

HP Performance Agent

For the Windows[®] Operating System

Software Version: 4.70

Application Response Measurement 2.0 API Guide

Document Release Date: September 2007

Software Release Date: September 2007



Legal Notices

Warranty

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

The information contained herein is subject to change without notice.

Restricted Rights Legend

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

Copyright Notices

© Copyright 2007 Hewlett-Packard Development Company, L.P.

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® and Windows® are U.S. registered trademark of Microsoft Corporation.

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

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

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

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

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

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

UNIX® is a registered trademark of The Open Group.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Support

You can visit the HP Software Support web site at:

www.hp.com/go/hpsoftwaresupport

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

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

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

To find more information about access levels, go to:

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

To register for an HP Passport ID, go to:

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

Contents

1	Application Response Measurement API	7
	Application Response Measurement	7
	Measuring Service Levels	7
	ARMing Your Applications	8
	What's New in Version 2.0 of the ARM API	9
2	Basic Tasks for Instrumenting an Application	11
	What to Instrument	12
3	The Software Developer's Kit (SDK)	13
	The Logging Agent	14
	The Header File	14
4	Getting Started	15
	UNIX systems	15
	OS/2, Windows 2000, Windows 2003 or Windows XP systems	15
	Using the Logging Agent	16
	Overview of the ARM API Function Calls	16
	Adding ARM Function Calls to an Application	17
	Definition of Data Type Terminology	19
	Testing Your Instrumentation	19
	Logging Agent Sample Output	20
5	arm_init	21
6	arm_getid	23
7	arm_start	25
8	arm_update	27
9	arm_stop	29
10	arm_end	33
A	Performance Agent Information	35
B	Examples	43
	arm.h Header File	44
	C/C++ (all platforms) Sample 1	48
	C/C++ (all platforms) Sample 2	52

1 Application Response Measurement API

Application Response Measurement

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

Application Response Management

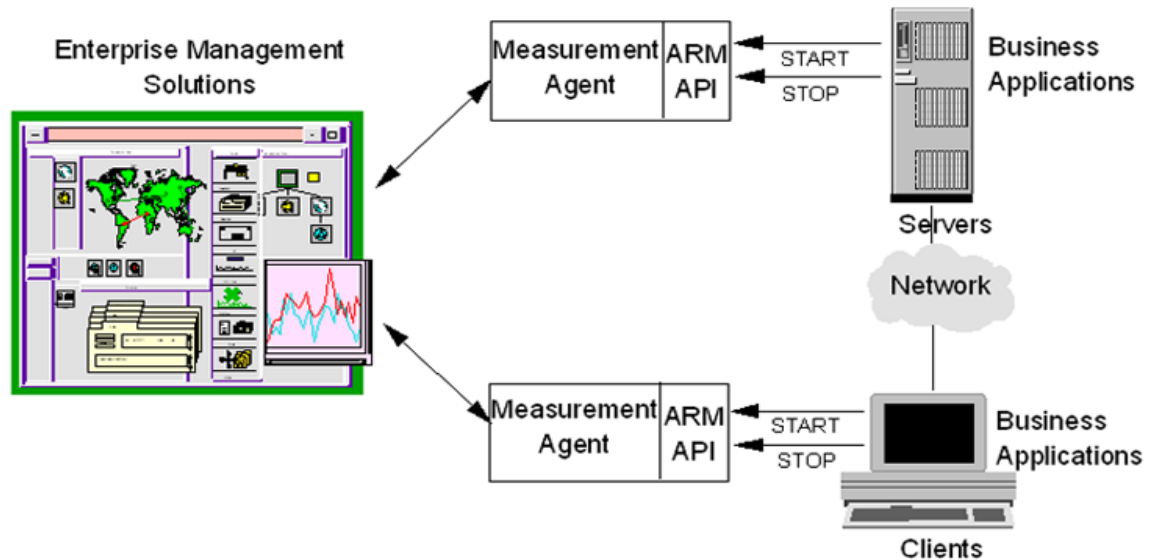


Figure 1 shows how enterprise management applications, measurement agents that implement the ARM API, and business applications that call the ARM API work together to provide a robust way to monitor application response.

What's New in Version 2.0 of the ARM API

Several additions to the ARM API improve the ways your application can be managed.

- 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.
- You can 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 (called application-defined metrics) is useful to better understand response times, and how the application can be tuned to perform better.
- You can use the new logging agent to do simple verification of your instrumentation. It allows you to determine if the correct parameters are being passed on each call, but it does not function as a measurement agent.



