Technical Standard

Application Response Measurement (ARM)

Issue 4.0 – C Binding

THE *Open* GROUP

Copyright © 2003, The Open Group

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the copyright owner.

It is fair use of this specification for implementers to use the names, labels, etc. contained within the specification. The intent of publication of the specification is to encourage implementations of the specification.

This specification has not been verified for avoidance of possible third-party proprietary rights. In implementing this specification, usual procedures to ensure the respect of possible third-party intellectual property rights should be followed.

Technical Standard

Application Response Measurement (ARM) Issue 4.0 – C Binding

ISBN: 1-931624-35-6

Document Number: C036

Published by The Open Group, October 2003.

Comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

Contents

1 Introduction 1
1.1 What is ARM? 1
1.2 How is ARM Used? 1
1.3 Selecting Transactions to Measure 2
1.4 The Evolution of ARM 3
1.5 Compatibility between ARM Versions 4
1.6 ARM 4.0 C Bindings Overview 4
1.7 Linking to an ARM Implementation 6
1.8 The ARM Software Developers Kit (SDK) 7
1.9 Terminology 7

2 Using ARM 4.0 C Bindings 9
2.1 Example 9

3 Programming Options 11
3.1 ARM Measures Response Times 11
3.2 Application Measures Response Time 12
3.3 Selecting which Option to Use 12

4 Understanding the Relationships between Transactions 13
4.1 A Typical Distributed Transaction 13

5 Additional Data About a Transaction 16
5.1 Diagnostic Data 16
5.2 Metric Categories 17
5.2.1 Counters 17
5.2.2 Gauges 17
5.2.3 Numeric IDs 18
5.2.4 Strings 18
5.3 How to Provide Diagnostic and Metric Data 18
5.3.1 Building Sub-Buffers 19
5.4 Processing Multiple Values of the Same Metric 19
5.4.1 Counters 20
5.4.2 Gauges 20
5.4.3 Numeric IDs 21
5.4.4 Strings 21

6 API Overview 22
6.1 Overall API Structure 22
6.2 Structure of Optional Buffer and Sub-Buffers 24
6.3 API Functions and Thread-Safe Behavior 25

6.4 Identity and Context Properties 25
6.5 Overview of API Functions to Register Metadata 26
6.6 Overview of Common API Functions to Measure Transactions 28
6.7 Overview of Other API Functions 29
6.8 Byte Order Markers in Character Strings 30

7 Error Handling Philosophy 31
7.1 Reserved Error Codes 31

8 The API Functions 32
arm_bind_thread() 33
arm_block_transaction() 34
arm_destroy_application() 36
arm_discard_transaction() 37
arm_generate_correlator() 38
arm_get_arrival_time() 40
arm_get_correlator_flags() 41
arm_get_correlator_length() 42
arm_get_error_message() 43
arm_is_charset_supported() 45
arm_register_application() 47
arm_register_metric() 49
arm_register_transaction() 52
arm_report_transaction() 54
arm_start_application() 56
arm_start_transaction() 58
arm_stop_application() 61
arm_stop_transaction() 63
arm_unbind_thread() 65
arm_unblock_transaction() 66
arm_update_transaction() 68

9 Optional Buffer and Sub-Buffers 69
9.1 User Data Buffer 69
9.2 User Sub-Buffer 72
9.3 Arrival Time Sub-Buffer 73
9.4 Metric Values Sub-Buffer 74
9.5 System Address Sub-Buffer 77
9.6 Diagnostic Detail Sub-Buffer 80
9.7 Application Identity Sub-Buffer 81
9.8 Application Context Values Sub-Buffer 83
9.9 Transaction Identity Sub-Buffer 84
9.10 Transaction Context Sub-Buffer 87
9.11 Metric Bindings Sub-Buffer 89
9.12 Character Set Encoding Sub-Buffer 91

10 <arm4.h> Header File for Compiling 93

11 <arm4os.h> Header File for Compiling 105

A Application Instrumentation Sample 106

B Information for Implementers 115

Preface

The Open Group

The Open Group, a vendor and technology-neutral consortium, has a vision of Boundaryless Information Flow achieved through global interoperability in a secure, reliable, and timely manner. The Open Group mission is to drive the creation of Boundaryless Information Flow by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate open specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

The Open Group provides opportunities to exchange information and shape the future of IT. The Open Group members include some of the largest and most influential organizations in the world. The flexible structure of The Open Group membership allows for almost any organization, no matter what their size, to join and have a voice in shaping the future of the IT world.

More information is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/testing.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/pubs.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Technical Standard for the C Binding to Application Response Measurement (ARM) Issue 4.0. It has been developed and approved by The Open Group.

ARM is a standard for measuring service levels of single-system and distributed applications. ARM measures the availability and performance of transactions, both visible to the users of the business application and those visible only within the IT infrastructure.

Typographical Conventions

The following typographical conventions are used throughout this document:

- Bold font is used in text for filenames, type names, and data structures
- Italic strings are used for emphasis. Italics in text also denote variable names and functions.
- Normal font is used for the names of constants and literals.
- Syntax and code examples are shown in fixed width font.

Trademarks

Boundaryless Information Flow™ and IT DialTone™ are trademarks and UNIX® and The Open Group® are registered trademarks of The Open Group in the United States and other countries.

Hewlett-Packard® is a registered trademark of Hewlett-Packard Company.

Java® is a registered trademark of Sun Microsystems, Inc.

Tivoli™ is a trademark of Tivoli Systems, Inc.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this document:

- Mark Johnson, IBM
- Bill Furnas, Hewlett-Packard
- Stefan Ruppert, tang-IT gmbH

Referenced Documents

The following documents are referenced in this document:

ARM 2.0 — Technical Standard, July 1998, Systems Management: Application Response Measurement (ARM) (C807), published by The Open Group.

ARM 3.0 — Technical Standard, October 2001, Application Response Measurement (ARM) Issue 3.0 Java Binding (C014), published by The Open Group.

ARM 4.0 (Java Binding)
Technical Standard, October 2003, Application Response Measurement (ARM) Issue 4.0 – Java Binding (C037), published by The Open Group.

DCE 1.1: RPC
Technical Standard, August 1997, DCE 1.1: Remote Procedure Call (C706), published by The Open Group.

IETF RFC 1321
The MD5 Message-Digest Algorithm, April 1992.

1 Introduction

1.1 What is ARM?

It is hard to imagine conducting business around the globe without computer systems, networks, and software. People distribute and search for information, communicate with each other, and transact business. Computers are increasingly faster, smaller, and less expensive. Networks are increasingly faster, have more capacity, and are more reliable. Software has evolved to better exploit the technological advances and to meet demanding new requirements. The IT infrastructure has become more complex. We have become more dependent on the business applications built on this infrastructure because they offer more services and improved productivity.

No matter how much applications change, administrators and analysts responsible for the applications care about the same things they have always cared about:

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time experienced by the end-user?
- Which sub-transactions of the user transaction take too long?
- Where are the bottlenecks?
- How many of which transactions are being used?
- How can the application and environment be tuned to be more robust and perform better?

ARM helps answer these questions. ARM is a standard for measuring service levels of single-system and distributed applications. ARM measures the availability and performance of transactions (any units of work), both those visible to the users of the business application and those visible only within the IT infrastructure, such as client/server requests to a data server.

1.2 How is ARM Used?

ARM is a means through which business applications and management applications cooperate to measure the response time and status of transactions executed by the business applications.

Applications using ARM define transactions that are meaningful within the application. Typical examples are transactions initiated by a user and transactions with servers. As shown in Figure 1, applications on clients and/or servers call ARM when transactions start and/or stop. The agent in turn communicates with management applications, as shown in Figure 2, which provide analysis and reporting of the data.

The management agent collects the status and response time, and optionally other measurements associated with the transaction. The business application, in conjunction with the agent, may also provide information to correlate parent and child transactions. For example, a transaction that is invoked on a client may drive a transaction on an application server, which in turn drives ten other transactions on other application and/or data servers. The transaction on the client would be the parent of the transaction on the application server, which in turn would be the parent of the ten other transactions.

From the application developer's perspective ARM is a set of interfaces that the application loads and calls. What happens to the data after it calls the interfaces is not the developer's concern.

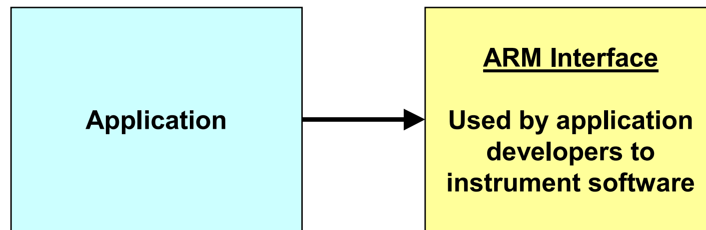

Figure 1: Application – ARM Interface

From the system administrator's perspective, ARM consists of the library that applications load and the functions in the library that the applications call, plus programs to process the data, as shown in Figure 2. How the data is processed is not part of the ARM specification, but it is, of course, important to the system administrator.

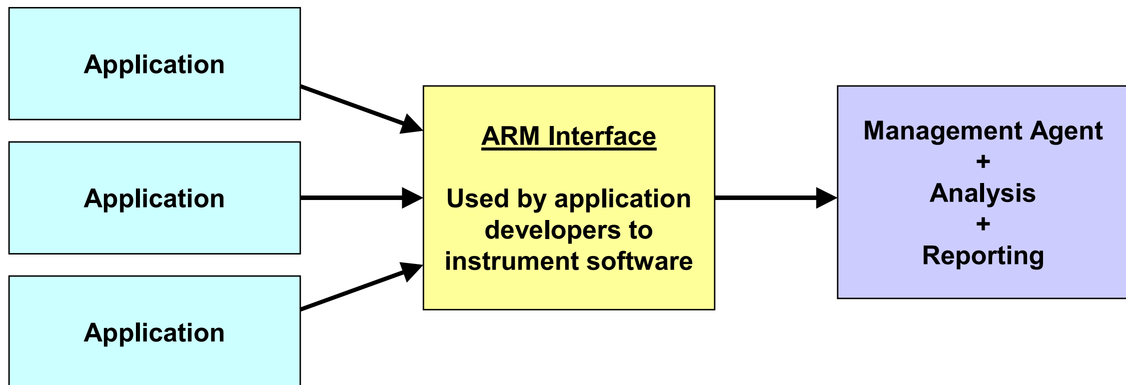

Figure 2: Application – ARM Management System Interaction

1.3 Selecting Transactions to Measure

ARM is designed to measure a unit of work, such as a business transaction, or a major component of 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.

Some questions to ask that aid in selecting which transactions to measure 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, what corrective actions will be taken?

1.4 The Evolution of ARM

ARM 1.0 was developed by Tivoli and Hewlett-Packard and released in June 1996. It provides a means to measure the response time and status of transactions. The interface is in the C programming language.

ARM 2.0 was developed by the ARM Working Group in 1997. The ARM Working Group was a consortium of vendors and end-users interested in promoting and advancing ARM. ARM 2.0 was approved as a Technical Standard of The Open Group in July 1998, part of the IT DialTone initiative. ARM 2.0 added the ability to correlate parent and child transactions, and to collect other measurements associated with the transactions, such as the number of records processed. The interface is in the C programming language.

ARM 3.0 was developed by The Open Group in 2001. It added new capabilities and specified interfaces in the Java programming language. ARM 3.0 added the following capabilities:

- Java bindings
- Changes to the length of application and transaction identifiers and handles
- The ability to report the status and response time of a transaction with a single call, executed after the transaction completes; the transaction could have executed on a different system
- The ability to identify a user on a per-transaction basis

ARM 4.0 has been developed to implement new capabilities, and to provide equivalent functions for both C and Java programs. This document describes the C program bindings. A companion document describes the Java program bindings. ARM 4.0 adds the following capabilities:

- A richer and more flexible model for specifying application and transaction identity
- Report attributes of a transaction that change on a per-instance basis
- Bind a transaction to a thread
- Indicate the amount of time a transaction is blocked waiting for an external event
- Indicate the true time when a transaction started executing for a specialized situation in which the standard indication of a start [*arm_start_transaction*() in ARM 4.0 C bindings] will not yield an accurate time

1.5 Compatibility between ARM Versions

ARM defines low-level programming interfaces to be used between programs that are dynamically linked together. This limits how much an interface can change from version to version and still link with programs using a previous version. In particular, changing function entry points, or the call signatures of an entry point, prevents working with a different version.

ARM 1.0 and ARM 2.0 are interoperable. More specifically, any application instrumenting with ARM 1.0 can link to an ARM implementation using ARM 2.0, and *vice versa*. No other versions of ARM are similarly interoperable. It is expected that management agents may simultaneously support multiple versions of ARM. For example, a product may support both the ARM 2.0 and ARM 4.0 C interfaces. However, a business application loading and using the ARM 2.0 library cannot load and link to an ARM 4.0 library, and *vice versa*.

