

A monitoring product running at the host is able to measure the service levels of host-centric applications. The monitor observes the input request message that starts the transaction, and then observes the outbound response back to the terminal. The difference between the two times is the amount of time to process the transaction on the host. The monitor generally also measures the time for the outbound response to be sent to the terminal and an acknowledgment to be received, using this as an approximation of the transit time. The combination of the host and transit times is an approximation of the service level seen by the user.

Monitoring the performance and the availability of distributed applications has not proven easy to do. Some of the fundamental assumptions that the host-centric methods depend on do not hold true. Some examples:

- The user is typically running an application on a multitasking PC or workstation. When the user presses a key or the mouse button, the specified transaction starts, but the user may be able to continue doing other operations. Put another way, there is no reliable way to correlate keyboard or mouse input operations with business transactions.
- One user transaction (which would be classified as a business transaction) may spawn several other component transactions, some of which may execute locally and some remotely. Any measurement agents that exist only in the network layer or in a host (server) will not see the entire picture.
- The data may be sent through the network using various protocols, not just one, making the task of packet decoding and correlation much more difficult.
- Client/server applications can be complex, taking different execution paths and spawning different component transactions, depending on the results of previous component transactions. Every permutation could take a different form when it goes across the communication link, making it that much harder to reliably correlate network or host (server) observations with what the user sees.

In spite of these difficulties, the need to monitor distributed applications has never been greater. They are increasingly being used in mission-critical roles. An approach that solves the problems listed above is to let the application itself participate in the process. A developer knows unambiguously when transactions begin and end, both those that are visible to the user, and the component transactions that invoke transactions on remote servers.

ARMing Your Applications

With the Application Response Measurement (ARM) API, a developer can easily mark sections of an application to define business transactions. By invoking ARM API function calls at the beginning and end of each transaction, you can enable your application to be monitored by any of the measurement agents that use data generated by the ARM API. Programs executing on client or server systems can be instrumented.

By instrumenting your application to call the ARM API, you enable your application to be managed by any of the measurement agents that implement ARM. The advantage of this approach is that your application customers can choose the measurement agent that best meets their needs without your application needing to change.

System administrators will be able to answer some key questions such as:

- Is the application working correctly (available)?
- How is the application performing? What is the response time? What is the workload throughput? You will be measuring the actual service levels experienced by your users.
- Why is an application not available or performing poorly? What operation was the application performing when the problem occurred? If a remote server/application was being invoked when the problem occurred, which one?
- Who is using the application, how much are they using it, and what kind of operations are being performed? Which servers are providing the services? This information is useful for capacity planning and for charge-back accounting.

Figure 1 ARM in the Enterprise

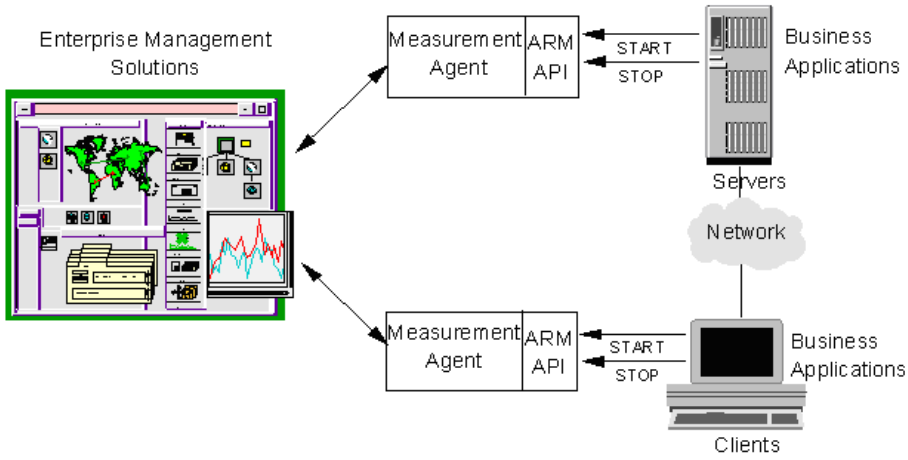


Figure 1 shows how enterprise management applications, measurement agents that implement the ARM API, and business applications that call the ARM API work together to provide a robust way to monitor application response.

Additional Information

This Guide is intended for the application developers who want to know how to instrument an application for transaction monitoring using the standard Application Response Measurement (ARM) API function calls.

The 2.0 version of the ARM Software Developer's Kit has been developed with the help of the ARM Working Group of the Computer Measurement Group (CMG).

The 2.0 ARM SDK, including this documentation, is available on CD and from the following CMG web site:

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

This web site also contains information on performance measurement agents that use the data generated by the ARM API function calls and the latest information regarding future updates or changes to the API.

A public discussion list for ARM, `cmgarm` is now available at:

cmgarm@cmg.org

To subscribe, send the following to **majordomo@cmg.org**

```
subscribe cmgarm
```

2 Basic Tasks for Instrumenting an Application

Introduction

There are three basic tasks involved in instrumenting an application with the ARM API.

1 Define the key business transactions.

This is the most important step. Each application developer needs to define who needs what kind of data, and what the data will be used for. It is common and useful for this process to be a joint collaboration between the users and developers of an application, and system and network administrators.

There are two kinds of transactions that will generally provide the greatest benefit if they are instrumented. The following procedure is suggested:

- Start with transactions that are visible to users or that represent major business operations. These are the building blocks for service level agreements, for workload monitoring, and for early problem detection.
- Next, focus on transactions that are dependent on external services, such as a database operation, a Remote Procedure Call (RPC), or a remote queue operation. These generally are components of a user/business transaction. Knowing how these types of transactions are performing can be invaluable when analyzing problems, tuning applications, and reconfiguring systems and networks.

2 Modify the application to include calls to the ARM API.

The NULL libraries and logging agent in the ARM SDK can be used for initial testing. The key is to decide where to place calls to the ARM API, by doing a good job defining the key business transactions.

3 Replace the NULL libraries or logging agent from the SDK with an ARM-compliant agent and associated management applications. The distributed applications can now be monitored in ways that previously could only be hoped for.

What to Instrument

The Application Response Measurement API is designed to instrument a unit of work, such as a business transaction, that is performance sensitive. These transactions should be something that needs to be measured, monitored, and for which corrective action can be taken if the performance is determined to be too slow.

This API is *not* designed to be a programmer profiling tool. The measurement agents using data generated by this API are designed to give application/ system managers data to understand how their environment is performing, and whether all services are available.

For information on measurement agents that do transaction monitoring, refer to the web site mentioned earlier under the section Additional Information on page 13, in Chapter 1 . Links may be found on this site to commercially available measurement agent solutions.

Some questions you may want to ask yourself when instrumenting a transaction are:

- What unit of work does this transaction define?
- Are the transaction counts and/or response times important?
- Who will use this information?
- If performance of this transaction is too slow, is there some corrective action that can take place (for example, offload work from the machine, add memory, relocate remote files, etc.)?

3 The Software Developer's Kit (SDK)

Introduction

This ARM SDK contains everything you need to prepare your application for transaction monitoring. It comes with a default no-operation (NULL) shared library that contains all the function calls you will need and a header file. The NULL library allows developers to instrument and run their applications without having one of the measurement agents installed.

Additionally, the source used to create the NULL library is part of the SDK. This is provided so a shared library can be created for applications that exist on platforms not currently supported by the measurement technologies. The SDK contains NULL libraries compiled for UNIX systems (HP-UX, IBM AIX, NCR MP-RAS, and Sun Solaris) and PC based systems (OS/2, Windows NT, and Windows 95). The kit installs the correct library for the system.

A C language header file is supplied for applications written in either C or C++.

The source code and header file for a logging agent is supplied for use in testing your instrumentation.

Sample programs for C/C++ are provided as examples of how to instrument applications. Examples for other programming languages from the ARM 1.0 SDK are also available on the CD and the web site.

The `arm.lst` file on the CD-ROM contains a detailed listing of all the files on the CD-ROM.

The ARM Shared Library (libarm)

The library specified here is a NULL shared library provided to resolve externals in the code. If you are working with a specific vendor's performance measurement agent you may want to use the `libarm` library supplied for that agent instead of the NULL library. The agent-specific library will return errors that may be helpful during development, whereas the NULL library will *always* return a non-error condition (0).

After installation `libarm.*` shared libraries reside in the directory where the system libraries are installed. For example:

HP-UX 10.x	`/usr/lib/libarm.sl`
IBM AIX	`/usr/lib/libarm.a`
Sun Solaris	`/usr/lib/libarm.so`
NCR MP-RAS	`/usr/lib/libarm.so`
Windows NT	`$windir$\SYSTEM32\LIBARM32.DLL`
Windows 95	`$windir$\SYSTEM32\LIBARM32.DLL`
OS/2 (32-bit)	`$os2dir$\DLL\LIBARM.DLL`

It is recommended that the library be used from the standard location. This is so applications can locate the library in a standard location and be able to take advantage of a measurement agent once it is installed on the system.

The Logging Agent

The source code for a logging agent, `logagent.c`, has been included for use in testing your instrumentation. The path is:

- UNIX systems

  `<install directory>/lib/logagent/logagent.c`

- PC systems

  `<install directory>/ARM_SDK/LIB/logagent/logagent.c`

Unlike the NULL libraries, it is only in source format so it needs to be compiled. For more information, see the section Using the Logging Agent on page 27, in Chapter 4.

The Header File

A C language header file, `arm.h`, is supplied for applications written in either C or C++. If you are using a language other than C or C++, the data structures and external references need to be translated to the language you are using.

Not all hardware systems or compilers provide native support for 64-bit integers – nor is there yet a standard type declaration for them. For these reasons the distributed version of the `arm.h` header file does not assume native support for 64 bit integers. However, the symbol "INT64" can be defined near the front of the file to customize the header for compilers and systems with 64 bit integer support.

4 Getting Started

Introduction

This chapter gives you the information you need to begin instrumenting your application with the ARM API function calls.

Installation

To get started, you need to install the ARM SDK files on your system. The installation process installs the appropriate NULL shared library, the header files, the shared library source code, logging agent source, documentation files and sample program files for your system.

The installation utility prompts you for a directory to install the ARM source files.

The NULL libraries for ARM 1.0 and ARM 2.0 are interchangeable, so a failure to install will have no impact. You should contact your measurement agent vendor if you need to update your agent's shared library to ARM 2.0.

For UNIX systems

1. Place the CD-ROM in the drive and mount the CD-ROM device onto your system.
2. Type **`cd <mount directory>`**.
3. Type **`./install`** (or **`./INSTALL`** for HP-UX only), then follow the prompts in the install process.

If a `libarm.*` shared library exists in the default directory, the install utility will *not* install the library. This is so the installation will not overlay an installation of one of the measurement agent's libraries. Install will not copy the library to the default (`/usr/lib`) directory if the directory is not writable by the user.

For OS/2, Windows NT, or Windows 95 systems

1. Place the CD-ROM in the drive.
2. Create a DOS window.
3. Change the current drive to the CD-ROM drive.
4. Type **INSTALL** <*drive letter:\install directory*>

   Where <*drive letter*> is the letter of the drive where you want to install the ARM SDK and <*install directory*> is the directory path for the location of where you want to install the ARM SDK. The install utility will put the files into a directory called `ARM_SDK` under the <*install directory*> specified.
5. Copy the `LIBARM*.DLL` to the standard location for the platform as shown below. Do *not* copy the library if the library already exists in the destination directory since you may be overwriting a measurement agent-specific library with a NULL library.

   - OS/2:

     copy <*install dir*>**\ARM_SDK\LIB\OS2\LIBARM.DLL**
     $os2dir$\DLL\LIBARM.DLL

   - Windows95/Windows NT:

     copy <*install dir*>**\ARM_SDK\LIB\WIN95_NT\LIBARM32.DLL**
     windir$\SYSTEM32\LIBARM32.DLL

Using the Logging Agent

The logging agent is provided for use in testing your instrumentation. It provides more information than the NULL library that only returns zeros but it does *not* function as a measurement agent.

The logging agent is provided in source format only, so it must be compiled. The logging agent source code file, `logagent.c`, can be included and compiled with an application implemented in C or it can be compiled into a library object and linked to an application.