Version 2.0 of the ARM API is backward compatible with version 1.0. Applications instrumented to the ARM 1.0 API can continue to function correctly with agents that implement the additional features of the 2.0 API. ARM 2.0 instrumented applications will function correctly with agents that implement the features of the 1.0 API.

2 Basic Tasks for Instrumenting an Application

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

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.

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.

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 “For Your Information”. 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:

- 1 What unit of work does this transaction define?
- 2 Are the transaction counts and/or response times important?
- 3 Who will use this information?
- 4 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)

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 2000, Windows 2003 and Windows XP). 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	<code>x/usr/lib/libarm.sl</code>
IBM AIX	<code>/usr/lib/libarm.a</code>
Sun Solaris	<code>/usr/lib/libarm.so</code>
NCR MP-RAS	<code>/usr/lib/libarm.so</code>
Windows 2000	<code>\$windir\$\SYSTEM32\LIBARM32.DLL</code>
Windows2003	<code>\$windir\$\SYSTEM32\LIBARM32.DLL</code>

Windows XP	\$windir\$\SYSTEM32\LIBARM32.DLL
OS/2 (32-bit)	\$os2dir\$\DLL\LIBARM.DLL
HP-UX 10	x/usr/lib/libarm.sl

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 (see [Using the Logging Agent](#) on page 16 in Chapter 4, for more information on this).

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

This section 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.

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.

OS/2, Windows 2000, Windows 2003 or Windows XP 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`

Windows 2000/Windows 2003/Windows XP:

- `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 19 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.

Measure Agent	Description
<code>arm_init</code>	During the initialization of your application, call <code>arm_init</code> 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 <code>arm_getid</code> .
<code>arm_getid</code>	Use <code>arm_getid</code> 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. <code>arm_getid</code> returns a unique identifier that must be passed to <code>arm_start</code> .

arm_start	Each time a transaction class is executed, this is a transaction instance. 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
arm_update	This is an optional function call that can be made any number of times after arm_start and before arm_stop. arm_update gives information about the transaction instance, such as a “heartbeat” after a group of records has been processed
arm_stop	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.

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.

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.

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.

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 2000, Windows 2003, and Windows XP.

```

/*****
/* sample.c */
/*****
#include <stdio.h>
(1) #include "arm.h"
int32 appl_id = -1; /* Unique identifier for the application */
int32 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()
{
    int32 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 Parameter

ARM API Function Call Parameters

	arm_init	arm_getid	arm_start	arm_update	arm_stop	arm_end
appl_name	X					
appl_user_id (optional) (user name)	X					
tran_name		X				
tran_detail (optional)		X				
tran_status					X	
data and data_size (optional)		X	X	X	X	
Return Codes						
appl_id (appl/user)	○	▲	▲	▲	▲	▲
tran_id		○	▲	▲	▲	
start_handle (transaction)			○	▲	▲	

where ○ indicates the code is returned from one call and passed to another

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:

<code>inc32</code>	A signed 32-bit integer
<code>char *</code>	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:

<code>int64</code>	A signed 64-bit integer
<code>unsigned32</code>	An unsigned 32-bit integer
<code>unsigned64</code>	An unsigned 64-bit integer
<code>bit8</code>	A byte containing 8 single-bit flags. In this document, when a <code>bit8</code> is represented as eight flags using the notation <code>abcdefgh</code> , <code>a</code> is the most significant bit, and <code>h</code> is the least significant bit.
<code>unsigned16</code>	An unsigned 16-bit integer
<code>unsigned8</code>	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 didn't inadvertently alter the program in a way that affects its basic function.
- 3 Compile the logging agent source, `logagent.c`, if you haven't 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>
```

5 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 won't 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

<code>appl_name (char*)</code>	The name used to identify the application. The maximum length is 128 bytes including the NULL string terminator.
<code>appl_user_id (char*)</code>	The name of the application user. On UNIX and Windows 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).
<code>flags (int32)=0</code>	Reserved for future use. It must be set to zero.
<code>data (char*)=0</code>	Reserved for future use. A NULL value (0) must be used.
<code>data_size (int32)=0</code>	Reserved for future use. It must be set to zero.

Return Code

<code>appl_id (int32)</code>	A unique value to reference an application/user identifier. This id must be passed to the <code>arm_getid</code> 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.