1.6 ARM 4.0 C Bindings Overview

This specification describes an interface that consists of a set of API calls. To satisfy the requirements of the specification, all of the API calls must be implemented. The calls are typically implemented in a library that is dynamically loaded. While developing the program, programmers insert code into the application to call the library. The program is compiled using the provided header file, **<arm4.h>**.

At runtime, applications load and link to the library in their own process, as shown in Figure 3. After the library has been linked, the application (including middleware, such as an application server instrumented to call ARM) calls the library to identify itself and its transactions, and to indicate when transactions start and stop.

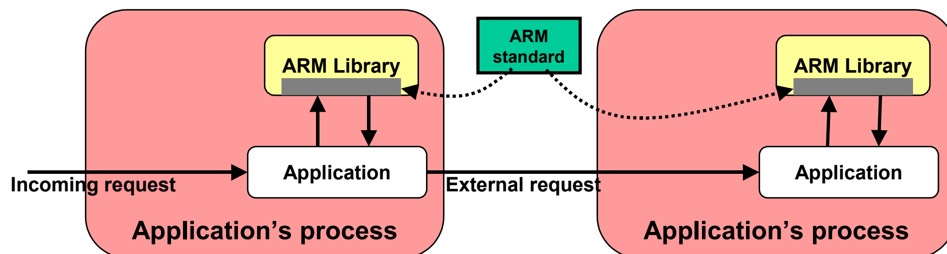

Figure 3: What ARM Looks Like to an Application

How the ARM library processes the information is transparent to the application. This characteristic is one of the strengths of the ARM architecture, because a programmer can "ARM" his or her application without being dependent on or constrained by any particular management solution. Purchasers of the application can select the solution that best meets their needs.

Figure 4 shows a typical ARM implementation. A provider of a management solution, either sold commercially or developed in-house, provides both the ARM library and a matching agent, and perhaps also management applications. The agent runs in a separate process. The ARM library and agent communicate through some sort of inter-process communications mechanism. The agent performs functions such as creating summaries, monitoring response time thresholds,

capturing trace data, managing log files, and interfacing with management applications. The management applications provide functions such as managing service-level agreements and problem determination. Figure 4 shows a separate agent for each process, but in practice there is often one agent for a system that simultaneously supports libraries in many processes.

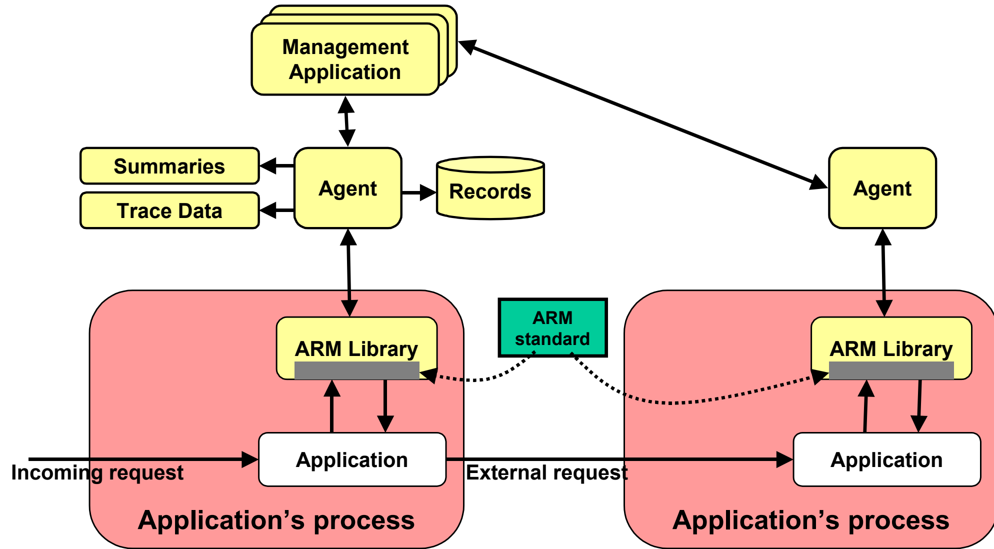

Figure 4: Behind the Scenes of the ARM Interface

1.7 Linking to an ARM Implementation

The most common and most interoperable ways to link to an ARM implementation are via a dynamically linked or loaded (shared) library. The default name of the library is **libarm4** (although some operating environments may need to add additional names in the future). The full name, including suffix, depends on the platform. Here are some example defaults:

Operating System/File System Environment	Application Mode	Library Name
HP-UX		**libarm4.sl**
IBM AIX		**libarm4.a**
IBM OS/400		**LIBARM4**
IBM zOS- Partitioned Data Set Extended	31-bit non-XPLINK	**LARM431**
	31-bit XPLINK	**LARM43X**
	64-bit	**LARM464**
IBM zOS – Hierarchical File System	31-bit non-XPLINK	**libarm4_31.so**
	31-bit XPLINK	**libarm4_3x.so**
	64-bit	**libarm4_64.so**
Linux		**libarm4.so**
Microsoft Windows		**libarm4.dll**
Sun Solaris		**libarm4.so**

Different operating systems have different conventions for how an application locates a library. Two suggested approaches are the following:

1. Provide a way for a system administrator to specify the name and path to the library as a runtime parameter. For example, the path could be in a property file. This provides maximum flexibility, and could even allow a way for different ARM implementations to be simultaneously active, each supporting a different set of applications.

2. Load a library with the default library name (e.g., **libarm4)** and assume that if one can be found somewhere in the path, it is the one to use. This would also be a good choice for a default path in approach (1) above.

ARM implementations may provide other ways to link to them, either out of necessity or choice. For example, operating systems in small pervasive devices may have no file system and no concept of dynamic linking or loading of a library. In this case, the application statically links to an ARM implementation. Some operating systems may contain the API calls in a runtime environment available to all applications without specifically linking and loading a library. Applications that use mechanisms like these become bound to that one implementation; their customers do not have the option of using a different ARM implementation.

1.8 The ARM Software Developers Kit (SDK)

The ARM 4.0 SDK also contains some optional convenience routines that can further simplify preparing data to pass when it requires additional formatting; for an example, see Section 5.3.1.

The SDK is available from regions.cmg.org/regions/cmgarmw/.

1.9 Terminology

The following terminology is used throughout this document:

Can
: Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

Implementation-defined
: (Same meaning as "implementation-dependent".) Describes a value or behavior that is not defined by this document but is selected by an implementer. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations. The implementer shall document such a value or behavior so that it can be used correctly by an application.

Legacy
: Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

May
: Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. To avoid ambiguity, the opposite of "may" is expressed as "need not", instead of "may not".

Must
: Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.

Shall
: Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

Should
: For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

Undefined
: Describes the nature of a value or behavior not defined by this document that results from use of an invalid program construct or invalid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

Unspecified
: Describes the nature of a value or behavior not specified by this document that results from use of a valid program construct or valid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

Will
: Same meaning as "shall"; "shall" is the preferred term.

2 Using ARM 4.0 C Bindings

2.1 Example

Following is a simple “Hello World” example. In a more realistic application, step 3 would repeat many times – each time a transaction executes. The other four steps are typically only done when the application initializes or terminates.

```
/* ------------------------------------------------------------------------- */
/* -------------------- ARM 4.0 C API Example: hello world ----------------- */
/* ------------------------------------------------------------------------- */

#include <stdio.h>

#include "arm4.h"

/* ------------------------------------------------------------------------- */
/* Example: simple start stop                                                */
/*                                                                           */
/* Description:                                                              */
/*                                                                           */
/* This simple hello world example shows how to instrument an application    */
/* with the ARM 4.0 C API. At first view it seems a little difficult,        */
/* but the instrumentation could be split into the following major steps:    */
/*                                                                           */
/* 1. Register any ARM class such as applications, transactions, or metrics  */
/*    using arm_register_*() calls.                                          */
/* 2. Start an application instance using arm_start_application().           */
/* 3. Start and stop any transaction using the arm_start_transaction() and   */
/*    arm_stop_transaction() calls.                                          */
/* 4. Stop the ARM application instance using arm_stop_application().        *’
/* 5. Destroy all registered ARM classes using arm_destroy_application().    */
/* ------------------------------------------------------------------------- */

static void hello_world()
{
    arm_id_t app_id;
    arm_id_t tran_id;
    arm_app_start_handle_t app_handle;
    arm_tran_start_handle_t start_handle;

    /* -------------------------- 1. step -------------------------- */
    /* Register application class with ARM agent. */
    arm_register_application("HelloWorld", ARM_ID_NONE,
        ARM_FLAG_NONE, ARM_BUF4_NONE,
        &app_id);
    /* Register transaction class with ARM agent. */
    arm_register_transaction(&app_id,
        "HelloTran", ARM_ID_NONE,
        ARM_FLAG_NONE, ARM_BUF4_NONE,
        &tran_id);
```

```
    /* -------------------------- 2. step -------------------------- */
    /* Now start the application instance for the ARM agent. */
    arm_start_application(&app_id,
        "Examples", NULL,
        ARM_FLAG_NONE, ARM_BUF4_NONE,
        &app_handle);

    /* -------------------------- 3. step -------------------------- */
    /* Now start the transaction measurement. */
    arm_start_transaction(app_handle, &tran_id, ARM_CORR_NONE,
        ARM_FLAG_NONE, ARM_BUF4_NONE,
        &start_handle, ARM_CORR_NONE);
    /* Real transaction takes place here! */
    printf("Hello world!\n");

    /* Stop the measurement and commit the transaction to ARM. */
    arm_stop_transaction(start_handle, ARM_STATUS_GOOD,
        ARM_FLAG_NONE, ARM_BUF4_NONE);

    /* -------------------------- 4. step -------------------------- */
    /* Stop the application instance. */
    arm_stop_application(app_handle, ARM_FLAG_NONE, ARM_BUF4_NONE);

    /* -------------------------- 5. step -------------------------- */
    /* Destroy all registered metadata. */
    arm_destroy_application(&app_id, ARM_FLAG_NONE, ARM_BUF4_NONE);
    return;
}

int main(int argc, char *argv[])
{
    int rc = 0;

    /* Simple start/stop example. */
    hello_world();

    return rc;
}
```

3 Programming Options

The application has two options for providing measurement data:

- In Option 1, the preferred, and more widely used option, the application calls ARM just before and after a transaction executes. Based on the time when these calls are made, ARM measures the response time and the time of day.
- In Option 2, the application makes all the measurements itself and reports the data some time later. Option 2 should only be used in situations that preclude using Option 1.

3.1 ARM Measures Response Times

Figure 5 shows Option 1, the preferred, and most widely used option. Immediately prior to starting a transaction, the application invokes *arm_start_transaction*(). The ARM 4.0 library captures and saves the timestamp and returns a unique handle. Immediately after the transaction ends, the application calls *arm_stop_transaction*(), passing the previously returned handle plus the completion status of the transaction. The ARM library captures the stop time. The difference between the stop time and the start time is the response time of the transaction.

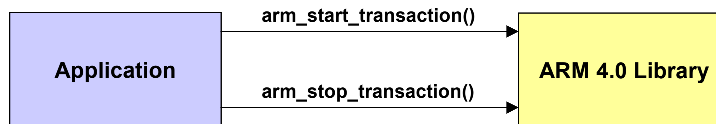

Figure 5: Measurement using Start/Stop

The application optionally provides any number of heartbeat and progress indicators using *arm_update_transaction*() between an *arm_start_transaction*() and an *arm_stop_transaction*(). This is shown in Figure 6. Heartbeats are useful for long-running transactions, such as a batch job.

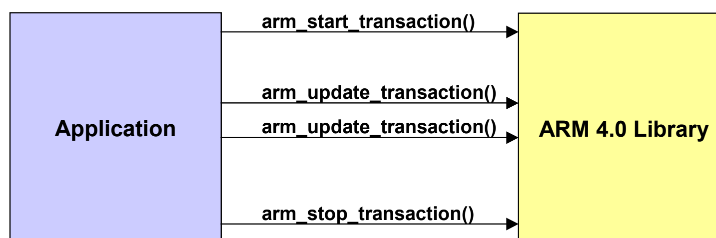

Figure 6: Application using Heartbeats

3.2 Application Measures Response Time

Figure 7 shows Option 2. The application itself measures the response time of the transaction. After the transaction completes (the delay could be short or long), it calls *arm_report_transaction*() to communicate the status, response time, and stop time to the ARM library.

Figure 7: Measurement by the Application

3.3 Selecting which Option to Use

In many situations the business application can use either programming option. In general, the recommendation is to use Option 1 (separate start and stop calls), unless that is not practical.