Statically link with the logging agent and then run your application. Programmatic calls to the ARM API by the application result in the creation of a text file log (`logfile` by default) that contains a time-stamped history of the calls and the parameter values associated with those calls. See the section Testing Your Instrumentation on page 35 for a sample output file and more information on using the logging agent.

Overview of the ARM API Function Calls

The ARM API is made up of a set of function calls that are contained in a shared library. All the performance measurement agents that support the ARM API provide their own implementation of the shared library. When you insert the ARM API function calls in your application, it can be monitored by the agents that implement the shared library. The advantage of this approach is that your application customers can choose any measurement agent that best meets their needs without your application needing to change.

Table 2 ARM API Function Calls

`arm_init`	During the initialization of your application, call `arm_init` which names your application and optionally the users, and initializes the ARM environment for your application. A unique identifier is returned that must be passed to `arm_getid`.
`arm_getid`	Use `arm_getid` to name each transaction class you use in your application. This is often done during the initialization of your application. A transaction class is a description of a unit of work, such as "Check Account Balance". In each program, each transaction class may be executed once or many times. The `arm_getid` returns a unique identifier that must be passed to `arm_start`.
`arm_start`	Each time a transaction class is executed, this is a transaction instance. The `arm_start` signals the start of execution of a transaction instance and returns a unique handle to be passed to `arm_update` and `arm_stop`.

Table 2 ARM API Function Calls

`arm_update`	This is an optional function call that can be made any number of times after `arm_start` and before `arm_stop`. The `arm_update` gives information about the transaction instance, such as a "heartbeat" after a group of records has been processed.
`arm_stop`	The `arm_stop` signals the end of the transaction instance.
`arm_end`	At termination of the application call `arm_end` which cleans up the ARM environment for your application. There should be no problem if this call is not made, but memory may be wasted because it is allocated by the agent even though it is no longer needed.

Adding ARM Function Calls to an Application

The following steps show how to add ARM API function calls to an application. Also shown is a very simple application that has been instrumented with the `libarm` calls. Each numbered step below (1-4) is highlighted in the source code for the sample application that follows.

1. Once the SDK is installed, include the header file reference (`arm.h` for C and C++) in your source code and modify the compile link to reference the library.
2. Identify the start and the end of the application and place the calls to `arm_init` and `arm_end`. These calls are used for initialization and cleanup of the ARM environment for your application, and therefore should be called from the initialization and exit sections of your application.
3. Determine what transaction classes you want to instrument and the names to use to uniquely identify each transaction class. Modify the code to call `arm_getid` for each transaction class. The `arm_getid` calls can also be made from the application initialization section.
4. Call `arm_start` just prior to the start of execution of the transaction and `arm_stop` just after the transaction completes.

When distributing your application, the NULL shared library *must* be included in your installation package. By doing this you will insure that your application will load and execute correctly, even if no measurement agent is installed. If the `libarm.*` file already exists on the system where your application is being installed, do *not* overwrite the library. The library that exists may be the NULL library or it could be one of the measurement agent's libraries.

The API calls use the C calling conventions for UNIX systems, the PASCAL calling conventions for OS/2 and the _std calling conventions for Windows NT and Windows 95.

```
/*************************************************/
/*  sample.c                                     */
/*************************************************/

#include <stdio.h>
(1) #include "arm.h"
arm_int32_t appl_id = -1;       /*Unique indentifer for the
application */
arm_int32_t tran_id = -1;       /*Unique identifier for the
transaction */
void init()
{
(2) appl_id = arm_init("ARM sample program",       /*
application name */
                  "*",                        /* use default
user */
                   0,0,0);
if (appl_id < 0)
   printf("ARM sample program not registered.\n");
(3) tran_id = arm_getid(appl_id,   /*application id from
arm_init   */
                       "Sample_transaction", /* transaction
name */
                      "First Transaction in Sample program",
                        0,0,0);
  if (tran_id < 0)
     printf("Sample_transaction is not registered.\n");
} /* init */
void transaction()
{
  arm_int32_t  tran_handle;
```

```
(4)  tran_handle = arm_start(tran_id, /* transaction id from
     arm_getid */
                              0,0,0);
       /*******************************************/
       /* Perform actual transaction processing here */
       /*******************************************/
       sleep(1);

(4)  arm_stop(tran_handle,        /* transaction handle from
     arm_start */
                ARM_GOOD,        /* successful completion define
     = 0  */
                0,0,0);
       return;
     }  /* transaction */
     main()
     {
       int continue_processing = 1;
       init();
       while (continue_processing)
       {

     transaction();
       }

(2)  arm_end(appl_id,             /* application id from arm_init
     */
             0,0,0);
      return(0);
     }
```

Figure 2 ARM API Function Call Parameters

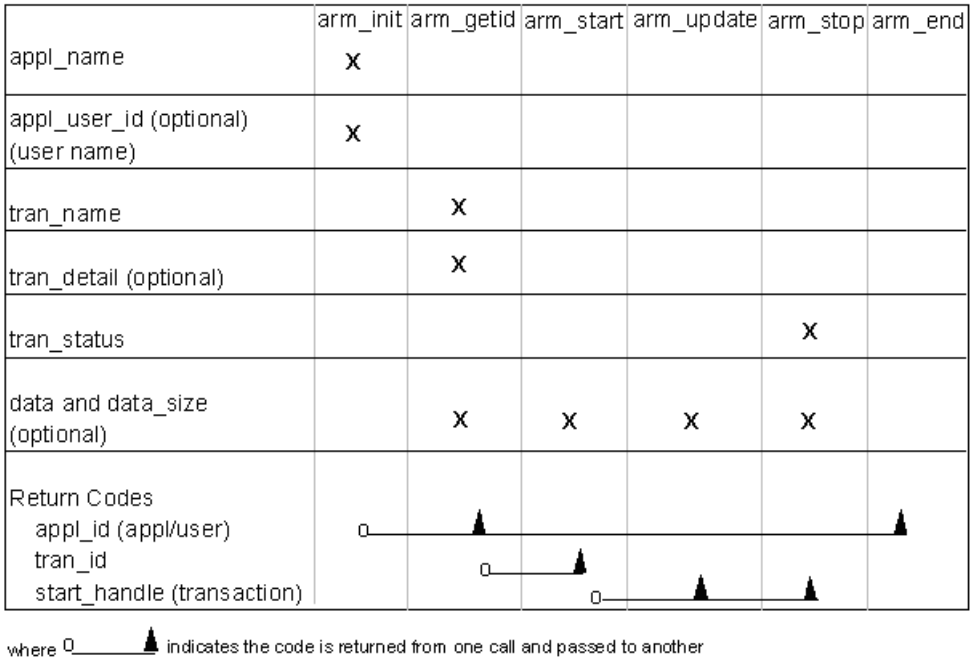


Figure 2 shows which parameters are used in each of the ARM API function calls and what is passed on from one function call to another.

Definition of Data Type Terminology

The API calls use the following terminology to define each of the parameters.

The standard API calls use the following terminology to define each of the parameters:

`arm_int32_t`	A signed 32-bit integer.
`char*`	A 32-bit pointer to a character string or data structure. Strings must be NULL terminated unless specified otherwise. Strings are expected to be displayed, put in reports, etc., so choose appropriate characters.

The more advanced functions in the API use the following terminology to define each of the parameters:

`int64`	A signed 64-bit integer.
`unsigned32`	An unsigned 32-bit integer.
`unsigned64`	An unsigned 64-bit integer.
`bit8`	A byte containing 8 single-bit flags. In this document, when a bit8 is represented as eight flags using the notation abcdefgh, a is the most significant bit, and h is the least significant bit.
`unsigned16`	An unsigned 16-bit integer.
`unsigned8`	An unsigned 8-bit integer.

These formats are in the native format of the hardware platform. This accommodates the difference between "Big-Endian" and "Little-Endian" systems, that is, the difference between hardware architectures in which the most significant bit position is on the left versus the right.

Testing Your Instrumentation

The following tasks are recommended for testing your instrumentation after you have included the ARM API calls in your program.

1. Link to the NULL library that is part of the ARM SDK. If the link fails, it means that you are not linking to the correct library, or you are using incorrect names or parameters in at least one of the ARM API calls.
2. Once you can link successfully, then run your application, including the calls to the API, and verify that your application performs correctly. No testing of the API calls is done except for the linking parameters, because the NULL library simply returns zero every time it is called. Running the application is useful to insure that you did not inadvertently alter the program in a way that affects its basic function.
3. Compile the logging agent source, `logagent.c`, if you have not already.
4. Link to the logging agent generated in the previous step. Run your application, including the calls to the ARM API and verify that your application performs correctly.
5. Manually review the log created by the logging agent to verify that the correct parameters are passed on each call. These parameters include transaction ids to connect start calls to the correct transaction class, start handles to connect stop calls to the correct start calls, and any of the optional parameters. Optional advanced parameters include correlators that indicate the parent/child relationship between transactions and components, and metrics about the transaction or application state.

   Search the log for error messages (identified by "ERROR" in the text) and informative messages (identified by "INFO" in the text) after your application has run for a considerable period of time in a simulated production environment. Upon successful completion of this test, you should be confident that your ARM API calls are correct. A sample log is provided on the next page.
6. Link to a performance measurement product (if available) and run the application under typical usage scenarios. This will test the entire system of application plus management tools.

Logging Agent Sample Output

```
7:47:39.sss: arm_init: Application <Appl_0> User <User_0> =
Appl_id <1>

17:47:39.sss: arm_getid: Application <Appl_0> User <User_0>
Transaction <Tran_0> Detail <This is transaction type 0>

17:47:39.sss: arm_getid: Application <Appl_0> User <User_0>
Transaction <Tran_0> = Tran_id <1>

17:47:39.sss: arm_getid: Application <Appl_0> User <User_0>
Transaction <Tran_0> Metric Field <1> Type <1> Name <This is
a Counter32 user metric            >

17:47:39.sss: arm_start: Application <Appl_0> User <User_0>
Transaction <Tran_0> = Start_handle <1>

17:47:39.sss: arm_start: Application <Appl_0> User <User_0>
Transaction <Tran_0> Start_handle <1> Metric < This is a
Counter32 user metric            > : <0>

17:47:40.sss: arm_update: Application <Appl_0> User <User_0>
Transaction <Tran_0> Start_handle <1> Metric < This is a
Counter32 user metric            > : <2>

17:47:41.sss: arm_stop: Application <Appl_0> User <User_0>
Transaction <Tran_0> Start_handle <1> Status <0>

17:47:41.sss: arm_stop: Application <Appl_0> User <User_0>
Transaction <Tran_0> Start_handle <1> Metric < This is a
Counter32 user metric            > : <4>

17:47:41.sss: arm_end: Application <Appl_0> User <User_0>
appl_id <1>
```

arm_init

Use `arm_init` to define the application or a unique instance of the application and user. You *must* call `arm_init` before any other ARM API calls. It is often called when an application initializes. The return code is an application/user identifier that is input as a parameter on the `arm_getid` to associate transactions with the application.

Each application needs to be identified by a unique name. It is your responsibility to choose a name that is meaningful, and that will not likely duplicate the names other developers will choose for their applications. Suggestions for names would be the product name and version number or a project name.

There can be any number of application instances executing simultaneously that use the same application name, or the same application and user names. A measurement agent may assign a unique application identifier to each application instance, or it may assign an identifier that is shared across identically named instances.

Syntax:

```
appl_id=arm_init(appl_name,appl_user_id,flags,data,\
data_size)
```

Parameters:

`appl_name (char*)`	The name used to identify the application. The maximum length is 128 bytes including the NULL string terminator.
`appl_user_id (char*)`	The name of the application user. On UNIX and Windows NT you can set this value to `*` to indicate the login user ID of the person running the application. The maximum length is 128 bytes including the NULL string terminator. If you do not provide a value for this parameter, you must specify the NULL value (0).
`flags (arm_int32_t)=0`	Reserved for future use. It must be set to zero.
`data (char*)=0`	Reserved for future use. A NULL value (0) must be used.
`data_size (arm_int32_t)=0`	Reserved for future use. It must be set to zero.

Return Code:

`appl_id (arm_int32_t)`	A unique value to reference an application/user identifier. This ID must be passed to the `arm_getid` call.