6 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 wasn't 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

<code>appl_id (int32)</code>	The unique reference to an application/user identifier returned from the <code>arm_init</code> call. If the <code>appl_id</code> is less than zero, this <code>arm_getid</code> call will be treated as a NULL operation, and a negative <code>tran_id</code> returned.
<code>tran_name (char*)</code>	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 <code>arm_init</code> call). The maximum length is 128 bytes including the NULL string terminator.
<code>tran_detail (char*)</code>	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 <code>appl_id/tran_name</code> pair. If the contents of the field change on later calls using the same <code>appl_id/tran_name</code> pair, the new contents are ignored. The maximum length is 128 bytes including the NULL string terminator. If no <code>tran_detail</code> is associated with this transaction, you must specify the NULL value (0).
<code>flags (int32)=0</code>	Reserved for future use. It must be set to zero.
<code>data (char*)</code>	A pointer to a buffer that describes the format of additional data that can be passed on <code>arm_start</code> , <code>arm_update</code> , and <code>arm_stop</code> 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 <code>arm_getid</code> " on page 46 for the detailed buffer format.
<code>data_size (int32)</code>	The length in bytes of the buffer pointed to by <code>data</code> . If <code>data</code> is set to zero (0), <code>data_size</code> must also be set to zero.

Return Code

<code>tran_id (int32)</code>	The unique identifier assigned for this transaction class. This ID needs to be passed on <code>arm_start</code> 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.

7 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`. See the “Advanced Topics” section for details on how to pass this information.

Syntax

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

Parameters

<code>tran_id (int32)</code>	The unique identifier assigned to the transaction class. This is the id generated by <code>arm_getid</code> . If the <code>tran_id</code> is less than zero, this <code>arm_start</code> call will be treated as a NULL operation, and a negative <code>start_handle</code> returned.
<code>flags (int32)=0</code>	Reserved for future use. It must be set to zero.
<code>data (char*)</code>	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 <code>arm_start</code> , <code>arm_update</code> , and <code>arm_stop</code> " on page 50 for the detailed buffer format.
<code>data_size(int32)</code>	The length in bytes of the buffer pointed to by the <code>data</code> parameter. If <code>data</code> is set to zero (0), <code>data_size</code> must also be set to zero.
<code>tran_id (int32)</code>	The unique identifier assigned to the transaction class. This is the id generated by <code>arm_getid</code> . If the <code>tran_id</code> is less than zero, this <code>arm_start</code> call will be treated as a NULL operation, and a negative <code>start_handle</code> returned.

Return Code

<code>start_handle (int32)</code>	The unique transaction handle assigned to this instance of a transaction. This handle must be passed on <code>arm_stop</code> and any <code>arm_update</code> 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.

8 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 (see the section “Three Ways to Instrument within a Transaction Instance” on page 55).

`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 the section “Format of Data Buffer in `arm_start`, `arm_update`, and `arm_stop`” Format 1, 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 can’t 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

<code>start_handle (int32)</code>	The unique handle from the <code>arm_start</code> call that marked the start of this transaction instance. The <code>start_handle</code> must be passed in each <code>arm_update</code> 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 <code>start_handle</code> is less than zero, this <code>arm_update</code> call will be treated as a NULL operation, and a negative <code>error_status</code> returned.
<code>data (char*)</code>	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).
<code>flags (int32)=0</code>	Reserved for future use. It must be set to zero. There are two possible buffer formats: <ul style="list-style-type: none">— If the <code>Format</code> field contains the value 1, then application-defined metrics as defined in <code>arm_getid</code> can be passed. The <code>correlator</code> field is not used in the <code>arm_update</code> call.— If the <code>Format</code> 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 <code>arm_start</code> , <code>arm_update</code> , and <code>arm_stop</code> " on page 50 for the detailed buffer formats.
<code>data_size (int32)</code>	The length in bytes of the buffer pointed to by <code>data</code> . If <code>data</code> is set to zero (0), <code>data_size</code> should also be set to zero.

Return Code

<code>error_status (int32)</code>	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.