There is one situation for which the application must use Option 1:

- To provide heartbeats, the application must use *arm_update_transaction*() between *arm_start_transaction*() and *arm_stop_transaction*(). Heartbeats are particularly valuable for long-running transactions. An ARM implementation may process updates, such as a real-time progress display, or check a threshold for a transaction that is taking too long.

There are two situations for which applications must use Option 2 [*arm_report_transaction*()]:

- Option 1 requires that inline synchronous *arm_start_transaction*() and *arm_stop_transaction*() calls are used. The calls must be made at the moment the real transaction starts and stops. If they are not, the timings will not be accurate. If the application finds this inconvenient or impractical, it must perform the measurements itself with *arm_report_transaction*().
- If the transaction executes on System A but is reported to ARM on System B, *arm_report_transaction*() must be used for all the reasons stated above. In addition, the application provides additional information that identifies the system of the remote system where the transaction ran.

4 Understanding the Relationships between Transactions

There are several solutions available that measure transaction response times, such as measuring the response time as seen by a client, or measuring how long a method on an application server takes to complete. ARM can be used for this purpose as well. This is useful data, but it doesn't provide insight into how transactions on servers are related to business transactions executed by users or other application programs. ARM provides a facility for correlating transactions within and across systems. This section describes how this is done.

4.1 A Typical Distributed Transaction

Most modern applications consist of programs distributed across multiple systems, processes, and threads. Figure 8 is an example.

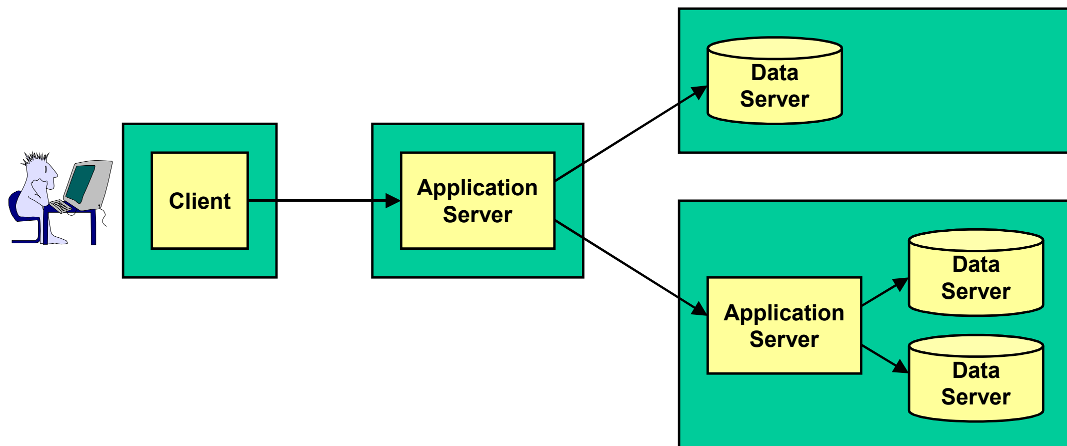

Figure 8: A Common Distributed Application Architecture

Figure 9 is an example transaction that runs on this application architecture. More correctly, Figure 9 shows a hierarchy of several transactions. To the user there is one transaction, but it is not unusual for the one transaction visible to the end-user to consist of tens or even over 100 sub-transactions.

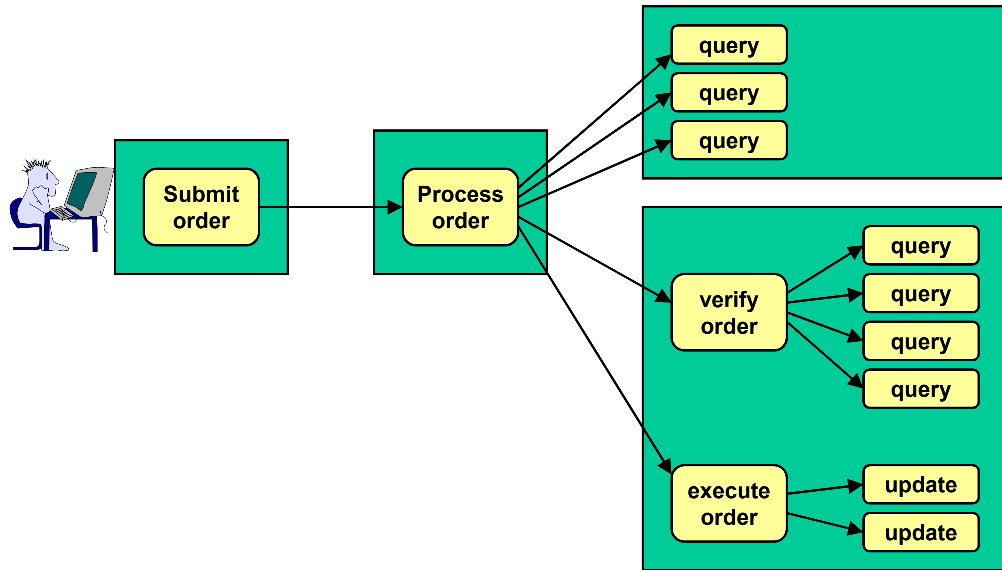

Figure 9: An Example of a Distributed Transaction

In ARM each transaction instance is assigned a unique token, named in ARM parlance a "correlator". To the application a correlator appears as an opaque byte array. There actually is a well-defined format to a correlator, and management agents and applications that understand it can take advantage of the information in it to determine where and when a transaction executed, which can aid enormously in problem diagnosis. Figure 10 shows the same transaction hierarchy as Figure 9, except that the descriptive names in Figure 9 have been replaced with identifiers. The lines are dotted instead of solid to indicate that without additional information, this would look to a management application like thirteen unrelated transactions.

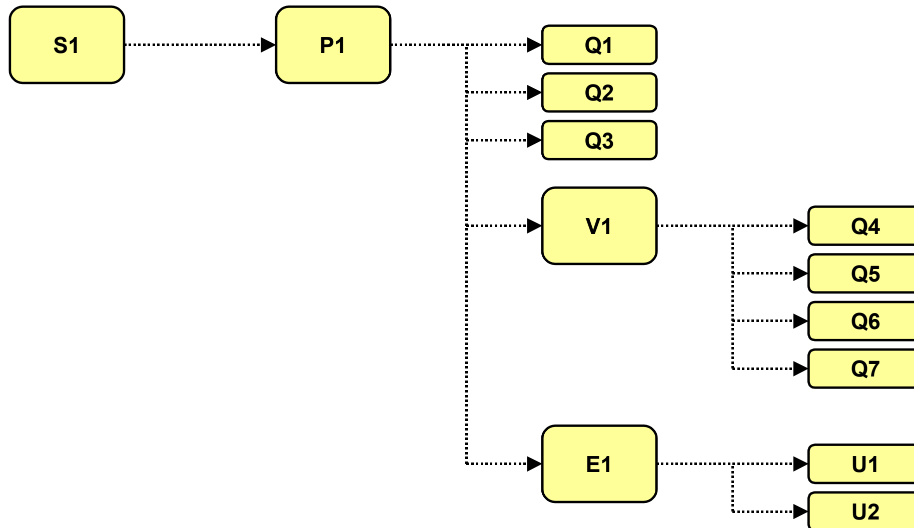

Figure 10: Distributed Transactions that Appear Unrelated

To relate the transactions together, the application components are each instrumented with ARM. In addition, each transaction passes the correlator that identifies itself to its children. In Figure 9 and Figure 10, the Submit Order transaction passes its correlator (S1) to its child, Process Order. Process Order passes its correlator (P1) to its five children – three queries, Verify Order, and Execute Order. Verify Order passes its correlator (V1) to its four children, and Execute Order passes its correlator (E1) to its two children.

The last piece in the puzzle is that each of the transactions instrumented with ARM passes its parent correlator to the ARM library. The ARM library knows the correlator of the current transaction. The correlators can be combined into a tuple of (parent correlator, correlator). Some of the tuples in Figure 10 are (S1,P1), (P1,Q1), (P1,E1), and (E1, U1). By putting the different tuples together, the management application can create the full calling hierarchy using the correlators to identify the transaction instances, as shown in Figure 11.

As an example of how this information could be used, if S1 failed, it would now be possible to determine that it failed because P1 failed, P1 failed because V1 failed, and V1 failed because Q6 failed.

Similar types of analysis could determine the source of response time problems. To analyze response time problems, additional information is needed. It's necessary to know if the child transactions execute serially, in parallel, or some combination of the two. The information may also be useful in locating unacceptable network latencies. For example, if the response time of S1 is substantially more than the response time of P1, and it is known that there is very little processing done on P1 that isn't accounted for in the measured response times, it suggests that there are unacceptable network or queuing delays between S1 and P1.

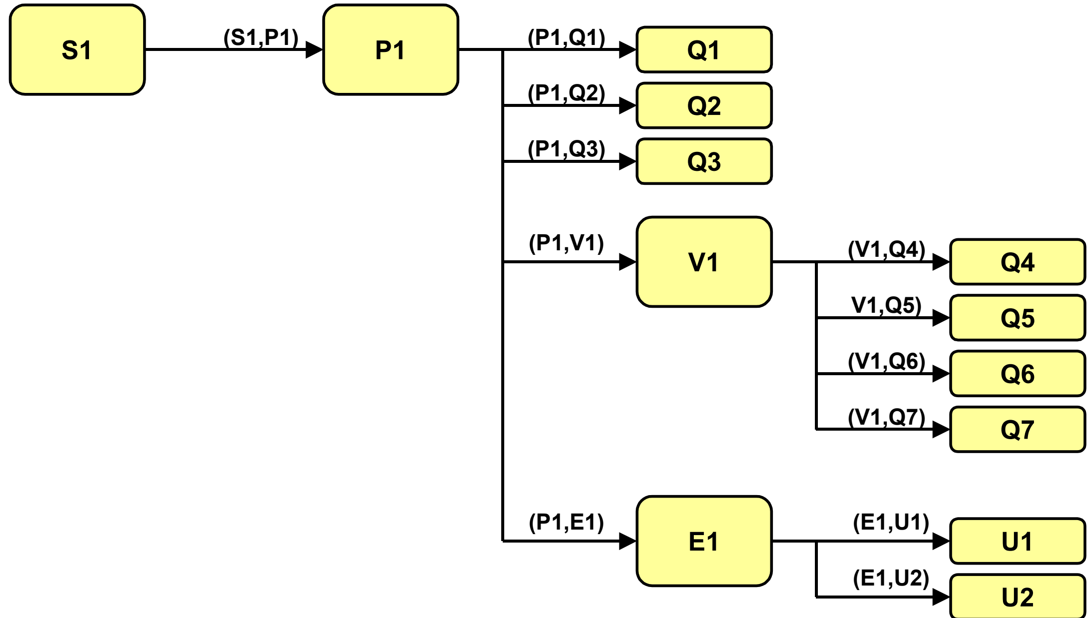

Figure 11: A Distributed Transaction Calling Hierarchy

5 Additional Data About a Transaction

The identification information and measurement information (status, response time, stop time) for any transaction measured with ARM provides a great deal of value, and there may be no requirement to augment the information. However, there are situations in which additional information could be useful, such as:

- How "big" is a transaction? Knowing a backup operation took 47 seconds may not be sufficient to know if the performance was good. Additional information, such as the number of bytes or files backed-up, provides much more meaning to the 47 seconds measurement.
- A transaction such as "get design drawings" may execute in less than a second for a simple part (e.g., a bracket). For complex parts, such as an engine, it may take many seconds to retrieve all the drawings, even if the system is performing well. Knowing the part number in this case makes the response time meaningful.
- The performance of a transaction will be affected by other workloads running on the same physical or logical system. Performance management tools may capture other information (e.g., CPU utilization) and combine it with response time measurements to plot the effect of CPU time on response time, which could be useful for planning the capacity of a system. However, other information that could be useful may not be available to performance management tools (e.g., the length of a queue internal to a program). It would be helpful for the application to provide this information.
- If a transaction fails it can be useful to know why. The required ARM status has four possible values: Good, Failed, Aborted, and Unknown. A detailed error code would be useful to understand why a transaction failed or was aborted. Capturing the code along with the other transaction information simplifies analysis by avoiding a later merge with, for example, error messages in a log file.

ARM provides two ways for applications to provide these types of data. One is a long character string, named "diagnostic data". The other is short numeric or character strings named "metrics" in ARM. The use of diagnostic data and metrics is *optional*.

ARM is not intended as a general-purpose interface for recording data. It is good practice to limit the use of metrics to data that is directly related to a transaction, and that helps to understand measurements about the transaction.