Example:

```
        my_appl_id = arm_init ("Parts Inventory Manager 1.1", /
* appl name  */
                              "*",
             /* user id     */
                              0, 0, 0);                    /*
reserved for future use */
```

Error Handling:

If the value returned in `appl_id` is less than zero, an error occurred in communicating with the measurement agent. The value returned on an error can be passed to `arm_getid` which will cause `arm_getid` to function as a NULL operation. The error should be logged so corrective action can be taken.

arm_getid

The arm_getid function call is used to assign a unique identifier to a transaction class, and optionally to describe the format of additional data passed on arm_start, arm_update, and arm_stop calls. This is often done during the initialization of your application. The identifier returned by arm_init is passed as a parameter in arm_start calls to identify which class of transaction is starting.

A transaction class is a description of a unit of work, such as "Check Account Balance". Any number of transaction classes can be defined within each application. The transaction class name should help a person understand what function the transaction performs. The call to arm_getid need be made only once for each transaction class each time the application is started. A call to arm_getid can be made with the same information as a previous call, in which case the transaction identifier (tran_id) that is returned will be the same as the previous calls. Four types of information are tested to see if the information is the same. If any of these are different, a different tran_id will be returned.

- The application identifier (appl_id).
- The transaction name (tran_name).
- The data pointer (data) was NULL on previous calls and is not NULL, or it was not NULL on previous calls and now it is NULL.
- If the data pointer (data) is not NULL on previous calls and this call, and the contents and size (data_size) of the buffer pointed to by the data parameter differ.

Any number of transaction classes can be defined within each application. In each application, each transaction class may be executed any number of times. Each time a transaction class is executed (via arm_start), it is called a transaction instance. There can be any number of instances of each transaction class executing simultaneously.

Syntax:

```
tran_id=arm_getid(appl_id,tran_name,tran_detail,flags,\
data,data_size)
```

Parameters:

`appl_id (arm_int32_t)`	The unique reference to an application/user identifier returned from the `arm_init` call. If the `appl_id` is less than zero, this `arm_getid` call will be treated as a NULL operation, and a negative `tran_id` returned.
`tran_name (char*)`	The unique name of the transaction class. It is defined for each transaction class by the application developer. It must be unique within the application (for each `arm_init` call). The maximum length is 128 bytes including the NULL string terminator.
`tran_detail (char*)`	Transaction detail allows a developer to provide additional information about a transaction class. It is a free-form text area that is set once for each `appl_id/tran_name` pair. If the contents of the field change on later calls using the same `appl_id/tran_name` pair, the new contents are ignored. The maximum length is 128 bytes including the NULL string terminator. If no `tran_detail` is associated with this transaction, you must specify the NULL value (0).
`flags (arm_int32_t)=0`	Reserved for future use. It must be set to zero.
`data (char*)`	A pointer to a buffer that describes the format of additional data that can be passed on `arm_start`, `arm_update`, and `arm_stop` calls. If no additional data is passed on these calls, this parameter must be set to zero (0). See the section Format of Data Buffer in arm_getid in Chapter 5 for the detailed buffer format.
`data_size (arm_int32_t)`	The length in bytes of the buffer pointed to by `data`. If `data` is set to zero (0), `data_size` must also be set to zero.

Return Code:

`tran_id (arm_int32_t)` The unique identifier assigned for this transaction class. This ID needs to be passed on `arm_start` calls.

Example:

```
  my_tran_id = arm_getid (my_appl_id,    /* application name
*/
                "Part Number Query",   /* transaction name
*/
                     "Call to Server XYZ",  /* transaction
details     */
                      0,                    /* reserved for
future use */
                      my_buffer_ptr,        /* metrics data/
metrics meta-data */
                      my_buffer_length);    /* length of data
buffer   */
```

Error Handling:

If the value returned in `tran_id` is less than zero, an error occurred in communicating with the measurement agent. The most likely cause is passing an invalid value for `appl_id`. The value returned on an error can be passed to `arm_start` which will cause `arm_start` to function as a NULL operation. The error should be logged so corrective action can be taken.

arm_start

Use `arm_start` to mark the beginning of execution of a transaction. Each time a transaction executes, it is called a transaction instance. You *must* call `arm_start` in your application at the beginning of each transaction instance you want monitored.

Additional information about the transaction can be provided in the optional data buffer. If no additional information is provided, pass a null pointer. This information can be provided on any or all of the `arm_start`, `arm_update`, and `arm_stop` calls, except correlation information which is passed only on `arm_start`. Chapter 5, Advanced Topics for details on how to pass this information.

Syntax:

```
start_handle=arm_start(tran_id,flags,data,data_size)
```

Parameters:

`tran_id (arm_int32_t)`	The unique identifier assigned to the transaction class. This is the ID generated by `arm_getid`. If the `tran_id` is less than zero, this `arm_start` call will be treated as a NULL operation, and a negative `start_handle` returned.
`flags (arm_int32_t)=0`	Reserved for future use. It must be set to zero.
`data (char*)`	A pointer to a buffer with additional data that can optionally be passed. If no additional data is passed, this parameter must be set to zero (0). See the section Format of Data Buffer in arm_start, arm_update, arm_stop on page 72, in Chapter 5 for the detailed buffer format.
`data_size (arm_int32_t)`	The length in bytes of the buffer pointed to by the `data` parameter. If `data` is set to zero (0), `data_size` must also be set to zero.

Return Code:

`start_handle (arm_int32_t)`	The unique transaction handle assigned to this instance of a transaction. This handle must be passed on `arm_stop` and any `arm_update` calls.

Example:

```
    my_handle = arm_start (my_tran_id,        /* transaction
handle      */
                           0,                 /* reserved
for future use */
                            my_buffer_ptr,     /* metrics
data/correlator */
                          my_buffer_length); /* length of
data buffer   */
```

Error Handling:

If the value returned in `start_handle` is less than zero, an error occurred in communicating with the measurement agent. The most likely cause is passing an invalid value for `tran_id`. The value returned on an error can be passed to `arm_update` and `arm_stop` calls, which will cause these calls to function as NULL operations. The error should be logged so corrective action can be taken.

arm_update

Use `arm_update` for the following purposes. This is an optional call.

- To show the progress of a long transaction.

  Put the `arm_update` call into your application program after `arm_start` and before `arm_stop` each time you want to send a "heartbeat" indicating that the transaction instance is still running. This would typically be done after a fixed interval of time (such as every minute) or after a fixed amount of work is completed (such as 1000 records processed). There can be any number of `arm_update` calls between an `arm_start/arm_stop` pair. This call is most useful for long-running transactions that take minutes or hours to complete. Another way to capture data about the steps within a long transaction is to use component transactions. For more information, see the sectionThree Ways to Instrument within a Transaction Instance on page 78 in Chapter 5.

  The `arm_update` is also useful for updating any of the metric or string variables passed in the buffer pointed to by the `data` parameter (as defined in `arm_getid`). This could be used to show not only that the transaction is progressing, but also how far it has progressed. For example, every time another 1000 records are processed, an `arm_update` call could be made with an updated count in the buffer.

- To provide extra information about a transaction.

  Put the call into your application program after `arm_start` and before `arm_stop` each time you want to provide special information about a transaction instance. If there is no additional information to be provided, pass a null pointer. There are several types of additional information that may be useful: information about the size of the transaction (such as the number of bytes in a print job), information about the state of the application (such as the number of threads that are running), and diagnostic information. This type of information can be provided via application-defined metrics on any or all of the `arm_start`, `arm_update`, and `arm_stop` calls. See Table 3 on page 73, in Chapter 5 for the detailed buffer format.

- To provide a larger opaque application private buffer.

  Information that does not conform well to application-defined metrics (for example long diagnostic messages) may be provided via an opaque buffer containing up to 1020 bytes of data (Format 2). Except for the four-byte Format field the content of the buffer is entirely up to the application developer. Because the contents of the buffer containing the information is known only to the application developer, measurement agents cannot do much with the data in this field. A typical measurement agent might provide an option to write the buffer with the information to a log file, but this is the most that can be expected.

  Measurement agents are *not* required to do anything with the information in this call.

Syntax:

```
error_status=arm_update(start_handle,flags,data,data_size
)
```

Parameters:

`start_handle (arm_int32_t)`	The unique handle from the `arm_start` call that marked the start of this transaction instance. The `start_handle` must be passed in each `arm_update` call. Many transaction instances may be executing at the same time from this and other applications, so this handle is essential to identify which transaction instance is being updated. If `start_handle` is less than zero, this `arm_update` call will be treated as a NULL operation, and a negative `error_status` returned.
`flags (arm_int32_t)=0`	Reserved for future use. It must be set to zero.
`data (char*)`	A pointer to a buffer with additional data that can optionally be passed. If no additional data is passed, this parameter should be set to zero (0). There are two possible buffer formats: If the `Format` field contains the value `1`, then application-defined metrics as defined in `arm_getid` can be passed. The `Correlator` field is *not* used in the `arm_update` call. If the `Format` field contains the value `2`, then a status message up to 1020 bytes in length may be passed in. See the section Format of Data Buffer in arm_start, arm_update, arm_stop on page 72, in Chapter 5 for the detailed buffer formats.
`data_size (arm_int32_t)`	The length in bytes of the buffer pointed to by `data`. If `data` is set to zero (0), `data_size` should also be set to zero.

Return Code:

`error_status (arm_int32_t)`	Contains a zero if successful and a negative value if an error occurred.

Example:

```
      status = arm_update (my_handle,     /* transaction
handle        */
                           0,                /* reserved for
future use   */
                             my_buffer_ptr,     /* data
description          */
                             my_buffer_length); /*length of
data description */
```

Error Handling:

If the value returned in `error_status` is less than zero, an error occurred in communicating with the measurement agent. The most likely cause is passing an invalid value for `start_handle`. The error should be logged so corrective action can be taken.

arm_stop

Use `arm_stop` to mark the end of a transaction instance that was started with `arm_start`. Call `arm_stop` from your application program just after each transaction instance ends.

In addition to signaling the end of the transaction instance, which allows a measurement agent to calculate the elapsed time since the `arm_start`, additional information about the transaction can be provided in the optional data buffer. This information can be provided on any or all of the `arm_start`, `arm_update`, and `arm_stop` calls.

Syntax:

```
error_status=arm_stop(start_handle,tran_status,flags,dat
a,\ data_size)
```

Parameters:

`start_handle (arm_int32_t)`	The unique handle from the `arm_start` call that marked the start of this transaction instance. The `start_handle` must be passed in each `arm_stop` call. Many transaction instances may be executing at the same time from this and other applications, so this handle is essential for the measurement agent to use to identify which transaction instance is stopping. If `start_handle` is less than zero, this `arm_stop` call will be treated as a NULL operation, and a negative `error_status` returned.

`tran_status (arm_int32_t)`	The completion code of the transaction, as determined by the application. 0 = Transaction successful (defined as `ARM_GOOD` in `arm.h`). Use this value when the operation completed normally and as expected. 1 = Transaction aborted (defined as `ARM_ABORT` in `arm.h`). Use this value when there was a fundamental failure in the system. For example, a timeout from a communications protocol stack, or an error when doing a database operation. 2 = Transaction failed (defined as `ARM_FAILED` in `arm.h`). Use this value in applications where the transaction worked properly, but no result was generated. For example, when making an airline reservation, a server indicates no seats are available on the requested flight. Since no reservation was made, the transaction was not successful; but since the reservation system is operating correctly, it is not an aborted transaction either. In this case, you might want to record the transaction as a failed transaction.

`flags (arm_int32_t)=0`	Reserved for future use. It must be set to zero.
`data (char*)`	A pointer to a buffer with additional data that can optionally be passed. If no additional data is passed, this parameter should be set to zero (0). The format is identical to the `arm_start` call, except the `Correlator` field is not used in the `arm_stop` call. See the section Format of Data Buffer in arm_start, arm_update, arm_stop on page 72, in Chapter 5 for the detailed buffer format.
`data_size (arm_int32_t)`	The length in bytes of the buffer pointed to by the `data` parameter. If `data` is set to zero (0), `data_size` should also be set to zero.

Return Code:

`error_status (arm_int32_t)`	Contains a zero if successful and a negative value if an error occurred.