9 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,data,data_size)
```

Parameters

<code>start_handle (int32)</code>	The unique handle from the <code>arm_start</code> call that marked the start of this transaction instance. <code>start_handle</code> , must be passed in each <code>arm_stop</code> 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 <code>start_handle</code> is less than zero, this <code>arm_stop</code> call will be treated as a <code>NULL</code> operation, and a negative <code>error_status</code> returned.
<code>tran_status (int32)</code>	The completion code of the transaction, as determined by the application. 0 = Transaction successful (defined as <code>ARM_GOOD</code> in <code>arm.h</code>). Use this value when the operation completed normally and as expected. 1 = Transaction aborted (defined as <code>ARM_ABORT</code> in <code>arm.h</code>). 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 <code>ARM_FAILED</code> in <code>arm.h</code>). Use this value in applications where the transaction worked properly, but no result was generated. For example, when makingtracking an airline reservation, a server indicates no seats arecall comes in and the travel agent sees available on the requested flight. Since no reservation was made, recordthe transaction wasn't successful; but since the reservation system is operating correctly, it isn't an aborted transaction either. In this case, you might want to record the transaction as a failed transaction.
<code>flags (int32)=0</code>	Reserved for future use. It must be set to zero.
<code>data (char*)</code>	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 <code>arm_start</code> call, except the Correlator field is not used in the <code>arm_stop</code> call. See the section "Format of Data Buffer in <code>arm_start</code> , <code>arm_update</code> , and <code>arm_stop</code> " on page 50 for the detailed buffer format.
<code>data_size (int32)</code>	The length in bytes of the buffer pointed to by the <code>data</code> parameter. If <code>data</code> is set to zero (0), <code>data_size</code> should also be set to zero.

Return Code

<code>error_status (int32)</code>	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 error should be logged so corrective action can be taken.

10 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

<code>appl_id</code> (int32)	A unique reference to an application/user identifier returned from the <code>arm_init</code> call. If <code>appl_id</code> is less than zero, this <code>arm_end</code> call will be treated as a NULL operation, and a negative <code>error_status</code> returned.
<code>flags</code> (int32)=0	Reserved for future use. It must be set to zero.
<code>data</code> (char*)=0	Reserved for future use. A NULL pointer (0) must be used.
<code>data_size</code> (int32)=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.

A Performance Agent Information

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:

Format of the Correlator

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

Command Format	Description
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 isn’t 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 <code>arm_start</code> buffer.
1 byte	Correlator format (unsigned8)=1 Only one format is defined at this point, but others could be added in the future.

Command Format	Description
1 byte	<p>Flags</p> <p>First Byte (bit8)</p> <p>ab000000, where <i>a</i> and <i>b</i> are bit flags:</p> <p><i>a</i> = 1 if a trace of this transaction and any nested component transactions is requested by the agent.</p> <p><i>b</i> = 1 if a trace of this transaction and any nested component transactions is requested by the application. The application requests this by setting the “<i>d</i>” bit (in <i>abcdefgh</i> notation) in the first flag byte in the buffer passed on <code>arm_start</code>. The agent will decide whether to set this bit, based on its capabilities and how it is configured.</p> <p>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 <i>all</i> 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).</p> <p>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.</p> <p>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.</p>

Command Format	Description
2 bytes	<p>Format of the Address field (unsigned16)</p> <p>The following formats are defined:</p> <ul style="list-style-type: none"> 0 = reserved 1 = IPv4 2 = IPV4+port number 3 = IPv6 4 = IPv6+port number 5 = SNA 6 = X.25 7:32767 = reserved <p>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.</p> <p>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.</p>

Command Format	Description
2 bytes	<p data-bbox="602 264 899 296">Vendor ID (unsigned16)</p> <p data-bbox="602 306 1451 590">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:</p> <p data-bbox="602 600 1459 852">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.</p> <p data-bbox="602 863 1459 989">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.</p> <p data-bbox="602 999 1463 1251">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's 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:</p> <p data-bbox="651 1262 1224 1335" style="text-align: center;">ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers</p> <p data-bbox="602 1346 1459 1598">For organizations that don't 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</p>

Command Format	Description
2 bytes	<p>Agent Version (unsigned16)</p> <p>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.</p>
2 bytes	<p>Agent Instance (unsigned16)</p> <p>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 aren't 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.</p> <p>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 isn't sufficient information, because the id/handle combinations can be duplicated.</p> <p>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</p>
4 bytes	Transaction instance (<code>start_handle</code> returned from an <code>arm_start</code>)
4 bytes	Transaction class ID (<code>tran_id</code> returned from an <code>arm_getid</code>)
2 bytes	Length of the address field (unsigned16)

Command Format	Description
Maximum 146 bytes	<p data-bbox="604 266 716 296">Address:</p> <p data-bbox="604 306 1461 491">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).</p> <p data-bbox="604 506 1461 722">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.</p> <p data-bbox="604 737 1461 953">Correlators are passed on <code>arm_start</code> 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 <code>arm_start</code> 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.</p> <p data-bbox="604 968 1461 1115">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.</p>

Command Format	Description
	<p>0 = reserved</p> <p>1 = IPv4 Bytes 0:3 4 byte IP address</p> <p>2 = IPV4+port number Bytes 0:3 4 byte IP address Bytes 4:5 2 byte IP port number</p> <p>3 = IPv6 Bytes 0:15 16 byte IP address</p> <p>4 = IPv6+port number Bytes 0:15 16 byte IP address Bytes 16:17 2 byte IP port number</p> <p>5 = SNA Bytes 0:7 EBCDIC-encoded network ID Bytes 8:15 EBCDIC-encoded network accessible unit (control point or LU)</p> <p>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.</p> <p>7:32767 = reserved</p> <p>32768-65535 = undefined and available for agent implementers to use</p>

B Examples

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 “For Your Information” on page iv.

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 arithmetic 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 arithmetic 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 int32 ;
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 {
    int32 upper;
    int32 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 */
    int32 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 */
    int32 gauge32; /* Gauge32 */
    int64 gauge64; /* Gauge64 */
    arm_gaugedivr32_t gaugedivr32; /* Gauge32 + Divisor32 */
    unsigned32 numericid32; /* NumericID32 */
    unsigned64 numericid64; /* 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 {
    int32 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 {
    int32 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 {
    int32 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 {
    int32    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_f0x80/* Correlator from parent */
#define ARM_CorrReq_f0x40/* Request correlator generation */
#define ARM_CorrGen_f0x20/* New correlator generated in data */
#define ARM_TraceReq_f0x10/* 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 int32 ARM_API arm_init(
    char*    appl_name,/* application name */
    char*    appl_user_id,/* Name of the application user */
    int32    flags,/* Reserved = 0 */
    char*    data,/* Reserved = NULL */
    int32    data_size);/* Reserved = 0 */

extern int32 ARM_API arm_getid(
    int32    appl_id,/* application handle */
    char*    tran_name,/* transaction name */
    char*    tran_detail,/* transaction additional info */
    int32    flags,/* Reserved = 0 */
    char*    data,/* format definition of user metrics */