5.1 Diagnostic Data

The diagnostic data is in the form of a long null-terminated character string. It can contain any information that the application thinks may be useful. For example, if a database query fails, the application might provide the text of the SQL query.

The expected usage of diagnostic data is for unusual situations, such as errors or detailed activity traces.

5.2 Metric Categories

ARM supports nine data types. The data types are grouped in four categories. The categories are counters, gauges, numeric IDs, and strings.

5.2.1 Counters

A counter is a monotonically increasing non-negative value up to its maximum possible value, at which point it wraps around to zero and starts again. This is the IETF (Internet Engineering Task Force) definition of a 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.

ARM supports three counter types:

- 32-bit integer: (*format*=1, ARM_METRIC_FORMAT_COUNTER32)
- 64-bit integer: (*format*=2, ARM_METRIC_FORMAT_COUNTER64)
- 32-bit integer plus a 32-bit divisor, used to simulate floating-point data, without needing to specify floating-point formats:
  (*format*=3, ARM_METRIC_FORMAT_CNTRDIVR32)

5.2.2 Gauges

A gauge value can go up and down, and it can be positive or negative. This is the IETF definition of a 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 the amount of memory used over 20 transactions in an interval is measured and the average usage for each of these transactions was 15MB, it does not make sense to say that 20*15=300MB of memory were used over the interval. It would make sense to say that the average was 15MB, that the median was 12MB, and that the standard deviation was 8MB. The values can be averaged, maximums and minimums per transaction calculated, and other kinds of statistical calculations performed.

ARM supports three gauge types:

- 32-bit integer: (*format*=4, ARM_METRIC_FORMAT_GAUGE32)
- 64-bit integer: (*format*=5, ARM_METRIC_FORMAT_GAUGE64)
- 32-bit integer plus a 32-bit divisor, used to simulate floating-point data, without needing to specify floating-point formats:
  (*format*=6, ARM_METRIC_FORMAT_GAUGEDIVR32)

5.2.3 Numeric IDs

A numeric ID is a numeric value that is used as an identifier, and not as a measurement value. Examples are message numbers and error codes.

Numeric IDs are classified as non-calculable because it doesn't make sense to perform arithmetic with them. For example, the mean of the last seven message numbers would hardly ever provide useful information. By using a data type of numeric ID instead of a gauge or counter, the application indicates that arithmetic with the numbers is probably nonsensical. An agent could create statistical summaries based on these values, such as generating a frequency histogram by part number or error number.

ARM supports two numeric ID types:

- 32-bit integer: (*format*=7, ARM_METRIC_FORMAT_NUMERICID32)
- 64-bit integer: (*format*=8, ARM_METRIC_FORMAT_NUMERICID64)

5.2.4 Strings

A string is used in the same way that a numeric ID is used. It is an identifier, not a measurement value. Examples are part numbers, names, and messages.

ARM supports one string type:

- Strings of 1-32 characters: (*format*=10, ARM_METRIC_FORMAT_STRING32)

5.3 How to Provide Diagnostic and Metric Data

The application provides the values in optional formatted buffer(s), named sub-buffers. Any time one of the four measurement functions is executed (*start*(), *update*(), *stop*(), or *report*()) the ARM library tests for the existence of the sub-buffers. In addition, for the metric sub-buffer, it checks the "valid" flag for each entry to see whether metric data is being passed. If it is, the library can process it, depending on its capabilities and how it is configured.

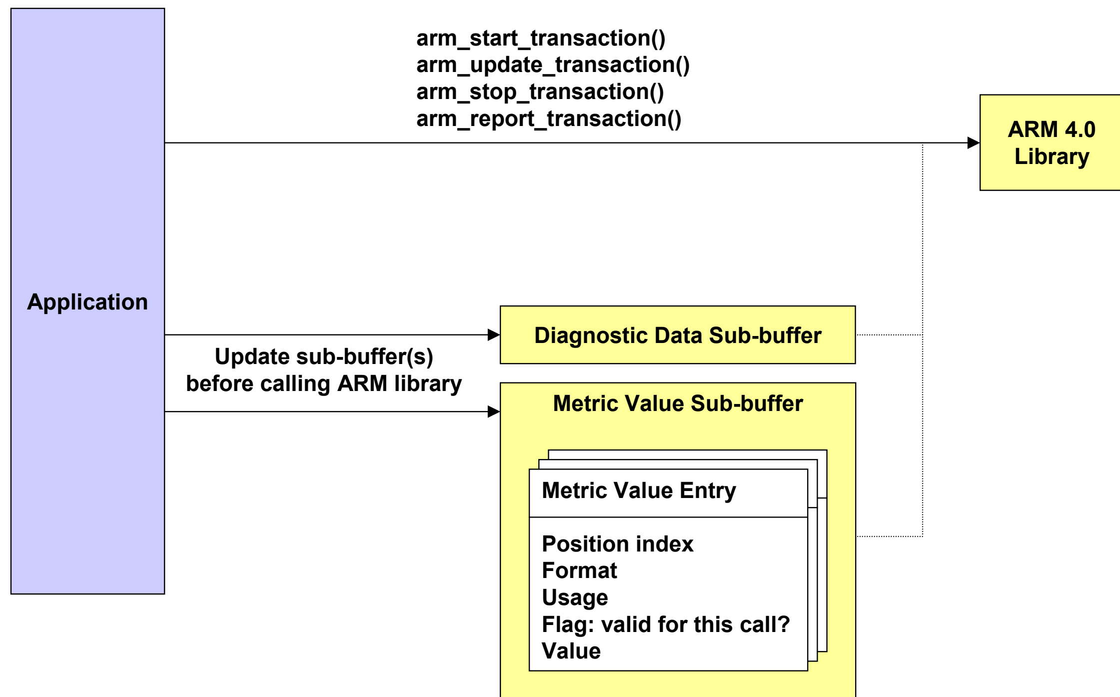

Figure 12: Providing Additional Data using Sub-Buffers

5.3.1 Building Sub-Buffers

The ARM 4.0 SDK contains a large set of convenience routines to build the buffer, sub-buffers, and embedded data structures. The use of the convenience routines is optional. For example, the *armsdk_metric_counter32_set*() function will populate a metric value entry in the metric value sub-buffer (see the prototype below), using a pre-formatted structure named **armsdk_subbuffer_metric_values_t**. Similar routines exist for all sub-buffers and embedded structures.

```
arm_error_t
armsdk_metric_counter32_set(
    armsdk_subbuffer_metric_values_t *subbuffer,
    const arm_metric_slot_t slot,
    const arm_boolean_t valid,
    const arm_metric_usage_t usage,
    const arm_metric_counter32_t value);
```

5.4 Processing Multiple Values of the Same Metric

Additional semantics are defined when using *start*(), *update*(), and *stop*() in order to eliminate ambiguity. The ambiguity arises because the metric may be valid on some or all of the *arm_start_transaction*(), *arm_update_transaction*(), and *arm_stop_transaction*() function calls. The following sections describe the semantics for each of the data type categories.

This section does not apply when using *arm_report_transaction*(), because at most one value is provided per transaction. So the value provided is the value used.

5.4.1 Counters

If a counter is used, its initial value must be set at the time of the *start*() call. The difference between the value when the *start*() executes and when the *stop*() executes (or the value in the last *update*() if no metric value is passed on *stop*()) is the value attributed to this transaction. Similarly, the difference between successive *update*()s, or from the *start*() to the first *update*(), or from the last *update*() to the *stop*(), equals the value for the time period between the respective calls.

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

- The counter is set to zero at *start*() and to some value at *stop*() (or the last *update*()). In this case, the application probably measured the value for this transaction and provided that value in the *stop*(). The application always sets the value to zero at the *start*() so the value at *stop*() reflects both the difference from the *start*() value and the absolute value. When using *arm_report_transaction*(), the value provided is equivalent to the difference between zero and the provided value.
- The counter is *x*1 at *start*(), *x*2 at its corresponding *stop*(), *x*2 at the next *start*(), and *x*3 at its corresponding *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 *x*1 at *start*(), *x*2 at *stop*(), *x*3 (not equal to *x*2) at the next *start*(), and *x*4 at its *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 *stop*() to the next *start*() for the same transaction class.

5.4.2 Gauges

Gauges can be set before *start*(), *update*(), and *stop*() calls. This creates the potential for different interpretations. If several values are provided for a transaction [e.g., one at *start*(), one at each *update*(), and one at *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 between and including the *start*() and *stop*() calls.
- The minimum value for a transaction will be the smallest valid value passed at any time between and including the *start*() and *stop*() calls.
- The mean value for a transaction will be the mean of all valid values passed at any time between and including the *start*() and *stop*() calls. All valid values will be weighted equally each time a *start*(), *update*(), or *stop*() executes.
- The median value for a transaction will be the median of all valid values passed at any time during the transaction. All valid values will be weighted equally each time a *start*(), *update*(), or *stop*() executes.

- The last value for a transaction will be the last valid value passed whenever any *start*(), *update*(), or *stop*() executes.

5.4.3 Numeric IDs

The last value passed when any of the *start*(), *update*(), or *stop*() calls is made will be the value attributed to the transaction instance. For example, if a value is valid at *start*() but not when any *update*() or *stop*() executes, the value passed at the *start*() is used. If a value is valid when *start*() executes and when *stop*() executes, the value when *stop*() executes is the value for the transaction instance. This convention is identical to the string convention.

5.4.4 Strings

The last value passed when any of the *start*(), *update*(), or *stop*() calls is made will be the value attributed to the transaction instance. For example, if a value is valid at *start*() but not when any *update*() or *stop*() executes, the value passed at the *start*() is used. If a value is valid when *start*() executes and when *stop*() executes, the value when *stop*() executes is the value for the transaction instance. This convention is identical to the numeric ID convention.

6 API Overview

6.1 Overall API Structure

Figure 13 shows the overall structure of the API calls. All calls have function parameters. Some calls also have a pointer to an optional buffer, which in turn contains sub-buffers. For function calls that use sub-buffers, the application builds the buffer and sub-buffers first (step 1), then makes the call (step 2).

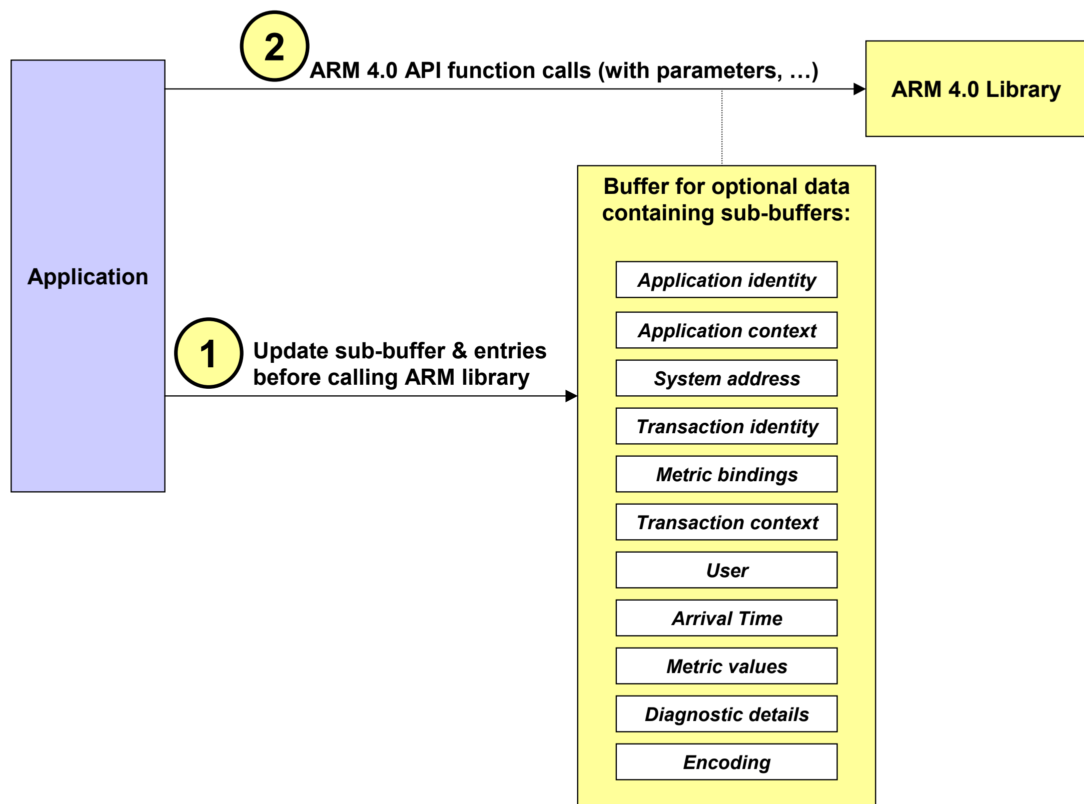

Figure 13: Overall API Structure

The valid sub-buffers vary depending on which call is being made, as shown in Figure 14. The labeled arrows represent the function names. The bulleted items in the white boxes represent the sub-buffer names.