Example:

```
    status = arm_stop (my_handle,    /* transaction handle
*/
                              ARM_GOOD,          /* transaction
status         */
                             0,                /* reserved for
future use    */
                              buffer_ptr,        /* data
description           */
                           buffer_length);    /* length of data
description */
```

Error Handling:

If the value returned in `error_status` is less than zero, an error occurred in communicating with the measurement agent. The most likely cause is passing an invalid value for `start_handle`. The error should be logged so corrective action can be taken.

arm_end

Use arm_end when you are finished initiating new activity using the ARM API. It is typically called when an application/user instance is terminating. Each arm_end is paired with one arm_init to mark the end of an application.

An arm_end is a signal from the application that it does not intend to issue any more arm_getid calls using this appl_id, or any arm_start calls using any tran_id defined using this appl_id. After arm_end, the measurement agent may ignore any arm_getid or arm_start calls. It is acceptable to call arm_update or arm_stop for any incomplete transaction instances started with arm_start.

Syntax:

```
error_status=arm_end(appl_id,flags,data,data_size)
```

Parameters:

appl_id (arm_int32_t)	A unique reference to an application/user identifier returned from the arm_init call. If appl_id is less than zero, this arm_end call will be treated as a NULL operation, and a negative error_status returned.
flags (arm_int32_t)=0	Reserved for future use. It must be set to zero.
data (char*)=0	Reserved for future use. A NULL pointer (0) must be used.
data_size (arm_int32_t)=0	Reserved for future use. It must be set to zero.

Return Code:

error_status	Contains a zero if successful and a negative value if an error occurred.

Example:

```
      status = arm_end (my_appl_id,   /* transaction handle
*/
                                      0,0,0);      /* reserved
for future use */
```

Error Handling:

If the value returned in `error_status` is less than zero, an error occurred in communicating with the measurement agent. The most likely cause is passing an invalid value for `appl_id`. The error should be logged so corrective action can be taken.

5 Advanced Topics

Introduction

The following topics provide information on more advanced implementations using the ARM 2.0 API.

Additional Data Passed in the ARM Function Calls

The following two types of additional data can now be provided via the ARM 2.0 API.

- Transaction correlation data

  You can indicate that a transaction is a component of another transaction. You can do transaction correlation within one system or across multiple systems. This permits a better understanding of the overall transaction, how much time each part of the transaction is taking, and where problems are occurring.

- Application-defined metrics

  Application-defined metrics provide additional information about the transaction, such as the number of bytes or records being processed, or about the state of the application at the moment that the transaction is being processed, such as the length of a work queue. This information is useful to better understand response times, and how the application can be tuned to perform better.

Transaction Correlation

Many client/server transactions consist of one transaction visible to the user, and any number of nested component transactions that are invoked by the one visible transaction. These component transactions are the children of the parent transaction (or the child of another child component transaction). It is very useful to know how much each component transaction contributes to the total response time of the visible transaction. Similarly, a failure in one of the component transactions will often lead to a failure in the visible transaction, and this information is also very useful.

There are two facilities that the application developer can use to provide this information to measurement agents that implement the ARM 2.0 API.

1. On the same `arm_start`, the application can request that the measurement agent assign and return a correlator for this instance of the transaction (that is a parent correlator). Note that the agent has the option of not providing the correlator, because it may not support the capability (ARM Version 1.0 agents do not support correlators), or because it is operating under a policy to suppress generating them.
2. When indicating the start of a child transaction with an `arm_start`, the application can provide a correlator provided from a parent transaction. This allows the measurement agent to know the parent/child relationship.

Figure 3 Transaction Response Time Correlation

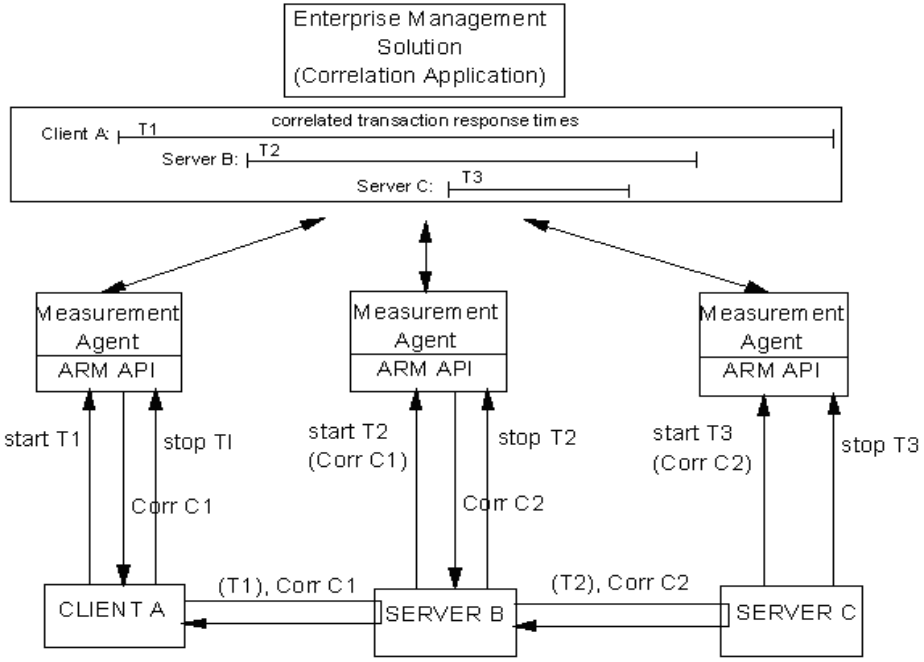


Figure 3 shows the concept for a simple model. The principle can be extended to a model of arbitrary complexity.

- Client A starts transaction T1, requesting a correlator via `arm_start`, and is assigned C1.
- Client A sends a request (T1) to Server B, and includes C1 in the request.
- Server B starts transaction T2, passing C1 as the parent. At the same time it requests a correlator and is assigned C2.
- Server B sends a request (T2) to Server C, and includes C2 in the request.
- Server C starts transaction T3, passing C2 as the parent.
- T3 stops, T2 stops, and T1 stops.

If the correlation application collects all the data about these transactions, it can put together the total picture, knowing that T1 is the parent of T2 (via C1), and T2 is the parent of T3 (via C2). The parent/child relationship could be from a client to a server, or within one program.

An application using the ARM API need not be concerned with the format of the correlators. Measurement agents generate correlators.

Changes Needed in the Applications for Transaction Correlation

Each application responsible for a component of the overall transaction (client and server) will require some modifications. Applications have three responsibilities:

- request correlators for transactions with one or more child transactions (via `arm_start`) by getting the appropriate flag.
- send the assigned correlators to the child transaction(s) along with the data needed to invoke the child transaction(s) itself. This is done by first checking that the agent assigned a correlator, and then sending the number of bytes in the correlator. The length is stored by the agent in the `Correlator Length` field.
- pass correlators received from parent transactions to the measurement agents (via `arm_start`) by storing the correlator in the optional buffer and setting the appropriate flag.

To enable a correlation application to analyze the correlators coming from different systems, measurement agents follow conventions when creating correlators. Included within the correlator is information identifying the system, the transaction class (from `arm_getid`), the transaction instance (from `arm_start`), and some flags. The format is flexible and extendible so more conventions can be added as the need arises. See Appendix A, Measurement Agent Information for information on the correlator format.

Correlators are passed in the `arm_start` calls by utilizing the `data` buffer. This same `data` buffer is used to pass application-defined metrics, as described in the section Format of Data Buffer in arm_start, arm_update, arm_stop on page 72. Correlators are ignored in `arm_update` and `arm_stop` calls.

If a correlator is being requested, the `data` buffer should be 256 bytes, to allow for a variable size correlator. If a correlator is being passed to the measurement agent, and none is requested, the length may be truncated based on the correlator length.

If you only wanted to do transaction correlation in your application and not provide application-defined metrics, you can zero out the metrics (set the Flags Second Byte to zero and fill with zeros 80 bytes for the metrics descriptions).

Other than the length, the correlator format need *not* be understood by the application developer, as it is opaque.

Application-Defined Metrics

Application-defined metrics can tell you more about the transaction or about the state of the application at the moment that the transaction is being processed. Three likely uses are envisioned as described below:

1. Specify characteristics of the transaction that will affect the response time, or that are useful for workload planning.

   Examples are the number of bytes in a file transfer or print job, or the number of records being processed. A file transfer of 100 megabytes would certainly be expected to take longer than a transfer of 100 kilobytes.

2. Specify information about the current state of the application.

   Examples would be the length of a workload queue, the amount of memory allocated, or the number of threads being used. This information is useful for adjusting workloads by shifting work between systems, or tuning the application. If a comparison of response times versus threads shows that congestion builds and response times increase dramatically if, for example, eight threads are used instead of twelve, the application can be recompiled or instructed to use more threads, which may result in a dramatic improvement in performance.

3. Specify information that can be used in diagnosing problems.

   Examples are error codes returned from services invoked by the application, or information about the transaction itself such as the part number being processed.

In setting up application-defined metrics, arm_getid is used to define the context (or "meta-data") for a buffer of values that can be passed at arm_start, arm_update or arm_stop. Actual values are passed in arm_start, arm_update and arm_stop. The length of the buffer is specified in the data_size parameter.

Choosing A Data Type

The additional data provided in the data buffer uses metric and/or string fields (see later sections for information on the format of the data buffer). Four general data types can be specified for each field (counter, gauge, numeric ID and string). This section provides some suggestions about which data type to use.

Counter

A counter should be used when it makes sense to sum up the values over an interval. Examples are bytes printed and records written. The values can also be averaged, maximums and minimums (per transaction) can be calculated, and other kinds of statistical calculations can be performed.

If a counter is used, its initial value must be set in the arm_start call. The difference between the value in the arm_start and the arm_stop (or the value in the last arm_update call if no metric value is passed in arm_stop), equals the amount attributed to this transaction. Similarly, the difference between successive arm_update calls, or from the arm_start to the first arm_update call, or from the last arm_update to the arm_stop call, equals the value for the time period between the calls.

Here are three examples of how a counter would probably be used:

- The counter is set to zero at arm_start and to some value at arm_stop (or the last arm_update call). In this case, the application probably measured the value for this transaction and provided that value in the arm_stop call. The application always sets the value to zero in the arm_start call so the value at arm_stop reflects both the difference from the arm_start value and the absolute value.
- The counter is x1 at arm_start, x2 at its arm_stop, x2 at the next arm_start, and x3 at its arm_stop. In this case, the application is probably keeping a rolling counter. Perhaps this is a server application that counts the total workload. The application simply takes a snapshot of

the counter at the start of a transaction and another snapshot at the end of the transaction. The agent determines the difference attributed to this transaction.

- The counter is x1 at `arm_start`, x2 at `arm_stop`, x3 (not equal to x2) at the next `arm_start`, and x4 at `arm_stop`. In this case, the application is probably keeping a rolling counter as in the previous example. But in this case the measurement represents a value affected by other users or transaction classes, so the value often changes from one `arm_stop` to the next `arm_start` for the same transaction class.

Gauge

A gauge should be used instead of a counter when it is not meaningful to sum up the values over an interval. An example is the amount of memory used. If you were measuring the amount of memory used over 20 transactions in an interval and the average usage for each of these transactions was 15 MB, it does not make sense to say that 20*15=300 MB of memory used over the interval. It would make sense to say that the average was 15 MB, that the median was 12 MB, and that the standard deviation was 8 MB. These are the kinds of operations that an agent will typically apply to gauges. The values can also be averaged, maximums and minimums per transaction calculated, and other kinds of statistical calculations performed.

Gauges can be provided on `arm_start`, `arm_update`, and `arm_stop` calls. This creates the potential for different interpretations. If several values are provided for a transaction (one on an `arm_start`, one on `arm_update`(s), and one on an `arm_stop`), which one(s) should be used? In order to have consistent interpretation, the following conventions apply. Measurement agents are free to process the data in any way within these guidelines.

- The maximum value for a transaction will be the largest valid value passed at any time during the transaction.
- The minimum value for a transaction will be the smallest valid value passed at any time during the transaction.
- The mean value for a transaction will be the mean of all valid values passed at any time during the transaction. All values will be weighted equally.
- The median value for a transaction will be the median of all valid values passed at any time during the transaction. All values will be weighted equally.