```

```

    int32    data_size);/* length of data buffer        */

extern int32 ARM_API arm_start(
    int32    tran_id, /* transaction name identifier */
    int32    flags, /* Reserved = 0 */
    char*    data, /* user metrics data */
    int32    data_size); /* length of data buffer */

extern int32 ARM_API arm_update(
    int32    start_handle,/* unique transaction handle */
    int32    flags,/* Reserved = 0 */
    char*    data,/* user metrics data */
    int32    data_size);/* length of data buffer */

extern int32 ARM_API arm_stop(
    int32    start_handle,/* unique transaction handle */
    int32    tran_status,/* Good=0, Abort=1, Failed=2 */
    int32    flags,/* Reserved = 0 */
    char*    data,/* user metrics data */
    int32    data_size);/* length of data buffer */

extern int32 ARM_API arm_end(
    int32    appl_id,/* application id */
    int32    flags,/* Reserved = 0 */
    char*    data,/* Reserved = NULL */
    int32    data_size);/* Reserved = 0 */

#else /* _PROTOTYPES */

extern int32 ARM_API arm_init();
extern int32 ARM_API arm_getid();
extern int32 ARM_API arm_start();
extern int32 ARM_API arm_update();
extern int32 ARM_API arm_stop();
extern int32 ARM_API arm_end();

#endif /* _PROTOTYPES */

#ifdef __cplusplus
}
#endif /* __cplusplus */

/* Type definitions for compatibility with version 1.0 of the ARM API */
typedef int32 arm_appl_id_t;
typedef int32 arm_tran_id_t;
typedef int32 arm_start_handle_t;
typedef unsigned32 arm_flag_t;
typedef char arm_data_t;
typedef int32 arm_data_sz_t;
typedef char arm_ptr_t;
typedef int32 arm_ret_stat_t;
typedef int32 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"

int32 appl_id = -1; /* Define an identifier for the application id */

int32 simple_tran_id = -1; /* Define a unique identifier for each */
int32 long_tran_id_1 = -1; /* TRANSACTION */
int32 long_tran_id_2 = -1;
int32 sub_tran_id_1 = -1;
int32 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_trans1
*****/

void simple_trans1()
{
    int32 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 100000
#define UPDATE_COUNT 100000 /* call update every 100,000 iterations */

    int32 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()
{
    int32  tran_handle;
    int32  sub_tran_handle1;
    int32  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"

int32 client_appl_id = -1; /* application id */
int32 client_tran_id = -1; /* transaction id */

int32 metric_appl_id = -1; /* application id */
int32 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)
{
    int32 server_appl_id = -1; /* unique application id */
    int32 server_tran_id = -1; /* unique transaction id */
    int32 server_tran_handle = -1; /* transaction instance */

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

    int32 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()

{
    int32 client_tran_handle = -1; /* transaction start handle */

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

    int32 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 application 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 use */

    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()
{
    int32 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 */
    };

    int32 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", ""}
        }};

    int32 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);
}

```