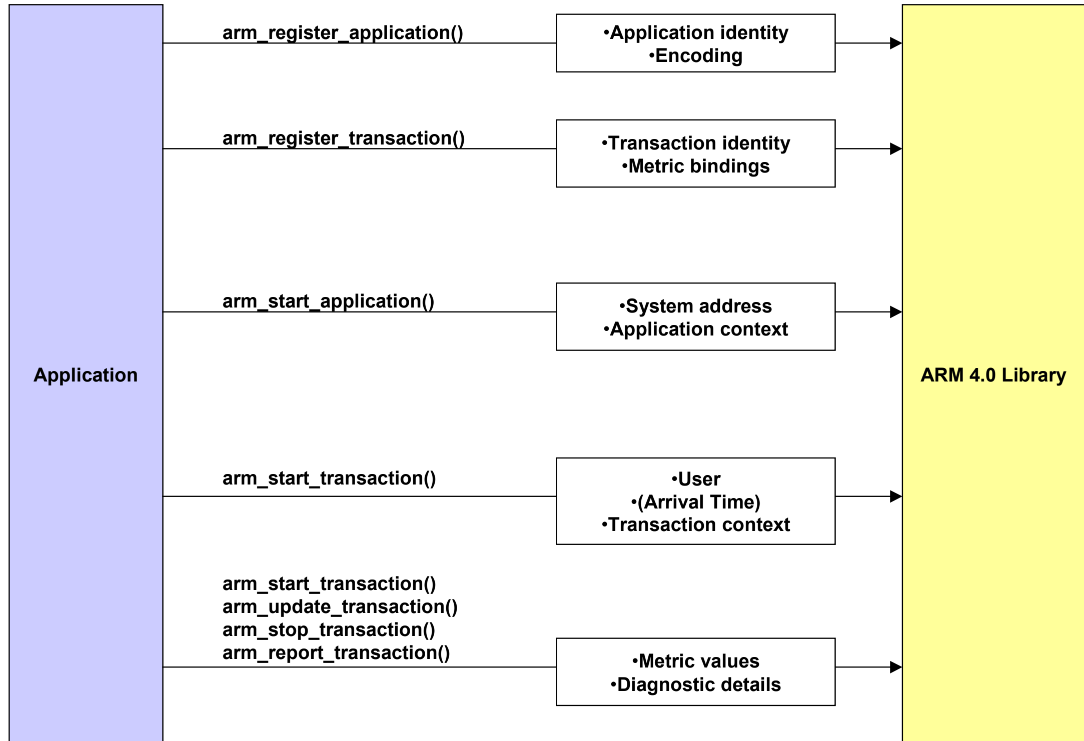

Figure 14: Valid Sub-Buffers per API Function

6.2 Structure of Optional Buffer and Sub-Buffers

Figure 15 shows the structure of the optional buffer ("Buffer4") that contains sub-buffers. The buffer is structured so that any number of sub-buffers can be supported, and the format of each sub-buffer can be determined. This allows new sub-buffer formats to be used without impacting the ability of existing ARM implementations to function correctly. If an ARM implementation encounters a sub-buffer it does not recognize, it ignores it and continues to step through the list to find any that it does know how to support.

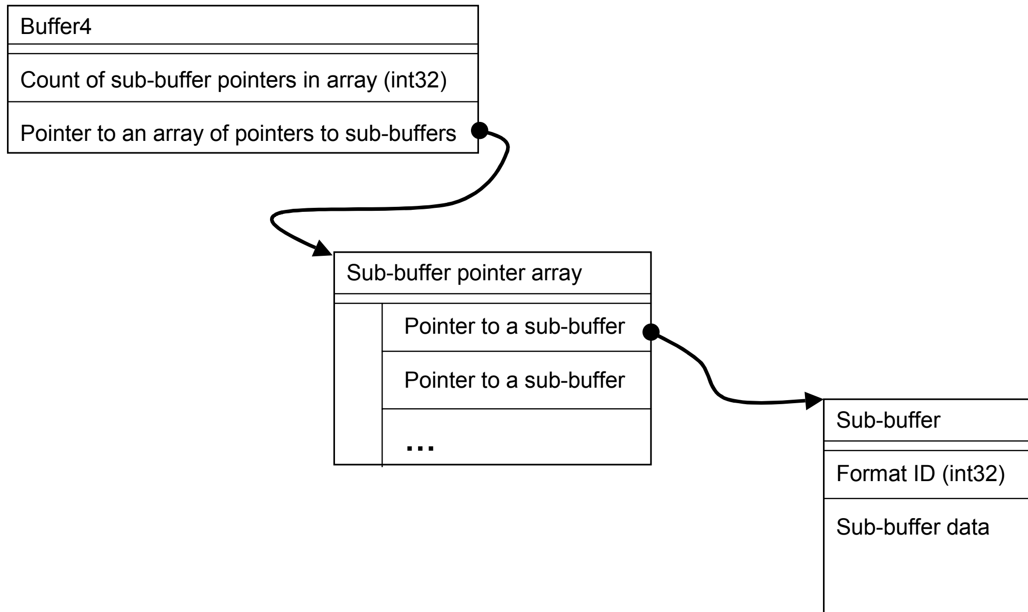

Figure 15: Structure of Optional Buffer and Sub-Buffers

6.3 API Functions and Thread-Safe Behavior

All API calls are thread-safe. On a platform that is multi-threaded, execution of a function in one thread will not impact the execution of a function in any other thread, *as long as the application doesn't share dynamic data across threads*. This would most likely happen when optional data is passed in the "Buffer4" in each API call. An example is a metric value that is changing – if it is updated in one thread while being processed by another thread, the results are unpredictable. If the application does share dynamic data across threads, the application is responsible for maintaining synchronization.

6.4 Identity and Context Properties

ARM 4.0 uses two types of properties to describe applications and transactions: "identity" properties and "context" properties.

- An identity property is a property that has the same value for all instances of an application or transaction. The *name* is an implicitly named and mandatory identity property. In addition there can be up to twenty identity properties of the pattern *name*=*value* pair, in which both *name* and *value* are separate character strings. Transactions can also have a URI that is implicitly named that is an identity property.

- A context property is also a property in which both the *name* and *value* are character strings. A context property name is the same for all instances of an application or transaction, but each instance may have different values.

  — In addition, applications have two implicitly named context properties – the group name and instance name.

  — In addition, transactions have two implicitly named context properties – the user's name and a URI.

When deciding whether to use identity or context properties, instrumenters should be aware of the trade-offs. Processing of identity properties can generally be optimized more than processing of context properties because it can be done once at registration time and apply to all transaction instances. Processing of context properties may occur for every transaction instance. On the other hand, if there are many combinations of identity properties, there may be an excessive number of different transaction definitions, which could result in reports that are not as useful as they might otherwise be.

6.5 Overview of API Functions to Register Metadata

Before any transactions can be measured, metadata describing the application and the transaction, and any metrics that are used, are registered. Figure 16 shows the three registration functions.

- The labeled arrows represent the function names.
- The boxes represent the function parameters and sub-buffers. In the box:
  - — Mandatory parameters are above the line in the box. Optional parameters are below the line.
  - — Optional data that is in italics is passed in sub-buffers. Optional data that is not in italics is passed as function parameters.

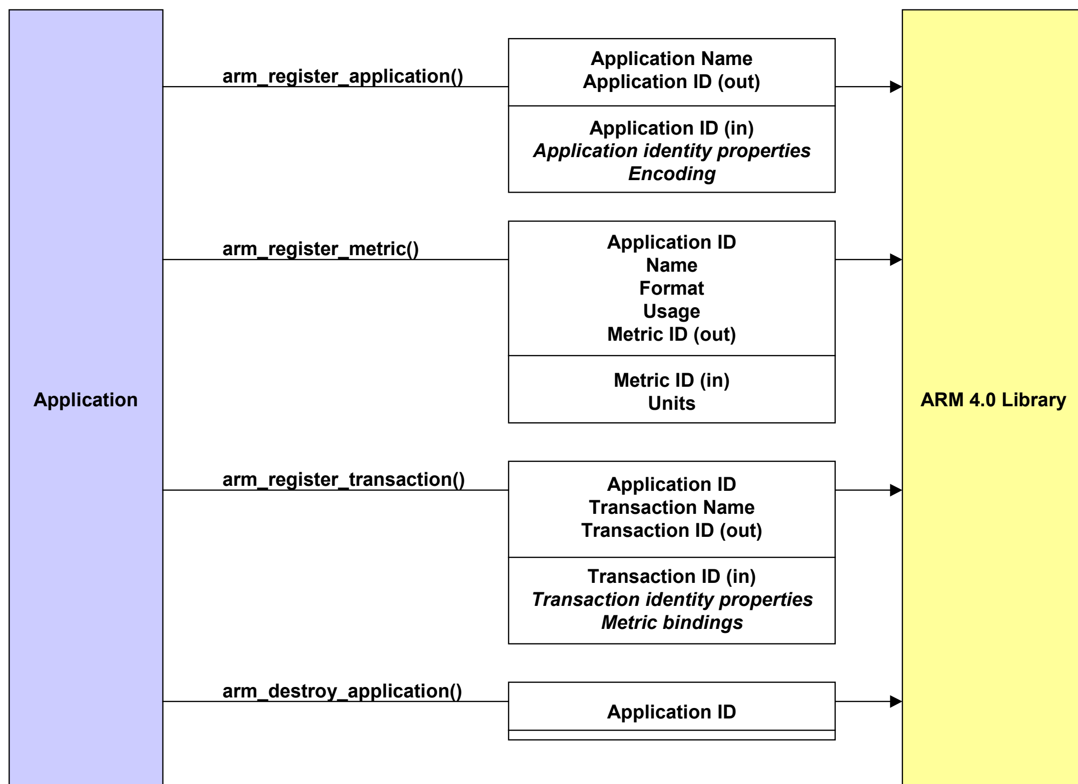

Figure 16: API Functions to Register Metadata

arm_register_application() establishes the identity of an application by a mandatory name (e.g., "Acme Billing Application Version 2.3") and optional identity properties. In most cases it is executed once when an application first loads the ARM library. However, it could be executed several times in the same process if there are multiple logical applications, or (for example) if middleware calls ARM *in lieu* of having the application itself do it. In this case, the middleware

would register an application for each application. There could be many instances of the same registered application active in the process at once.

arm_register_metric() is executed once for each unique metric. If a transaction uses metrics, the metrics it uses must be registered before the transaction is registered. Registration establishes the name, format, and special usage of the metric (if any), and optionally a string indicating the units (such as, “files backed up”).

arm_register_transaction() is executed once for each unique transaction. It establishes the identity of a transaction by a mandatory name (e.g., “Query Balance Due”) and optional identity properties. It also binds registered metric definitions to the transaction.

arm_destroy_application() indicates that no more API calls will be executed for this application. It can be treated as a signal that the ARM library can discard any data and storage it is holding for the application.

6.6 Overview of Common API Functions to Measure Transactions

Figure 17 lists the most commonly used functions after the metadata has been registered and the application is processing transactions.

- The labeled arrows represent the function names.
- The boxes represent the function parameters and sub-buffers. In the box:
  - Mandatory parameters are above the line in the box. Optional parameters are below the line.
  - Optional data that is in italics is passed in sub-buffers. Optional data that is not in italics is passed as function parameters.