- The last value for a transaction will be the last valid value passed at any time during the transaction.

Numeric ID

A numeric ID is simply a numeric value that is used as an identifier, and *not* as a measurement value. Examples are message numbers and error codes. It is *not* meaningful to sum, average, or manipulate these values in any arithmetic way. By using numeric ID instead of a gauge or counter, the application indicates this to the measurement agent. An agent could create statistical summaries based on these values, such as generating a frequency histogram by error code, but this is done by counting the numbers, *not* by summing them or performing any other arithmetic operation.

String

A measurement agent should process a string in the same way as a numeric ID. As with numeric ids it is *not* meaningful to do arithmetic operations on a string value.

Format of Data Buffer in arm_getid

Format	4 bytes	101 (`arm_int32_t`) (identifies "meta-data" format)

Flags
The flags indicate which metric and string descriptions are included in the buffer.
	4 bytes	First Byte (bit8) = 0
		Second Byte (bit8)
		`abcdefg0`, where `a` through `g` each denote the value of a bit flag:
		`a = 1 if there is a description for Metric #1, otherwise a = 0`
		`b = 1 if there is a description for Metric #2, otherwise b = 0`
		`c = 1 if there is a description for Metric #3, otherwise c = 0`
		`d = 1 if there is a description for Metric #4, otherwise d = 0`
		`e = 1 if there is a description for Metric #5, otherwise e = 0`
		`f = 1 if there is a description for Metric #6, otherwise f = 0`
		`g = 1 if there is a description for String #1, otherwise g = 0`
		
		Third Byte (bit8) = 0
		Fourth Byte (bit8) = 0

Metric #1 Description	48 bytes	The first 4 bytes (`arm_int32_t`) define the type of data that will be passed in the 8 byte field. See the description below this table for an explanation of the different data types.
		`1 = ARM_Counter32`
		`2 = ARM_Counter64`
		`3 = ARM_CntrDivr32`
		`4 = ARM_Gauge32`
		`5 = ARM_Gauge64`
		`6 = ARM_GaugeDivr32`
		`7 = ARM_NumericID32`
		`8 = ARM_NumericID64`
		`9 = ARM_String8`
		The last 44 bytes (`char*`) are the name of the metric. This is a NULL terminated character string. A possible use of this name is to display it along with the current value, either on a user interface or in a report.

Metric #2 Description	48 bytes	Same as Metric Description #1.
Metric #3 Description	48 bytes	Same as Metric Description #1.
Metric #4 Description	48 bytes	Same as Metric Description #1.
Metric #5 Description	48 bytes	Same as Metric Description #1.
Metric #6 Description	48 bytes	Same as Metric Description #1.
String #1 Description	48 bytes	The first 4 bytes (`arm_int32_t`) define the type of data that will be in the field. Only one data type is valid in this field. `10 = ARM_String32` The last 44 bytes (`char*`) are the name of the String #1 field. It is a NULL terminated character string. A possible use of this name is to display it along with the current value, either on a user interface or in a report.

Data Type Definitions

ARM_Counter32	An unsigned32 value that increases up to the maximum value that the counter can hold, at which point it resets to zero and continues counting up from zero. Except for the reset back to zero, the value can never decrease. The counter is in the first four bytes, and the second four bytes are unused.
ARM_Counter64	An unsigned 64 counter (see ARM_Counter32, except it is 64 bits long).
ARM_CntrDivr32	A combination of two unsigned32 integers, with ARM_Counter32 in the first four bytes, and an unsigned32 divisor in the second four bytes. The total value is ARM_CntrDivr32. The purpose of this format is to be able to represent decimal values without using floating point formats.
ARM_Gauge32	An arm_int32_t (signed) value that can increase or decrease. The gauge is in the first four bytes, and the second four bytes are unused.
ARM_Gauge64	An int64 (signed) gauge (see ARM_Gauge32, except it is 64 bits long).
ARM_GaugeDivr32	A combination of two integers, one an arm_int32_t (signed) and one an unsigned32. ARM_Gauge32 is in the first four bytes, and an unsigned32 divisor in the second four bytes. The total value is ARM_GaugeDivr32. The purpose of this format is to be able to represent decimal values without using floating point formats.

`ARM_NumericID32`	An `unsigned32` value that should *not* be used in arithmetic operations because it is used as an identifier, *not* as a measurement. For example, a message number or error code. The numeric ID is in the first four bytes, and the second four bytes are unused.
`ARM_NumericID64`	An `unsigned64` value that should *not* be used in arithmetic operations because it is used as an identifier, *not* as a measurement. An example is a message number or error code.
`ARM_String8,`	An 8 byte string that is not NULL terminated. If the string is less than eight bytes long, it must be padded with blanks. The character set is ASCII or EBCDIC, depending on whatever is standard for that platform. Unlike the NULL terminated character strings passed in various places in the API, these strings cannot be reliably converted to other code pages, so it is suggested you use only the common characters in the first 128 characters of the Latin code pages. See the section Internationalization on page 81 for more information.
`ARM_String32,`	A 32 byte string that is *not* NULL terminated. If the string is less than 32 bytes long, it must be padded with blanks. The character set is ASCII or EBCDIC, depending on whatever is standard on that platform. Unlike the NULL terminated character strings passed in various places in the API, these strings cannot be reliably converted to other code pages, so it is suggested you use only the common characters in the first 128 characters of the Latin code pages. See the section Internationalization on page 81 for more information.

Format of Data Buffer in arm_start, arm_update, arm_stop

Table 3

Format	4 bytes	1 (arm_int32_t) (2 is a special format for arm_update, see Table 4).
Flags The flags indicate which fields are included in the buffer.	4 bytes	First Byte (bit8) (Only valid for arm_start. Ignored on arm_update and arm_stop.) abcd0000, where a, b, c, d each denote the value of a bit flag. a, b, d are set by the application. c is set by the measurement agent. a = 1 if the application is passing the correlator from a parent transaction in the Correlator field; otherwise a = 0. b = 1 if the application is requesting that the agent generate a correlator for the transaction (the one indicated by this arm_start command); otherwise b = 0. If a correlator is being requested, the data buffer should be 256 bytes, to allow for a variable size correlator.c = 1 if the agent is returning a correlator in the Correlator field. When set, the value in the Correlator field overlays any previous value. This flag will only be set when three conditions are met, otherwise c = 0: The application has set bit b = 1. The agent supports this function (agents that only support version 1.0 of the ARM API do not). The agent is running in a mode where the generation of correlators is enabled (that is, there might be an installation policy to disable the generation of correlators, either temporarily or permanently). If this bit is not set to 1, there is no correlator, and therefore the application should not forward the contents of the Correlator field.

Table 3

		d = 1 if the application is requesting that the agent trace this transaction. This might be done when a dummy test transaction is being executed, or when an error has occurred. Each agent can choose how and if it should honor the request, and administrators who configure the agent may establish the policy. Second Byte (bit8) abcdefg0, where a through g each denote the value of a bit flag: a = 1 if a value is passed in Metric #1, otherwise a = 0 b = 1 if a value is passed in Metric #2, otherwise b = 0 c = 1 if a value is passed in Metric #3, otherwise c = 0 d = 1 if a value is passed in Metric #4, otherwise d = 0 e = 1 if a value is passed in Metric #5, otherwise e = 0 f = 1 if a value is passed in Metric #6, otherwise f = 0 g = 1 if a value is passed in String #1, otherwise g = 0 It is perfectly permissible for an application to pass none or some of the metrics on each call, and to change which metrics are passed from call to call. This holds true for arm_start, arm_update, and arm_stop calls.The one requirement that must be adhered to is that the meaning and position of the field must have been defined with the arm_getid call. For more information, see the section Format of Data Buffer in arm_getid. Third Byte (bit8)=0 Fourth Byte (bit8)=0

Table 3

`Metric#1`	8 bytes	The metric fields are used by the application to pass useful information about the transaction or the state of the application to the measurement agent. The field contains one or two integers, or a string variable. The use of the field and the format of the field are determined by the buffer passed on the `arm_getid` call. For more information, see the sectionFormat of Data Buffer in arm_getid. See the sections Choosing A Data Type, andData Type Definitions for more information.
`Metric#2`	8 bytes	Same as `Metric #1`.
`Metric#3`	8 bytes	Same as `Metric #1`.
`Metric#4`	8 bytes	Same as `Metric #1`.
`Metric#5`	8 bytes	Same as `Metric #1`.
`Metric#6`	8 bytes	Same as `Metric #1`.

Table 3

`String#1`	32 bytes	A string variable of up to 32 characters. The string is not NULL terminated, and is padded with blanks if it is less than 32 characters. Any information can be included in the string. Examples would be a part number being processed, or an error code.
`Correlator`		The field has two different uses depending on whether it is passed on the call from the application to the measurement agent, or if it is passed in the return from the agent: The application can pass in the correlator from a parent transaction to the agent. This allows the agent to correlate the parent transaction to the component transaction being started with this `arm_start` call. The agent can return a correlator for the transaction being started by this `arm_start` call. The application could then pass this correlator to applications that it invokes, and they in turn could pass it as the parent correlator in `arm_start` calls that they make. If the correlator returned bit is set (Flags First Byte `c = 1`), the application can either pass the entire 168 byte correlator. Or if you want to optimize, the application can choose to read the correlator length field and only pass the number of bytes containing data, starting with the 2 bytes of the correlator length. See the section Transaction Correlation for more information on correlating transactions. See Appendix A, Measurement Agent Information for more information on the content of the correlator.

Table 3

	Length 2 bytes Data 0-166 bytes	The `Correlator length` field (unsigned 16) specifies the length of a correlator (including this field) generated by a measurement agent (when bit `c` is set in the first Flags byte). If this value is zero, it means that the agent is not returning a correlator, and therefore there is not any reason to pass this correlator on to other parts of the application (or servers that it calls). This field is considered a part of the correlator and must be included in the forwarded correlator data. The `Correlator data` field is used to show the parent/child relationship between transactions. Note that the application instrumenter need not understand the correlator format as it is "opaque".

In the `arm_update` calls with a `Format` field containing the value `2`, the buffer may have the following format:

Table 4 Format 2

`Format`	4 bytes	2 (`arm_int32_t`)
`Data`	1020 bytes (maximum)	Contains the data. The length of the buffer is determined by the `data_size` parameter. The format of the data is not defined, but it is suggested that the data be formatted as plain-text characters so it can be understood without requiring a special formatting program. The agent cannot summarize the data over an interval, it must be treated as trace data. One suggestion is to format all information as plain-text characters so it can be read by a person without a special formatting program. Note that because the data in an opaque buffer cannot be summarized, and processing by the agent may consist of logging the data to a trace file, many calls at a high frequency could result in a loss of data or a slowing down of the system, most likely due to an excessive amount of file I/O. Therefore it is recommended that the call be used only in special situations. NULL termination is *not* required.

Three Ways to Instrument within a Transaction Instance

There are three methodologies for instrumenting within a transaction instance. The first two are useful when the transaction is within one application; the last one is useful when the transaction is distributed across applications or systems.

1 Instrument a transaction using `arm_update` as a "heartbeat", when it is an operation that takes a long time to complete (several minutes or hours) and you want to show the overall progress of the transaction in numeric form.

   If these transactions have different steps associated with processing each record, you may want to instrument these steps with component transactions (as described below), or use repeated calls to `arm_update` to

show the overall progress of the transaction. For example, the transaction may process a million records. A call to `arm_update` could be made for every 1000 records or every minute of processing. This could show the progress of the transaction based on the number of times `arm_update` was called or with one or more application-defined metrics.

2 Instrument a transaction using component transactions when it is a long transaction that has many steps. A transaction can be defined for the overall transaction and then nested transactions can be defined for each of the steps. A step might represent a single discrete operation, or it could represent a large number of operations, such as copying 1000 files. This allows for the monitoring of each of the steps as well as the overall transaction.

   For example, step 1 takes about 20 minutes, step 2 takes about 40 minutes, and step 3 takes about 10 minutes. Each step can have a defined transaction as well as the overall transaction. So you would define 3 component transactions monitoring each step, plus one transaction that monitors the overall transaction.

3 Instrument using transaction correlation when the transaction has components that span several applications or systems. This approach is more complex than the previous two as it requires changes to all the applications involved in processing components of the transaction, but it is the most accurate way to track transaction response time spanning systems.

Internationalization

The ARM API is designed to enable applications to use native code pages and languages, and for measurement agents to be able to support many different languages. Users of agents should contact the providers to see if the agent supports the needed code pages and languages.

The ARM API supports any code page as long as no characters are encoded with binary zero bytes (octets). This is because most strings are passed as NULL terminated strings, and the NULL terminator character is a binary zero byte. If a binary zero byte is encountered before the end of the string, the agent would interpret the zero byte as the NULL terminator and truncate the string. Most code pages meet this requirement.

These are code pages that contain binary zero bytes, but there are alternate ways to encode the characters. A well-known example is the Unicode standard. In its native format using 16 bit characters (UTC-2), there are binary zero bytes. However, the UTF-8 encoding of the same Unicode characters does not contain binary zero bytes, and this format is entirely compatible with the ARM API.

Agents that support native languages will often use the following technique. When the application links to the agent it links to a part of the agent that executes in the same process space as the application. Typically this small part of the agent communicates with the main part of the agent across an inter-process communications (IPC) channel. The small part of the agent that executes in the same process as the application can issue an operating system call to find out what code page and language the process is using. It can then pass this information to the main part of the agent, and the main part of the agent can convert from the native code page as necessary.

There are the following three restrictions on the use of native languages.

1. The strings can contain no binary zero bytes except for the NULL terminator character (as was mentioned above).
2. All the strings should be encoded using the same code page and language information as the process that executes the `arm_init` call. This also implies that the code page and language information should not change after the `arm_init` call.

3 This technique does not apply to any string data passed within the optional buffers on `arm_start`, `arm_update`, and `arm_stop`. This is because these strings are not null terminated. Note that it *does* apply to the metric descriptions passed within the optional buffer on `arm_getid`. Further, these strings are often about things that are external to the program, such as a part number or an error code, so the requirement to use the same code page and language information as the process is unacceptable. The application developer is strongly recommended to restrict these strings to the first 128 bytes of the standard Latin code pages for ASCII and EBCDIC (depending on the platform).

A Measurement Agent Information

Introduction

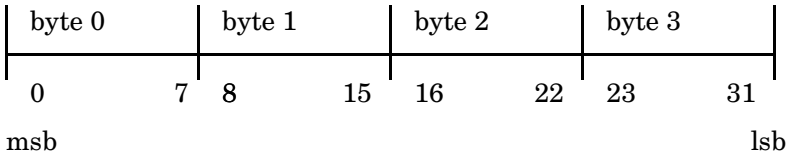
This appendix contains information provided for measurement agent implementers as opposed to ARM application instrumenters. For instrumenters it is provided as reference only, the correlator is "opaque" from an application instrumenter's perspective.

The agents provide the correlators, and within the correlator they provide information to uniquely identify agents. To enable an enterprise management solution (correlation application) to analyze the correlators coming from different systems in a heterogeneous environment, agents need to follow some conventions when creating correlators.

The following section documents a set of semantics for measurement agents to use in formatting the correlator and agent identifiers.

The correlator passed on `arm_start` calls is sent across systems, so it is always in network byte order. Network byte order is a standard described as follows:

Figure 4 Buffer word/byte/bit Format



Format of the Correlator

Correlators provided by agents and passed on the `arm_start` commands have the following format.

Table 5

2 bytes	`Length of the Correlator` (unsigned16) If this value is zero, it means that the measurement agent is not returning a correlator, and therefore there is not any reason to pass this correlator on to other parts of the application (or servers that it calls). A zero length provides another safeguard for agents. If an application passes a null correlator anyway, when any agent receives this correlator as the parent correlator for another transaction, the agent can see that the data in the correlator is invalid and ignore it, regardless of whether the "parent correlator" bit (Flags First Byte `a`) is set in the `arm_start` buffer.
1 byte	`Correlator` format (unsigned8)=1 Only one format is defined at this point, but others could be added in the future.
1 byte	`Flags` First Byte (bit8) `ab000000`, where `a` and `b` are bit flags: `a = 1` if a trace of this transaction and any nested component transactions is requested by the agent. `b = 1` if a trace of this transaction and any nested component transactions is requested by the application. The application requests this by setting the `d` bit (in `abcdefgh` notation) in the first flag byte in the buffer passed on `arm_start`. The agent will decide whether to set this bit, based on its capabilities and how it is configured.

Table 5

	The "trace this correlator" flag is a way to cause agents to trace and/or monitor a transaction and all component transactions associated with the transaction without having to trace or monitor *all* transactions on a system, or without requiring a complicated infrastructure to control tracing and monitoring. Note that this does *not* preclude other ways to control agents, nor is this intended to be a final and comprehensive solution. It is intended that this will be used in addition to other approaches. When an agent builds a correlator, it is free to turn on these flags. The agent might do this if an application has been experiencing unsatisfactory response times. Any agents that receive this correlator as the parent correlator for a component transaction will also see the flag, and they in turn could turn on the flag in any correlators they generate. This process could repeat, resulting in the passing of the trace flag through all the transactions of interest. All the agents might be configured to trace only the few transactions with this flag on, and this would both capture the information needed to diagnose the transaction problem, and avoid overloading the agents and their systems with attempts to trace all transactions. The reason there are separate flags for traces requested by an agent and an application is to provide additional flexibility in how policies for monitoring and tracing are implemented. It might be common for an installation to trace transactions only when requested by agents (based on how the administrator has configured the agents), because then the administrator would control all tracing. On the other hand, permitting the application to highlight when a transaction is special has advantages.

Table 5

2 bytes	`Format of the Address field` (unsigned16) The following formats are defined: 0 = reserved 1 = IPv4 2 = IPv4+port number 3 = IPv6 4 = IPv6+port number 5 = SNA 6 = X.25 7:32767 = reserved This list will be expanded as new requirements arise. The intent is to provide a value for any common addressing format as soon as the need is identified. 32768-65535 = undefined and available for agent implementers to use. There are no semantics associated with the address format. It will be an unusual situation where a new format is needed, but this provides a solution if this is needed. The preferred approach is to get a new format defined that is in the 0-32767 range. There is a risk that two different agent developers will choose the same ID, but this risk is small.
2 bytes	`Vendor ID` (unsigned16) The vendor ID is a way to identify who built the agent. Combining this information with the `Agent Version` field will provide a way for a management application to know what kind of agent generated a correlator. A management application may contain specialized functions or logic that only works with the agents from a particular vendor and/or supporting particular functions or interfaces. By putting these two fields in the correlator, a management application has a way to know whether the agent that generated the correlator has some of these specialized capabilities. For example:

Table 5

	The management application wants to contact the agent to know the name of the application, user, and transaction class running this transaction instance. Although the address of the agent is known from the `Address` field, the protocol that one uses to interface to the agent could be anything. The management application may know how to access several different agents, and could use these values to determine if the correlator came from an agent that it knows how to access. Alternately an agent has a special capability. For example, maybe version 3.3 of a vendor's agent analyzes data in a particular way, but previous versions do not. The management application could use this field to see what are the agent's capabilities. In order to minimize the possibility of two vendors using the same vendor ID, the value should be taken from the list of enterprise identifiers from the Internet Assigned Numbers Authority (IANA). This list was created for vendors who have SNMP agents. Although the ARM API specification does not require or endorse SNMP, it is likely that most or all the organizations that will create an ARM agent will have at least one enterprise ID assigned. The list of enterprise IDs can be found at: `ftp://ftp.isi.edu/in-notes/iana/assignments \ /enterprise-numbers` For organizations that do not have an enterprise identifier assigned by the IANA, the values between 32768-65535 are free for agent developers to use. There are no semantics associated with these IDs. It is expected that most or all agent developers will have a formally assigned `vendor ID`, and it will be an unusual situation where another ID is needed, but this provides a solution if this is needed. There is a risk that two different agent developers will choose the same ID, but this risk is very small.

Table 5

2 bytes	`Agent Version` (unsigned16) The `Agent Version` is used to distinguish between different versions of an agent, and will be most useful when the capabilities and/or interfaces of an agent change from one release to another. It will also be useful to distinguish between different agents from the same vendor. Each vendor is responsible for avoiding having multiple agents with different capabilities using the same `Agent Version` value. Refer to the explanation in the `Vendor ID` field above to understand how to use this field.
2 bytes	`Agent Instance` (unsigned16) Each agent assigns transaction IDs and start handles. Typically there will be one agent on each system, and this one agent is responsible for making sure that there are not any duplicate IDs or handles. From one system to another, however, duplicate IDs and handles will be common, i.e., an ID/handle combination assigned on system X will also be assigned on system Y. One of the main purposes of the `Address`, `Vendor ID`, and `Agent Version` fields is to tell a management application how to contact an agent in order to translate the transaction ID and start handle into the names of the application, user, and transaction class, and the instance of the transaction. As long as there is only one set of IDs and handles stored at that address, all the required information is there. However, if the address is not the address of an individual agent, but rather is the address of a directory that contains information about multiple agents, there is not sufficient information, because the ID/handle combinations can be duplicated. The purpose of the `Agent Instance` field is to provide a way to identify which agent generated a correlator, even if the correlation data from multiple agents is available at the address specified in the `Address` field.
4 bytes	Transaction instance (`start_handle` returned from an `arm_start`)

Table 5

4 bytes	Transaction class ID (`tran_id` returned from an `arm_getid`)

Table 5

2 bytes	`Length of the Address field` (unsigned16)
Maximum 146 bytes	`Address` This field is the address of the agent. More precisely, it is the address that a management application can contact in order to have the Transaction class ID mapped to the names of an application, user, and transaction class, and to get information about the transaction instance, or aggregated data about the transaction class (or any other data). The maximum length of this field is determined by an overall limit of 168 bytes for the correlator. In the correlator format described here, the maximum address length is 146 bytes. In actual practice, it is expected to be no more than 20 bytes for most implementations. If new correlator formats are added in the future, the maximum size of this field could change. The maximum correlator size of 168 bytes will not change. Correlators are passed on `arm_start` calls as part of the buffer pointed to by the `data` pointer. The maximum size of the buffer is 256 bytes, of which 88 bytes are used for other fields, leaving 168 bytes for the correlator. An application should allocate space for the full 256 bytes when making the `arm_start` call, but can then use the `Correlator Length` field to determine how long the correlator really is, and only forward that much data to other cooperating applications. Following are the formats that have been defined so far. The data is stored in network standard byte order, in which integers are sent most significant byte first, unless otherwise indicated. This list is not intended to be exhaustive, and will be extended whenever a new agent implementation requires a new format.