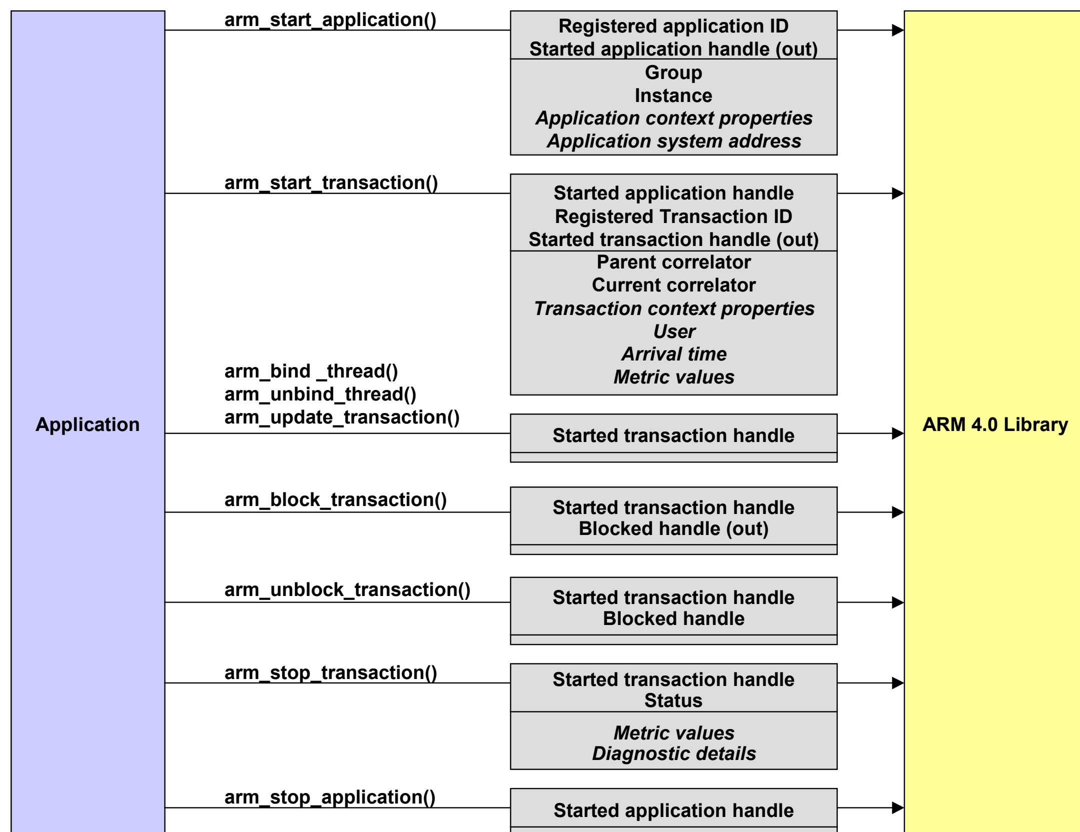

Figure 17: Most Frequently Used API Functions to Measure Transactions

arm_start_application() indicates that an instance of an application is executing. *arm_stop_application*() is the inverse function. It indicates that the application instance will not make any more ARM calls (which serves as a strong hint to the ARM implementation to release memory associated with this instance).

arm_start_transaction() indicates that a transaction has started. *arm_stop_transaction*() indicates that the transaction has completed. The process was described in Section 3.1.

arm_bind_thread() indicates that a thread is executing on behalf of the specified transaction. It is executed after *arm_start_transaction*() and before *arm_stop_transaction*(). *arm_unbind_thread*() is the inverse function.

arm_block_transaction() indicates that a transaction is blocked waiting on an external event (which may or may not be a child transaction). *arm_unblock_transaction*() is the inverse function.

6.7 Overview of Other API Functions

In addition to the functions in Figure 17, there are other functions that are used for specialized purposes.

arm_discard_transaction() is executed when for some reason a previously issued *arm_start_transaction*() should be ignored.

arm_generate_correlator() is used by applications that use *arm_report_transaction*() (see below) and that use correlators.

arm_get_arrival_time() is used by applications that incur significant delays after processing begins before *arm_start_transaction*() can be executed. This most commonly occurs when a query must be issued to retrieve the transaction context properties, and the query may take a non-trivial amount of time to process. (A team writing a plug-in for the Apache web server first observed this phenomenon.) *arm_get_arrival_time*() returns an integer that represents the time when the *arm_get_arrival_time*() executed. This integer can be passed in a sub-buffer to *arm_start_transaction*() so the arrival time is used as the start time.

arm_get_correlator_flag() returns the value of the specified flag in the correlator header.

arm_get_correlator_length() returns the length of a correlator so the application knows how many bytes to transmit to applications it calls.

arm_get_error_message() returns a character string that is associated with a non-zero return code from a function call.

arm_report_transaction() can be used in place of *arm_start_transaction*() and *arm_stop_transaction*(), as described in Chapter 3.

arm_update_transaction() can be used as a heartbeat or as a way to pass metric or diagnostic data while a transaction executes. This was described in Section 3.1.

6.8 Byte Order Markers in Character Strings

Applications must not pass in byte order marker characters in strings, even if the application is using a character set that defines such characters. They are not needed because strings are not being passed between systems that may use different byte orders. The application is responsible for stripping the character before calling the API.

7 Error Handling Philosophy

The error handling philosophy of the ARM specification can be summed up as the following:

"Programmers and system administrators need to know about errors; programs do not."

The practical effect of this philosophy is that applications do not need to check for errors, except when initially loading and linking to a library.

Many functions return an error code that the application may *optionally* test. If the value is not zero, an error occurred. However, any other data that is returned (in an *out* parameter) is usable without causing the program to fail, even when an error occurs. The measurements may be useless, but the program will not fail, even if the returned data is passed back to ARM on a later function call.

For example, an application may issue *arm_start_transaction*() using an ID that has not been registered. The ARM implementation will return a handle that can be input to *arm_stop_transaction*() without causing the program to fail, and the implementation may return an error code as well. In this case, the ARM implementation will probably discard the measurement data, and note the error in some way, such as writing a message to a log file.

An application that contains programming errors, or that receives invalid data, could generate invalid measurement data. This is a problem that programmers and system administrators should correct. But at runtime there's nothing an application can do about it, so the ARM interface takes the approach of being as unobtrusive as possible, and permitting the application logic to flow normally. Programmers testing programs, and system administrators managing systems using ARM, should check for error reports from ARM implementations.

Applications that want to test the error codes and report the error may use the *arm_get_error_message*() function to get a character string error message, which could then be written to a log file, for example.

7.1 Reserved Error Codes

At present all error return codes are specific to the ARM library. However, a range has been reserved for possible future use. The range is –20999 to –20000, inclusive. The ARM library must never return an error code in this range, unless the ARM specification assigns a value.

8 The API Functions

This chapter defines the ARM 4.0 API functions.

arm_bind_thread()

NAME

arm_bind_thread() – bind thread

SYNOPSIS

```
arm_error_t
arm_bind_thread(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[opt in]*/ const arm_int32_t flags,        /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);  /*no current use*/
```

DESCRIPTION

arm_bind_thread() indicates that the thread from which it is called is performing on behalf of the transaction identified by the start handle.

The thread binding could be useful for managing computing resources at a finer level of granularity than a process. There can be any number of threads simultaneously bound to the same transaction.

A transaction remains bound to a thread until either an *arm_discard_transaction*(), *arm_stop_transaction*(), or *arm_unbind_thread*() is executed passing the same start handle.

arm_bind_thread() and *arm_block_transaction*() are used independently of each other.

PARAMETERS

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_block_transaction(), *arm_discard_transaction*(), *arm_start_transaction*(), *arm_stop_transaction*(), *arm_unbind_thread*()

arm_block_transaction()

NAME

arm_block_transaction() – block transaction

SYNOPSIS

```
arm_error_t
arm_block_transaction(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[opt in]*/ const arm_int32_t flags,        /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4,   /*no current use*/
    /*[out]*/ arm_tran_block_handle_t *block_handle);
```

DESCRIPTION

arm_block_transaction() is used to indicate that the transaction instance is blocked waiting on an external transaction (which may or may not be instrumented with ARM) or some other event to complete. It has been found useful to separate out this "blocked" time from the elapsed time between the *arm_start_transaction*() and *arm_stop_transaction*().

A transaction remains blocked until *arm_unblock_transaction*() is executed passing the same *tran_block_handle*, or either an *arm_discard_transaction*() or *arm_stop_transaction*() is executed passing the same *tran_handle*.

The blocking conditions of most interest are those that could result in a significant and/or variable length delay relative to the response time of the transaction. For example, a remote procedure call would be a good situation to indicate with *arm_block_transaction*() or *arm_unblock_transaction*(), whereas a disk I/O would not.

A transaction may be blocked by multiple conditions simultaneously. In many application architectures *arm_block_transaction*() would be called just prior to a thread suspending, because the thread is waiting to be signaled that an event has occurred. In other application architectures there would not be a tight relationship between the thread behavior and the blocking conditions. *arm_bind_thread*() and *arm_block_transaction*() are used independently of each other.

PARAMETERS

block_handle
: Pointer to a handle that is passed on *arm_unblock_transaction*() calls in the same process. There are no requirements on what value it is set to, except that it must be possible to pass it on *arm_unblock_transaction*() without the application needing to do any error checking.

buffer4
: Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags
: Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle
: A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_bind_thread(), *arm_discard_transaction*(), *arm_start_transaction*(), *arm_stop_transaction*(), *arm_unblock_transaction*()

arm_destroy_application()

NAME

arm_destroy_application() – destroy application data

SYNOPSIS

```
arm_error_t
arm_destroy_application(
    /*[in]*/ const arm_id_t *app_id,
    /*[opt in]*/ const arm_int32_t flags,        /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);  /*no current use*/
```

DESCRIPTION

arm_destroy_application() indicates that the registration data about an application previously registered with *arm_register_application*() is no longer needed.

The purpose of this call is to signal the ARM implementation so it can release any storage it is holding. Ending a process or unloading the ARM library results in an implicit *arm_destroy_application*() for any previously registered applications.

It is possible for multiple *arm_register_application*() calls to be made in the same process with identical identity data. When this is done, *arm_destroy_application*() is assumed to be paired with one *arm_register_application*(). For example, if *arm_register_application*() is executed twice with the same output ID, and *arm_destroy_application*() is executed once using this ID, it is assumed that an instance of the application is still active and it is not safe to discard the application registration data associated with the ID.

PARAMETERS

app_id — Application ID returned from an *arm_register_application*() call in the same process.

buffer4 — Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null.

flags — Contains 32-bit flags. No values are currently defined. The field should be zero.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_application()

arm_discard_transaction()

NAME

arm_discard_transaction() – discard transaction

SYNOPSIS

```
arm_error_t
arm_discard_transaction(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[opt in]*/ const arm_int32_t flags,        /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);  /*no current use*/
```

DESCRIPTION

arm_discard_transaction() signals that the referenced *arm_start_transaction*() should be ignored and treated as if it never happened. Measurements associated with a transaction that is processing should be discarded. Either *arm_discard_transaction*() or *arm_stop_transaction*() is used – never both.

An example of when a transaction would be discarded could happen is if proxy instrumentation believes a transaction is starting, but then learns that it did not. It can be called from any thread in the process that executed the *arm_start_transaction*(). In general, the use of *arm_discard_transaction*() is discouraged, but experience has shown a few use cases that require the functionality.

arm_discard_transaction() clears any thread bindings [*arm_bind_thread*()] and blocking conditions [*arm_block_transaction*()].

PARAMETERS

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_bind_thread(), *arm_block_transaction*(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_generate_correlator()

NAME

arm_generate_correlator() – generate a correlator

SYNOPSIS

```
arm_error_t
arm_generate_correlator(
    /*[in]*/ const arm_app_start_handle_t app_handle,
    /*[in]*/ const arm_id_t *tran_id,
    /*[opt in]*/ const arm_correlator_t *parent_correlator,
    /*[opt in]*/ const arm_int32_t flags,
    /*[opt in]*/ const arm_buffer4_t *buffer4,
    /*[out]*/ arm_correlator_t *current_correlator);
```

DESCRIPTION

arm_generate_correlator() is used to generate a correlator for use with *arm_report_transaction*().

A correlator is a correlation token passed from a calling transaction to a called transaction. The correlation token may be used to establish a calling hierarchy across processes and systems. A correlator contains a two-byte length field, a one-byte format ID, a one-byte flag field, plus it may contain other data that is used to uniquely identify an instance of a transaction. Applications do not need to understand correlator internals. The maximum length is 512 (ARM_CORR_MAX_LENGTH) bytes.

It is useful to think about its use in the context of what *arm_start_transaction*() and *arm_stop_transaction*() do:

- *arm_start_transaction*() performs two functions. It establishes the identity of a transaction instance (encapsulated in the current correlator) and begins the measurements of the instance. *arm_stop_transaction*() causes the measurements to be captured. The start handle links an *arm_start_transaction*() and an *arm_stop_transaction*().
- *arm_generate_correlator*() establishes the identity of a transaction instance, like *arm_start_transaction*(), also encapsulating it in a correlator. It has no relationship to measurements about the transaction instance. *arm_report_transaction*() combines the measurement function of both *arm_start_transaction*() and *arm_stop_transaction*().

Based on this positioning, it should be clear that *arm_generate_correlator*() can be used whenever an *arm_start_transaction*() can be used. More specifically, the following calls must have been executed first: *arm_register_application*(), *arm_register_transaction*(), and *arm_start_application*(). The same parameters are also used, except there's no need for a start handle, and there's no need for the arrival time or metric values sub-buffers. The other sub-buffers may be used. The correlator that is output can be used in the same way that a correlator output from *arm_start_transaction*() is used.

PARAMETERS

app_handle The value returned from an *arm_start_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are **arm_subbuffer_trancontext_t** and **arm_subbuffer_user_t**.

current_correlator
A pointer to a buffer into which the ARM implementation will store a correlator for the transaction instance, if any. The length of the buffer should be (at least) ARM_CORR_MAX_LENGTH. The value must be non-null (because it's the only purpose for calling the function).

flags Contains 32-bit flags.

ARM_FLAG_TRACE_REQUEST (0x00000001) is 1 if the application requests/suggests a trace.

ARM_FLAG_CORR_IN_PROCESS (0x00000004) is 1 if the application is stating that it will not send the correlator outside the current process. An ARM implementation may be able to optimize correlator handling if it knows this, because it may be able to avoid serialization to create the correlator.

parent_correlator
A pointer to the parent correlator, if any. The pointer may be null.

tran_id A transaction ID returned in an *out* parameter from an *arm_register_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_application(), *arm_register_transaction*(), *arm_report_transaction*(), *arm_start_application*(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_get_arrival_time()

NAME

arm_get_arrival_time() – store current time

SYNOPSIS

```
arm_error_t
arm_get_arrival_time(
    /*[out]*/ arm_arrival_time_t *opaque_time);
```

DESCRIPTION

arm_get_arrival_time() stores a 64-bit integer representing the current time.

There are situations in which there is a significant delay between the time when processing of a transaction begins and when all the context property values that are needed before *arm_start_transaction*() can be executed are known. In order to get a more accurate response time, *arm_get_arrival_time*() can be used to capture an implementation-defined representation of the current time. This integer value is later stored in the arrival time sub-buffer when *arm_start_transaction*() executes. The ARM library will use the "arrival time" as the start time rather than the moment when the *arm_start_transaction*() executes.

PARAMETERS

opaque_time

Pointer to an **int64** that will contain the arrival time value. Note that the value is implementation-defined so the application should not make any conclusions based on its contents.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_start_transaction()

arm_get_correlator_flags()

NAME

arm_get_correlator_flags() – get value of flag

SYNOPSIS

```
arm_error_t
arm_get_correlator_flags(
    /*[in]*/ const arm_correlator_t *correlator,
    /*[in]*/ const arm_int32_t corr_flag_num,
    /*[out]*/ arm_boolean_t *flag);
```

DESCRIPTION

arm_get_correlator_flags() returns the value of a specified flag in the correlator header.

A correlator header contains bit flags. *arm_get_correlator_flags*() is used to test the value of those flags. See *arm_generate_correlator*() for a description of a correlator.

PARAMETERS

corr_flag_num
: An enumerated value that indicates which flag's value is requested. The enumerated values are:

  1 = Application trace flag (**#define** ARM_CORR_FLAGNUM_APP_TRACE)
  2 = Agent trace flag (**#define** ARM_CORR_FLAGNUM_AGENT_TRACE)

correlator
: Pointer to a buffer containing a correlator. It serves no purpose to make the call if the pointer is null.

flag
: Pointer to a boolean that is output indicating whether the flag is set.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_generate_correlator()

arm_get_correlator_length()

NAME

arm_get_correlator_length() – get length of correlator

SYNOPSIS

```
arm_error_t
arm_get_correlator_length(
    /*[in]*/ const arm_correlator_t *correlator,
    /*[out]*/ arm_correlator_length_t *length);
```

DESCRIPTION

arm_get_correlator_length() returns the length of a correlator, based on the length field within the correlator header. Note that this length is not necessarily the length of the buffer containing the correlator.

A correlator header contains a *length* field. *arm_get_correlator_length*() is used to return the length. The function handles any required conversion from the network byte order used in the header and the endian (big *versus* little) of the platform. See *arm_generate_correlator*() for a description of a correlator.

PARAMETERS

correlator Pointer to a buffer containing a correlator. It serves no purpose to make the call if the pointer is null.

length Pointer to an **int16** into which the ARM implementation will store the length. It serves no purpose to make the call if the pointer is null. If the pointer is not null, some value will be stored. The stored value will be the actual value if the value is apparently correct; otherwise, it will be zero. Examples of when zero would be stored are when the input correlator pointer is null or the length field is invalid, such as being greater than ARM_CORR_MAX_LENGTH.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_generate_correlator()

arm_get_error_message()

NAME

arm_get_error_message() – get error message

SYNOPSIS

```
arm_error_t
arm_get_error_message(
    /*[in]*/ const arm_charset_t charset, /* an IANA MIBenum value */
    /*[in]*/ const arm_error_t code,
    /*[out]*/ arm_message_buffer_t msg);
```

DESCRIPTION

arm_get_error_message() stores a string containing an error message for the specified error code.

ARM implementations return values that are specific to the implementation. The only enforced convention is that a return code of zero indicates that no errors are *reported* (though an error could have occurred), and a negative return code indicates that some error occurred. Some implementations may report an error at times when another implementation would not.

To help an application developer or administrator understand what a negative error code means, *arm_get_error_message*() can be used to store a string containing an error message for the specified error code. The ARM library is not obliged to return a message, even if it returned a non-zero return code.

PARAMETERS

charset An IANA (Internet Assigned Numbers Authority – see www.iana.org/) MIBenum value [see *arm_is_charset_supported*()]. If a non-null message is returned, it will be in this encoding. It is strongly recommended that no value be used for *charset* that has not been tested for support by the library using *arm_is_charset_supported*().

code An error code returned as **arm_error_t** from an API call.

msg Pointer to a buffer that can contain 256 characters (including the termination character) into which the null-terminated error message will be copied. The message will be in the encoding specified by the *charset* parameter. If the implementation cannot honor the request, the implementation must store at least the null termination character (e.g., which it would do if it does not return a message or does not recognize the error code or cannot return the message in the application's encoding). The function is ignored if the pointer is null and an error status may be returned.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is

returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_is_charset_supported()

arm_is_charset_supported()

NAME

arm_is_charset_supported() – check character encoding

SYNOPSIS

```
arm_error_t
arm_is_charset_supported(
    /*[in]*/ const arm_charset_t charset, /* an IANA MIBenum value */
    /*[out]*/ arm_boolean_t *supported);
```

DESCRIPTION

arm_is_charset_supported() indicates whether an ARM library supports a specified character encoding.

The default encoding for all strings provided by the application on all operating systems is UTF-8. An application may specify an alternate encoding when executing *arm_register_application*(), and then use it for all strings it provides on all calls including *arm_register_application*(), but should never do so without first issuing *arm_is_charset_supported*() to test whether the value is supported.

An ARM library on the operating systems listed in Table 1 must support the specified encodings. Applications are encouraged to use one of these encodings to ensure that any ARM implementation will support the application's ARM instrumentation. Another alternative is to use one of these encodings along with a preferred encoding. If the ARM library supports the preferred encoding, it is used; otherwise, one of the mandatory encodings is used.

IANA MIBenum		**Character Set Common Name**	**UNIX and Linux**	**Microsoft Windows**	**IBM OS/400**	**IBM zOS**
3	ARM_CHARSET_ASCII	ASCII-7 (US-ASCII)	Yes	Yes	Yes	Yes
106	ARM_CHARSET_UTF8	UTF-8 (UCS-2 characters only)	Yes	Yes	Yes	Yes
1014	ARM_CHARSET_UTF16LE	UTF-16 Little Endian (UCS-2 characters only)		Yes		
2028	ARM_CHARSET_IBM037	IBM037			Yes	
2102	ARM_CHARSET_IBM1047	IBM1047				Yes

Table 1: Mandatory Encodings by Platform

PARAMETERS

charset An IANA (Internet Assigned Numbers Authority – see www.iana.org) MIBenum value. Support for some values is mandatory by any ARM implementation on a specified platform, as shown in the table.

supported Pointer to a boolean value that is set to true or false to indicate whether *charset* is a supported encoding.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_application()

arm_register_application()

NAME

arm_register_application() – describe application

SYNOPSIS

```
arm_error_t
arm_register_application(
    /*[in]*/ const arm_char_t *app_name,
    /*[opt in]*/ const arm_id_t *input_app_id,
    /*[opt in]*/ const arm_int32_t flags,    /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4,
    /*[out]*/ arm_id_t *output_app_id;
```

DESCRIPTION

arm_register_application() describes metadata about an application.

The application uses *arm_register_application*() to inform the ARM library of metadata about the application. This metadata does not change from one application instance to another. It contains part of the function of the ARM 2.0 call *arm_init*(); *arm_start_application*() contains the other part.

ARM generates an ID that is passed in *arm_register_transaction*() and *arm_start_application*().

PARAMETERS

app_name Pointer to a null-terminated string containing the name of the application. The maximum length of the name is 128 characters, including the termination character. It serves no purpose and is illegal to make this call if the pointer is null or the string is blank.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffer formats that might be used are **arm_subbuffer_app_identity_t** and **arm_subbuffer_encoding_t**.

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

input_app_id

Pointer to an optional ID that is *unique* and that can be treated as an alias for the other metadata. It can be any value except all zeros or all ones. If the pointer is null, no ID is provided.

An ID is *unique* if the probability of the ID being associated with more than one set of metadata is vanishingly small. The selection of 128-bit IDs yields 3.4 x 10**38 unique IDs, so the objective is to select an ID that makes use of all 128 bits and is reasonably likely to not be selected by another person creating an ID of the same form. Two suggested algorithms that generate 128-bit values with these characteristics are:

1. The Universal Unique Identifier (UUID) algorithm that is part of The Open

Group specification: *DCE 1.1: Remote Procedure Call*. A developer could use the algorithm at the time the application is developed using a utility on his or her system, and be reasonably certain that nobody else would generate the same 128-bit ID.

2. The MD5 Message-Digest Algorithm, described in IETF RFC 1321. Applying this algorithm to a concatenation of all the metadata properties would almost certainly result in a value that would not collide with any other ID created with a different set of metadata properties.

If an ARM implementation is passed an ID that was previously registered within this process, the implementation can ignore the other metadata parameters and assume they are identical to the previously registered metadata. The application metadata consists of the following fields: *app_name* and the **arm_subbuffer_app_identity_t** sub-buffer passed in *buffer4*.

output_app_id

Pointer to a 16-byte field aligned on an 8-byte boundary. ARM will store a 16-byte value. There are no requirements on what value it is set to, except that it must be possible to pass it on other calls, such as *arm_start_application*(), without the application needing to do any error checking.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_transaction(), *arm_start_application*()

arm_register_metric()

NAME

arm_register_metric() – describe metrics

SYNOPSIS

```
arm_error_t
arm_register_metric(
    /*[in]*/ const arm_id_t *app_id,
    /*[in]*/ const arm_char_t *name,
    /*[in]*/ const arm_metric_format_t format,
    /*[in]*/ const arm_metric_usage_t usage,
    /*[opt in]*/ const arm_char_t *unit,
    /*[opt in]*/ const arm_id_t *input_metric_id,
    /*[opt in]*/ const arm_int32_t flags,        /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4,   /*no current use*/
    /*[out]*/ arm_id_t *output_metric_id);
```

DESCRIPTION

arm_register_metric() describes metadata about a metric.

The application uses *arm_register_metric*() to inform the ARM library of metadata about each metric the application provides.

ARM generates an ID that is passed in the metric binding sub-buffer to *arm_register_transaction*().

PARAMETERS

app_id Application ID returned from an *arm_register_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

format Describes the data type. The value must be one of the following values.

1 = ARM_METRIC_FORMAT_COUNTER32 = int32 counter
2 = ARM_METRIC_FORMAT_COUNTER64 int64 counter
3 = ARM_METRIC_FORMAT_CNTRDIVR = int32 counter + int32 divisor
4 = ARM_METRIC_FORMAT_GAUGE32 = int32 gauge
5 = ARM_METRIC_FORMAT_GAUGE64 = int64 gauge
6 = ARM_METRIC_FORMAT_GAUGEDIVR32 = int32 gauge + int32 divisor
7 = ARM_METRIC_FORMAT_NUMERICID32 = int32 numeric ID
8 = ARM_METRIC_FORMAT_NUMERICID64 = int64 numeric ID
9 = (DEPRECATED)
10 = ARM_METRIC_FORMAT_STRING32 = string (null-terminated) of a maximum length of 32 characters (33 including the null termination character)

input_metric_id

Pointer to an optional ID that is *unique* and that can be treated as an alias for the other metadata. It can be any value except all zeros or all ones. If the pointer is null, no ID is provided.

An ID is *unique* if the probability of the ID being associated with more than one set of metadata is vanishingly small. The selection of 128-bit IDs yields 3.4 x 10**38 unique IDs, so the objective is to select an ID that makes use of all 128 bits and is reasonably likely to not be selected by another person creating an ID of the same form. Two suggested algorithms that generate 128-bit values with these characteristics are:

1. The Universal Unique Identifier (UUID) algorithm that is part of The Open Group specification: *DCE 1.1: Remote Procedure Call*. A developer could use the algorithm at the time the application is developed using a utility on his or her system, and be reasonably certain that nobody else would generate the same 128-bit ID.