Table 5

	0 = reserved 1 = IPv4 Bytes 0:3 4 byte IP address 2 = IPV4+port number Bytes 0:3 4 byte IP address Bytes 4:5 2 byte IP port number 3 = IPv6 Bytes 0:15 16 byte IP address 4 = IPv6+port number Bytes 0:15 16 byte IP address Bytes 16:17 2 byte IP port number 5 = SNA Bytes 0:7 EBCDIC-encoded network ID Bytes 8:15 EBCDIC-encoded network accessible unit (control point or LU) 6 = X.25 Bytes 0:15 The X.25 network address (also referred to as an X.121 address). This is up to 16 ASCII character digits ranging from 0-9. The length is known from the `Length of the Address` field. An agent running over an X.25 link with the IP configured may choose to use this format or the IP format. This format must be used when IP is not configured above an X.25 link. 7:32767=reserved 32768-65535=undefined and available for agent implementers to use

B Examples

Introduction

These examples are shown for their simplicity. There are more elegant ways to program the same tasks, but the examples demonstrate the ARM API function calls. These sample programs and sample programs for languages other than C are also available on the ARM API CD-ROM and the ARM Web Site mentioned earlier in this book under the section Additional Information on page 13, in Chapter 1.

arm.h Header File

```
#ifndef ARM_H_INCLUDED
#define ARM_H_INCLUDED


/*****************************************************************************/
/* arm.h - ARM API Definitions                                               */
/*****************************************************************************/


#include <sys/types.h>      /* C types  definitions */

/*  Type definitions for various field sizes */

/*  64-bit integer compiler support                                          */
/*                                                                           */
/* If a type declaration supporting 64 bit integer arithmatic is defined     */
/* for the target platform and compiler, the "INT64" #define should be       */
/* set to that type declaration. E.g.,                                       */
/*                                                                           */
/*         #define INT64 long long                                           */
/*                                                                           */
/* If 64 bit arithmatic is not supported on the target platform or compiler,*/
/* remove (or comment out) the "INT64" #define and structures of two 32 bit */
/* values will be defined for the 64 bit fields.                             */


/*
#define INT64 long long
*/


typedef unsigned char  bit8 ;
typedef short int16 ;
typedef long  arm_int32_t ;
typedef unsigned char  unsigned8 ;
typedef unsigned short unsigned16 ;
typedef unsigned long unsigned32 ;

#ifdef INT64
typedef INT64 int64 ;
typedef unsigned INT64 unsigned64 ;
#else
typedef struct int64 {
    int3   upper;
    arm_int32_t  lower;
} int64 ;
```

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

/***    Symbol definitions    ***/

/*  Enumeration of transaction status completion codes  */

enum arm_tran_status_e { ARM_GOOD = 0, ARM_ABORT, ARM_FAILED };

/*  Enumeration of user data formats  */

enum arm_userdata_e { ARM_Format1 = 1, ARM_Format2, ARM_Format101 = 101 };

/*  Enumeration of metric types  */

typedef enum arm_metric_type_e {
     ARM_Counter32 = 1, ARM_Counter64, ARM_CntrDivr32,
     ARM_Gauge32, ARM_Gauge64, ARM_GaugeDivr32, ARM_NumericID32,
     ARM_NumericID64, ARM_String8, ARM_String32,
     ARM_MetricTypeLast
} arm_metric_type_e;

/***   Data definitions    ***/

/*  User metric structures */

typedef struct arm_cntrdivr32_t {        /*  Counter32 + Divisor32 */
    unsigned32  count;
    unsigned32  divisor;
} arm_cntrdivr32_t;

typedef struct arm_gaugedivr32_t {       /*  Gauge32 + Divisor32  */
    arm_int32_t       gauge;
    unsigned32  divisor;
} arm_gaugedivr32_t;

/*  Union of user ARM_Format1 metric types */

typedef union arm_user_metric1_u {
    unsigned32  counter32;              /* Counter32  */
    unsigned64    counter64;            /* Counter64             */
    arm_cntrdivr32_t cntrdivr32;        /* Counter32 + Divisor32 */
    arm_int32_t         gauge32;              /* Gauge32               */
    int64         gauge64;              /* Gauge64               */
    arm_gaugedivr32_t gaugedivr32;      /* Gauge32 + Divisor32   */
    unsigned32    numericid32;          /* NumericID32           */
```

```
    unsigned64    numeric64;            /* NumericID64           */
    char          string8[8];           /* String8               */
} arm_user_metric1_u;

/*  Application view of correlator */

typedef struct arm_app_correlator_t {
    int16         length;               /* Length of the correlator   */
    char agent_data[166];               /* Agent specific data fields */
} arm_app_correlator_t;

/*  User metrics ARM_Format1 structure definition */

typedef struct arm_user_data1_t {
   arm_int32_t        format;             /* Version/format id (userdata_e)
*/
    bit8          flags[4];             /* Flags for metrics' presence    */
    arm_user_metric1_u metric[6];       /* User metrics */
    char          string32[32];         /* 32 byte non-terminated string  */
    arm_app_correlator_t correlator;    /* Correlator   */
} arm_user_data1_t;

/*  User metrics ARM_Format2 structure definition */

typedef struct arm_user_data2_t {
   arm_int32_t        format;             /* Version/format id (userdata_e)
*/
    char          string1020[1020];     /* 1020 byte opaque blob  */
} arm_user_data2_t;

/*  User metric meta-data for ARM_Format101 structure */

typedef struct arm_user_meta101_t {
    arm_int32_t        type;                /* Type of metric
(arm_user_metric_e)*/
    char          name[44];            /* NULL-terminated string <= 44 char */
} arm_user_meta101_t;

/*  User meta-data ARM_Format101 structure definition */

typedef struct arm_user_data101_t {
   arm_int32_t        format;             /* Version/format id (userdata_e)
*/
    bit8          flags[4];            /* Flags for which fields are present*/
    arm_user_meta101_t meta[7];         /* User metrics meta-data  */
} arm_user_data101_t;

/*  Flag bit definitions (within bit8 fields) */

/*  flags[0] in arm_user_data1_t passed in arm_start */
```

```
#define ARM_CorrPar_f      0x80          /* Correlator from parent  */
#define ARM_CorrReq_f      0x40          /* Request correlator generation */
#define ARM_CorrGen_f      0x20         /* New correlator generated in data*/
#define ARM_TraceReq_f     0x10          /* User trace request  */

/*  flags[1] in arm_user_data101_t passed in arm_get_id and */
/*  flags[1] in arm_user_data1_t passed in arm_start, arm_update and arm_end
*/
#define ARM_Metric1_f      0x80          /*  Metric 1 present  */
#define ARM_Metric2_f      0x40          /*  Metric 2 present  */
#define ARM_Metric3_f      0x20          /*  Metric 3 present  */
#define ARM_Metric4_f      0x10          /*  Metric 4 present  */
#define ARM_Metric5_f      0x08          /*  Metric 5 present  */
#define ARM_Metric6_f      0x04          /*  Metric 6 present  */
#define ARM_AllMetrics_f   0xfc          /*  Metrics 1 - 6 present  */
#define ARM_String1_f      0x02          /*  String 1 present  */

#if defined  _WIN32
 #include <windows.h>
 #define ARM_API WINAPI
#elif defined __OS2__
 #define ARM_API _Pascal
#elif defined _OS216
 #define  arm_data_t char _far
 #define  arm_ptr_t  char _far
 #define ARM_API _far _pascal
#elif defined _WIN16 || _WINDOWS
 #include <windows.h>
 typedef BOOL (FAR PASCAL _export * FPSTRCB) (LPSTR, LPVOID);
 #define  arm_data_t char FAR
 #define  arm_ptr_t  char FAR
 #define ARM_API WINAPI
#else  /* unix */
#define ARM_API
#endif

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#ifdef _PROTOTYPES

/***    Function prototypes    ***/

extern arm_int32_t ARM_API arm_init(
    char*      appl_name,               /* application name             */
    char*      appl_user_id,            /* Name of the application user */
    arm_int32_t      flags,                 /* Reserved = 0                 */
    char*      data,                    /* Reserved = NULL              */
    arm_int32_t      data_size);            /* Reserved = 0                 */
```

```
extern arm_int32_t ARM_API arm_getid(
    arm_int32_t      appl_id,                /* application handle          */
    char*      tran_name,                /* transaction name            */
    char*      tran_detail ,             /* transaction additional info  */
    arm_int32_t      flags,                  /* Reserved = 0                */
    char*      data,                   /* format definition of user metrics */
    arm_int32_t      data_size);             /* length of data buffer       */

extern arm_int32_t ARM_API arm_start(
    arm_int32_t      tran_id,               /* transaction name identifier  */
    arm_int32_t      flags,                  /* Reserved = 0                */
    char*      data,                   /* user metrics data           */
    arm_int32_t      data_size);             /* length of data buffer       */

extern arm_int32_t ARM_API arm_update(
    arm_int32_t      start_handle,           /* unique transaction handle    */
    arm_int32_t      flags,                  /* Reserved = 0                */
    char*      data,                   /* user metrics data           */
    arm_int32_t      data_size);             /* length of data buffer       */

extern arm_int32_t ARM_API arm_stop(
    arm_int32_t      start_handle,           /* unique transaction handle    */
    arm_int32_t      tran_status,            /* Good=0, Abort=1, Failed=2    */
    arm_int32_t      flags,                  /* Reserved = 0                */
    char*      data,                   /* user metrics data           */
    arm_int32_t      data_size);             /* length of data buffer       */

extern arm_int32_t ARM_API arm_end(
    arm_int32_t      appl_id,                /* application id              */
    arm_int32_t      flags,                  /* Reserved = 0                */
    char*      data,                   /* Reserved = NULL             */
    arm_int32_t      data_size);             /* Reserved = 0                */

#else  /* _PROTOTYPES */

extern arm_int32_t     ARM_API arm_init();
extern arm_int32_t     ARM_API arm_getid();
extern arm_int32_t     ARM_API arm_start();
extern arm_int32_t     ARM_API arm_update();
extern arm_int32_t     ARM_API arm_stop();
extern arm_int32_t     ARM_API arm_end();

#endif /* _PROTOTYPES */

#ifdef __cplusplus
}
#endif /* __cplusplus */

/*  Type definitions for compatibility with version 1.0 of the ARM API */
```

```
typedef arm_int32_t        arm_appl_id_t;
typedef arm_int32_t        arm_tran_id_t;
typedef arm_int32_t        arm_start_handle_t;
typedef unsigned32   arm_flag_t;
typedef char         arm_data_t;
typedef arm_int32_t        arm_data_sz_t;
typedef char         arm_ptr_t;
typedef arm_int32_t        arm_ret_stat_t;
typedef arm_int32_t        arm_status_t;
#endif /* ARM_H_INCLUDED */
```

C/C++ (all platforms) Sample 1

Sample 1 uses standard ARM API calls, *not* advanced functions.

```
/*********************************************************************/
/* sample1.c                                                         */
/*                                                                   */
/* This program provides examples of how to use the features provided by */
/* version 1.0 and 2.0 of the ARM API.                               */
/*                                                                   */
/*********************************************************************/


#include <stdio.h>
#include "arm.h"

arm_int32_t appl_id = -1;          /* Define an indentifer for the application
id */

arm_int32_t simple_tran_id = -1;    /* Define a unique identifier for each
*/
arm_int32_t long_tran_id_1 = -1;    /* TRANSACTION                    */
arm_int32_t long_tran_id_2 = -1;
arm_int32_t  sub_tran_id_1 = -1;
arm_int32_t  sub_tran_id_2 = -1;


/*********************************************************************/
/* init                                                              */
/*********************************************************************/


void init()
{
   appl_id=arm_init("ARM sample program",          /* application name */
                    "*",                           /* use default user */
                    0,0,0);
   simple_tran_id = arm_getid(appl_id,
                              "Simple_transaction_1",/* transaction name */
                              "First Transaction in Sample program",
                              0,0,0);

   if (simple_tran_id < 0)
           printf("Simple_transaction_1 is not registered.\n");

   long_tran_id_1 = arm_getid(appl_id,
                              "Long_transaction_1", /* transaction name  */
```

```
                              "A long transaction using arm_update",
                              0,0,0);

   if (long_tran_id_1 < 0)
        printf("Long_transaction_1 is not registered.\n");

   long_tran_id_2 = arm_getid(appl_id,
                              "Long_transaction_2", /* transaction name */
                              "A long transaction using sub transactions",
                              0,0,0);

   if (long_tran_id_2 < 0)
        printf("Long_transaction_2 is not registered.\n");

   sub_tran_id_1 = arm_getid(appl_id,
                            "Sub_tran1_of_long_tran_2", /* transaction name */
                             "Subtransaction 1 of Long_trans2",
                             0,0,0);

   if (sub_tran_id_1 < 0)
        printf("Sub_tran_of_long_tran_2 is not registered.\n");

   sub_tran_id_2 = arm_getid(appl_id,
                            "Sub_tran2_of_long_tran_2", /* transaction name */
                             "Subtransaction 2 of Long_trans2",
                             0,0,0);

   if (sub_tran_id_2 < 0)
        printf("Sub_tran_of_long_tran_2 is not registered.\n");

} /* init */


/*************************************************************************/
/*simple_trans                                                           */
/*************************************************************************/


void simple_trans1()
{
   arm_int32_t  tran_handle;

   tran_handle = arm_start(simple_tran_id, /* transaction id from arm_getid */
                           0,0,0);

   /*****************************************/
   /* Perform actual transaction processing here */
   /*****************************************/
```

```
   arm_stop(tran_handle,         /* transaction handle from arm_start  */
            ARM_GOOD,            /* successful completion define = 0   */
            0,0,0);

   return;
}  /* simple_trans1 */


/*********************************************************************/
/* long_trans_using_update                                           */
/*                                                                   */
/* arm_update can show the progress of an iterative process          */
/*********************************************************************/


void long_trans_using_update()
{

#define MAX_COUNT  1000000
#define UPDATE_COUNT 100000     /* call update every 100,000 iterations */

   arm_int32_t  tran_handle;
   int  i;

   tran_handle = arm_start(long_tran_id_1, /* transaction id from arm_getid */
                           0,0,0);

   for (i=1;i<=MAX_COUNT;i++)
   {
       /* your processing goes here */

       if (i%UPDATE_COUNT == 0)
          arm_update(tran_handle,  /* update based on UPDATE_COUNT */
                     0,0,0);

   }

   arm_stop(tran_handle,           /* transaction handle from arm_start */
            ARM_GOOD,              /* successful completion define = 0  */
            0,0,0);

   return;

}  /* long_trans_using_update */


/*********************************************************************/
/* long_trans_using_sub_trans                                        */
/*                                                                   */
/* Sub-transactions can show the progress of the steps               */
```

```
/* of a long transaction.                                                  */
/*************************************************************************/


void long_trans_using_sub_trans()
{
   arm_int32_t  tran_handle;
   arm_int32_t  sub_tran_handle1;
   arm_int32_t  sub_tran_handle2;

   /* record the overall transaction processing (optional) */
   tran_handle = arm_start(long_tran_id_2, /* transaction id from arm_getid */
                           0,0,0);

   /* start recording the first step of the long transaction */
   sub_tran_handle1 = arm_start(sub_tran_id_1,
                                0,0,0);


   /************************************/
   /* Process step 1 on this transaction */
   /************************************/


   /* record the completion of the first step */
   arm_stop(sub_tran_handle1,    /*transaction handle from arm_start */
            ARM_GOOD,            /*successful completion define= 0 */
            0,0,0);

   /*start recording the second step of the long transaction */
   sub_tran_handle2 = arm_start(sub_tran_id_2,
                                0,0,0);

   /************************************/
   /* Process step 2 on this transaction */
   /************************************/

   /* record the completion of the second step */
   arm_stop(sub_tran_handle2,   /* transaction handle from arm_start */
            ARM_GOOD,           /* successful completion define = 0 */
            0,0,0);

   /* record the completion of the overall transaction */
   arm_stop(tran_handle,        /* transaction handle from arm_start */
            ARM_GOOD,           /* successful completion define = 0 */
            0,0,0);

   return;
}  /* long_trans_using_sub_trans */
```

```
/************************************************************************/
/* main                                                                 */
/************************************************************************/


main()
{

   int continue_processing = 1;

   init();

   while (continue_processing)
   {
       simple_trans1();
       long_trans_using_update();
       long_trans_using_sub_trans();

        continue_processing = 0;
}
   arm_end(appl_id,         /* application id from arm_init */
            0,0,0);

   return(0);
}
```

C/C++ (all platforms) Sample 2

Sample 2 uses the advanced functions of application-defined metrics and transaction correlation.

```
/*********************************************************************/
/* Sample2.c                                                         */
/*                                                                   */
/* This program provides examples of how to use two of the new features */
/* provided by version 2.0 of the ARM API, user defined  metrics and    */
/* correlation. For simplicity, this sample program does not perform    */
/* any  error checking.                                              */
/*********************************************************************/


#include <stdio.h>
#include "arm.h"


arm_int32_t    client_appl_id = -1;   /* application id */
arm_int32_t    client_tran_id = -1;   /* transaction id */


arm_int32_t    metric_appl_id = -1;   /* application id */
arm_int32_t    metric_tran_id = -1;   /* transaction id */


/*********************************************************************/
/* server_application                                                */
/*                                                                   */
/* This routine is included here to simplify this example. In a real     */
/* life situation, this piece of code would likely be running on a       */
/* separate system.                                                  */
/*********************************************************************/


void server_application(arm_app_correlator_t client_correlator)

{

    arm_int32_t   server_appl_id = -1;     /* unique application id  */
    arm_int32_t   server_tran_id = -1;     /* unique transacation id */
    arm_int32_t   server_tran_handle = -1; /* transaction instance   */

    arm_user_data1_t     *buf_ptr, buf = {
        1,                              /* header */
        {ARM_CorrPar_f, 0, 0, 0},       /* flags  */
```

```
        };

    arm_int32_t   buf_sz;

    int     i, data_len;


    server_appl_id = arm_init("Server_Application",  /* application name */
                              "*",                   /* use default user */
                              0,0,0);                /* reserved */

    server_tran_id = arm_getid(server_appl_id,       /* appl_id from arm_init
*/
                              "Server_transaction",  /* transaction name */
                              "First Transaction in Server program",
                              0,                     /* data buffer */
                             0,0);                 /* buffer pointer & size */

    /* Pass the parent correlator received from the client application to  */
    /* the ARM agent using the arm_start call.                            */

   buf_ptr = &buf;
   buf_ptr->flags[0] = ARM_CorrPar_f;

   buf_ptr->correlator.length = client_correlator.length;
   data_len = (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];

   buf_sz = (sizeof(buf)-sizeof(client_correlator) +
client_correlator.length);

   server_tran_handle = arm_start(server_tran_id,  /* tran_id from arm_getid
*/
                                  0,               /* reserved */
                                  (char *)buf_ptr,
                                  buf_sz);


   /*********************************************/
   /* Perform actual transaction processing here */
   /*********************************************/


   arm_stop(server_tran_handle,    /* transaction handle from arm_start */
            ARM_GOOD,              /* successful completion define = 0 */
            0,                     /* reserved for future use */
            0,0);                  /* buffer pointer & buffer size */

    arm_end(server_appl_id,        /* application id from arm_init */
```

```
            0,0,0);                 /* reserved for future use */

    return;

}    /* server_application() */


/*********************************************************************/
/*  client_transaction                                               */
/*********************************************************************/


void client_transaction()

{

    arm_int32_t  client_tran_handle = -1;    /* transaction start handle  */

    arm_user_data1_t    *buf_ptr, buf = {
        1,                              /* Header */
        };

    arm_int32_t   buf_sz;

    arm_app_correlator_t   correlator = {
        0,       /* correlator length */
        0,       /* agent data */
        };

    int   i, data_len;

    buf_ptr = &buf;
    buf_sz = sizeof(buf);

    /* The client appliation requests a correlator from the ARM Agent */

    buf_ptr->flags[0] = ARM_CorrReq_f;
    client_tran_handle = arm_start(client_tran_id, /* tran_id from arm_getid
*/
                                   0,             /* reserved for future use*/
                                   (char *)buf_ptr,/* metrics buf ptr       */
                                   buf_sz);       /* user metric buffer size*/

    /* If the ARM Agent returns a correlator, determine the size of the    */
    /* agent specific data in the correlator and pass the data, along with */
    /* the correlator length, to the server application.                   */

    if ((buf_ptr->flags[0] & ARM_CorrGen_f) == ARM_CorrGen_f) {
        correlator.length = buf_ptr->correlator.length;
        data_len = (correlator.length - sizeof(buf_ptr->correlator.length));
```

```
        for (i = 0; i < data_len; i++)
             correlator.agent_data[i] = buf_ptr->correlator.agent_data[i];
}

server_application(correlator);

    arm_stop(client_tran_handle,    /* transaction handle from arm_start */
             ARM_GOOD,              /* successful completion define = 0 */
             0,                     /* reserved for future use */
             0,0);                  /* buffer pointer & buffer size */

     return;

}    /* client_transaction() */


/*********************************************************************/
/*  init_client_application                                          */
/*********************************************************************/


void init_client_application()

{

    client_appl_id = arm_init("Client_Application", /* application name */
                              "*",                  /* use default user */
                            0,0,0);               /*reserved for future user*/
    client_tran_id = arm_getid(client_appl_id,      /* appl_id from arm_init
*/
                              "Client_transaction", /* transaction name */
                              "First transaction in Client application",
                              0,                    /* reserved */
                             0,0);                 /* buffer pointer & size */

    return;

} /* init_client_application */


/*********************************************************************/
/*  metric_transaction                                               */
/*********************************************************************/


void metric_transaction()

{
    arm_int32_t metric_tran_handle = -1;        /*transaction start handle */
```

```
    arm_user_data1_t *buf_ptr, buf = {
        1,                                                  /* Header */
        {0, ARM_AllMetrics_f | ARM_String1_f, 0, 0},    /* Flags  */
        };

    arm_int32_t   buf_sz;

    buf_ptr = &buf;
    buf_sz = sizeof(buf);

    buf_ptr->metric[0].counter32 = 0x32;
    buf_ptr->metric[1].gauge32 = 0x32;
    buf_ptr->metric[2].counter64.upper = 0x01234567;
    buf_ptr->metric[2].counter64.lower = 0x76543210;
    strcpy(buf_ptr->metric[3].string8, "String 8");
    buf_ptr->metric[4].cntrdivr32.count = 0x32;
    buf_ptr->metric[4].cntrdivr32.divisor = 0x32;
    buf_ptr->metric[5].numericid64.upper = 0x01234567;
    buf_ptr->metric[5].numericid64.lower = 0x76543210;
    strcpy(buf_ptr->string32,"This is a 32 character string ");

    metric_tran_handle = arm_start(metric_tran_id,  /*tran_id from arm_getid
*/
                                   0,               /* reserved */
                                   (char *)buf_ptr, /* metrics buf ptr*/
                                  buf_sz);         /*user metric buffer size*/

    /*********************************/
    /* Perform some processing here */
    /*********************************/


    arm_update(metric_tran_handle,   /* transaction handle from arm_start */
               0,                    /* reserved for future use */
               (char *)buf_ptr,      /* user metrics buffer pointer */
               buf_sz);              /* user metric buffer size */

    /************************************/
    /* Perform some more processing here */
    /************************************/


  arm_stop(metric_tran_handle,  /* transaction handle from arm_start */
           ARM_GOOD,            /* successful completion define = 0 */
           0,                   /* reserved for future use */
           (char *)buf_ptr,     /* user metrics buffer pointer */
           buf_sz);             /* user metric buffer size */

  return;
```

```
}    /* metric_transaction() */


/*************************************************************************/
/*  init_metric_application                                              */
/*************************************************************************/


void init_metric_application()

{

    arm_user_data101_t *buf_ptr, buf = {
        101,
        {0, ARM_AllMetrics_f | ARM_String1_f, 0, 0},
        {{1,  "Metric #1 - Type 1 is a COUNTER32    "},
        {4,  "Metric #2 - Type 4 is a GAUGE32       "},
        {2,  "Metric #3 - Type 2 is a COUNTER64     "},
        {9,  "Metric #4 - Type 9 is a STRING8       "},
        {3,  "Metric #5 - Type 3 is a COUNTER32/DIVISOR32"}
        {8,  "Metric #6 - Type 8 is a NUMERICID64     "},
        {10, "The last field is always a STRING32  "}
        }};

    arm_int32_t    buf_sz;

    buf_ptr = &buf;
    buf_sz = sizeof(buf);

    metric_appl_id=arm_init("Metric_Application",   /* application name */
                            "*",                    /* use default user */
                            0,0,0);                 /* reserved */

    metric_tran_id = arm_getid(metric_appl_id,      /* appl_id from arm_init
*/
                               "Metric_transaction", /* transaction name */
                               "First transaction in Metric application",
                               0,                   /* reserved */
                               (char *)buf_ptr,     /* buffer */
                               buf_sz);             /* buffer size */

    return;

} /* init_metric_application */


/*******************************************************/
/*  Main                                               */
/*******************************************************/
```

```
main()

{

    int continue_processing = 1;

    init_client_application();

    init_metric_application();

    while (continue_processing)

    {
        client_transaction();
        metric_transaction();
        continue_processing = 0;

    }

    arm_end(client_appl_id,         /* application id from arm_init */
            0,0,0);                 /* reserved for future use */

    arm_end(metric_appl_id,         /* application id from arm_init */
            0,0,0);                 /* reserved  for  future use */

    return(0);
}
```