2. The MD5 Message-Digest Algorithm, described in IETF RFC 1321. Applying this algorithm to a concatenation of all the metadata properties would almost certainly result in a value that would not collide with any other ID created with a different set of metadata properties.

If an ARM implementation is passed an ID that was previously registered within this process, the implementation can ignore the other metadata parameters and assume they are identical to the previously registered metadata. The metric metadata consists of the following fields: *app_id*, *name*, *format*, *usage*, and *unit*.

name

Pointer to a null-terminated string containing the name of the metric. The maximum length of the string is 128 characters, including the termination character. It serves no purpose and is illegal to make this call if the pointer is null or the string is blank.

The name can be any string, with one exception. Strings beginning with the four characters "ARM:" are reserved for the ARM specification. The specification will define names with known semantics using this prefix. One name format is currently defined. Any name beginning with the eight character prefix "ARM:CIM:" represents a name defined using the DMTF CIM (Distributed Management Task Force Common Information Model) naming rules. For example, "ARM:CIM:CIM_SoftwareElement.Name" indicates that the metric value has the semantics of the *Name* property of the *CIM_SoftwareElement* class. It is anticipated that additional naming semantics are likely to be added in the future.

output_metric_id

Pointer to a 16-byte field aligned on an 8-byte boundary. ARM will store a 16-byte value. There are no requirements on the value it is set to, except that it must be possible to pass it in the metric binding sub-buffer on the *arm_register_transaction*() call, without the application needing to do any error checking.

unit Pointer to a null-terminated string containing the units (such as “files transferred”) of the metric. The maximum length of the string is 128 characters, including the termination character. The pointer may be null.

usage Describes how the metric is used. The value must be one of the following values, or a negative value (any negative value is specific to the application; the negative values are not expected to be widely used).

0 = ARM_METRIC_USE_GENERAL = a metric without a specified usage. Most metrics are described with a GENERAL usage.
1 = ARM_METRIC_USE_TRAN_SIZE = a metric that indicates the “size” of a transaction. The “size” is something that would be expected to affect the response time, such as the number of bytes in a file transfer or the number of files backed up by a “backup” transaction. ARM implementations can use this knowledge to better interpret the response time.
2 = ARM_METRIC_USE_TRAN_STATUS = a metric that further explains the transaction status passed on *arm_stop_transaction*(), such as a sense code that explains why a transaction failed.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_transaction(), *arm_stop_transaction*()

arm_register_transaction()

NAME

arm_register_transaction() – describe transaction

SYNOPSIS

```
arm_error_t
arm_register_transaction(
    /*[in]*/ const arm_id_t *app_id,
    /*[in]*/ const arm_char_t *tran_name,
    /*[opt in]*/ const arm_id_t *input_tran_id,
    /*[opt in]*/ const arm_int32_t flags,       /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4,
    /*[out]*/ arm_id_t *output_tran_id);
```

DESCRIPTION

arm_register_transaction() describes metadata about a transaction.

The application uses *arm_register_transaction*() to inform the ARM library of metadata about the transaction measured by the application. This metadata does not change from one application instance to another. It is the equivalent of the ARM 2.0 call *arm_getid*().

ARM generates an ID that is passed in *arm_start_transaction*() and *arm_report_transaction*().

PARAMETERS

app_id Application ID returned from an *arm_register_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are **arm_subbuffer_metric_bindings_t** and **arm_subbuffer_tran_identity_t**.

The names of any transaction context properties are supplied in the **arm_subbuffer_tran_identity_t** sub-buffer. They do not change afterwards; that is, the names are immutable. The transaction context values may change with each *arm_start_transaction*() or *arm_report_transaction*().

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

input_tran_id

Pointer to an optional ID that is *unique* and that can be treated as an alias for the other metadata. It can be any value except all zeros or all ones. If the pointer is null, no ID is provided.

An ID is *unique* if the probability of the ID being associated with more than one set of metadata is vanishingly small. The selection of 128-bit IDs yields 3.4 x 10**38 unique IDs, so the objective is to select an ID that makes use of all 128 bits and is reasonably likely to not be selected by another person creating an ID of the same form. Two suggested algorithms that generate 128-bit values with these

characteristics are:

1. The Universal Unique Identifier (UUID) algorithm that is part of The Open Group specification: *DCE 1.1: Remote Procedure Call*. A developer could use the algorithm at the time the application is developed using a utility on his or her system, and be reasonably certain that nobody else would generate the same 128-bit ID.

2. The MD5 Message-Digest Algorithm, described in IETF RFC 1321. Applying this algorithm to a concatenation of all the metadata properties would almost certainly result in a value that would not collide with any other ID created with a different set of metadata properties.

If an ARM implementation is passed an ID that was previously registered within this process, the implementation can ignore the other metadata parameters and assume they are identical to the previously registered metadata. The transaction metadata consists of the following fields: *app_id*, *tran_name*, and the **arm_subbuffer_metric_bindings_t** and **arm_subbuffer_tran_identity_t** sub-buffers passed in *buffer4*.

output_tran_id
: Pointer to a 16-byte field aligned on an 8-byte boundary. ARM will store a 16-byte value. There are no requirements on the value it is set to, except that it must be possible to pass it on other calls, such as *arm_start_transaction*(), without the application needing to do any error checking.

tran_name
: Pointer to a null-terminated string containing the name of the transaction. Each transaction registered by an application must have a unique name. The maximum length of the string is 128 characters, including the termination character. It serves no purpose and is illegal to make this call if the pointer is null or the string is blank.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_application(), *arm_report_transaction*(), *arm_start_transaction*()

arm_report_transaction()

NAME

arm_report_transaction() – report transaction statistics

SYNOPSIS

```
arm_error_t
arm_report_transaction(
    /*[in]*/ const arm_app_start_handle_t app_handle,
    /*[in]*/ const arm_id_t *tran_id,
    /*[in]*/ const arm_tran_status_t tran_status,
    /*[in]*/ const arm_response_time_t response_time,
    /*[in]*/ const arm_stop_time_t stop_time,
    /*[opt in]*/ const arm_correlator_t *parent_correlator,
    /*[opt in]*/ const arm_correlator_t *current_correlator,
    /*[opt in]*/ const arm_int32_t flags,
    /*[opt in]*/ const arm_buffer4_t *buffer4);
```

DESCRIPTION

arm_report_transaction() is used to report statistics about a transaction that has already completed.

arm_report_transaction() may be used instead of a paired *arm_start_transaction*() and *arm_stop_transaction*() call. The application would have measured the response time. The transaction could have executed on the local system or on a remote system. If it executes on a remote system, the system address sub-buffer passed on the *arm_start_application*() provides the addressing information for the remote system.

When used, it prevents certain types of proactive management, such as monitoring for hung transactions or adjusting priorities, because the ARM implementation is not invoked when the transaction is started. Because of this, the use of *arm_start_transction*() and *arm_stop_transaction*() is preferred over *arm_report_transaction*().

The intended use is to report status and response time about transactions that recently completed (typically several seconds ago) in the absence of an ARM-enabled application and/or ARM implementation on the system on which the transaction executed.

PARAMETERS

app_handle A handle returned in an *out* parameter from an *arm_start_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are **arm_subbuffer_diag_detail_t**, **arm_subbuffer_metric_values_t**, **arm_subbuffer_tran_context_t**, and **arm_subbuffer_user_t**.

current_correlator

Pointer to the correlator for the transaction that has completed, if any. If the pointer is null, there is no current correlator.

flags Contains 32-bit flags. In the least significant byte, ARM_FLAG_TRACE_REQUEST (0x00000001) is 1 if the application requests/suggests a trace.

parent_correlator
Pointer to a parent correlator, if any. If the pointer is null, there is no parent correlator.

response_time
Response time in nanoseconds.

stop_time An **int64** that contains the number of milliseconds since Jan 1, 1970 using the Gregorian calendar. The time base is GMT (Greenwich Mean Time). There is one special value, ARM_USE_CURRENT_TIME (–1), which means use the current time.

tran_id A transaction ID returned in an *out* parameter from an *arm_register_transaction*() call in the same process.

tran_status Indicates the status of the transaction. The following values are allowed:

0 = ARM_STATUS_GOOD = transaction completed successfully
1 = ARM_STATUS_ABORTED = transaction was aborted before it completed. For example, the user might have pressed "Stop" or "Back" on a browser while a transaction was in process, causing the transaction, as measured at the browser, to be aborted.
2 = ARM_STATUS_FAILED = transaction completed in error
3 = ARM_STATUS_UNKNOWN = transaction completed but the status was unknown. This would most likely occur if middleware or some other form of proxy instrumentation measured the transaction. This instrumentation may know enough to know when the transaction starts and stops, but does not understand the application-specific semantics sufficiently well to know whether the transaction was successful.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_transaction(), *arm_start_application*(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_start_application()

NAME

arm_start_application() – check application is running

SYNOPSIS

```
arm_error_t
arm_start_application(
    /*[in]*/ const arm_id_t *app_id,
    /*[opt in]*/ const arm_char_t *app_group,
    /*[opt in]*/ const arm_char_t *app_instance,
    /*[opt in]*/ const arm_int32_t flags,       /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4,
    /*[out]*/ arm_app_start_handle_t *app_handle);
```

DESCRIPTION

arm_start_application() indicates that an instance of an application has started running and is prepared to make ARM calls.

arm_start_application() indicates that an instance of an application has started running and is prepared to make ARM calls. In many cases, there will be only one application instance in a process, but there are cases in which there could be multiple instances. An example of multiple application instances in the same process is if several Java applications run in the same JVM (Java Virtual Machine) in the same process, and they each call the ARM 4.0 C interface (either directly, or indirectly via an implementation of the ARM 4.0 Java interface). They might share the same application ID or they might be separately registered.

Application context properties may be used to differentiate between instances. The values do not have to be different from other instances, though making them unique is suggested. The context properties are provided through function parameters and/or a sub-buffer.

The group and instance names are provided as function parameters.

Up to twenty *name*=*value* pairs of context properties may be provided in a sub-buffer.

There is a special case in which a system address sub-buffer is provided. The system address sub-buffer is provided when *arm_report_transaction*() will be used to report data about transactions that executed on a different system. In this case, the *arm_start_application*() provides a scoping context for the transaction instances, but does not indicate that the application instance is running on the local system.

The combination of *arm_register_application*() and *arm_start_application*() is equivalent to the ARM 2.0 call *arm_init*().

PARAMETERS

app_group Pointer to a null-terminated string containing the identity of a group of application instances, if any. A null pointer indicates that there is no group. The maximum length of the string is 256 (ARM_PROPERTY_VALUE_MAX_CHARS) characters, including the termination character. The pointer may have a null value.

app_handle Pointer to an 8-byte field aligned on an 8-byte boundary. There are no requirements on the value it is set to, except that it must be possible to pass it on other calls, such

as *arm_start_transaction*(), without the application needing to do any error checking. Whether the data is meaningful, or partially meaningful, is at the discretion of the ARM implementation.

app_id Pointer to a 16-byte ID returned by an *arm_register_application*() call.

app_instance
Pointer to a null-terminated string containing the identity of this instance of the application. It might contain the process ID, or a UUID in printable characters, for example. The maximum length of the string is 256 (ARM_PROPERTY_VALUE_MAX_CHARS) characters, including the termination character. The pointer may have a null value.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffer formats that might be used are **arm_subbuffer_app_context_t** and **arm_subbuffer_system_address_t**.

If no system sub-buffer is provided on *arm_start_application*(), all transactions reported by this application instance execute in the current process.

If a system sub-buffer is provided on *arm_start_application*(), all transactions execute in a different process.

If a system sub-buffer is provided in which the system address length is zero, or the system address pointer is null, the system is the “local” system, as determined by the ARM implementation.

If a system sub-buffer is provided in which there is a non-null system address and length, the system may be the local system or a remote system. Interpretation of what is local *versus* remote is at the discretion of the ARM implementation.

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_register_application(), *arm_report_transaction*(), *arm_start_transaction*()

arm_start_transaction()

NAME

arm_start_transaction() – start transaction

SYNOPSIS

```
arm_error_t
arm_start_transaction(
    /*[in]*/ const arm_app_start_handle_t app_handle,
    /*[in]*/ const arm_id_t *tran_id,
    /*[opt in]*/ const arm_correlator_t *parent_correlator,
    /*[opt in]*/ const arm_int32_t flags,
    /*[opt in]*/ const arm_buffer4_t *buffer4,
    /*[out]*/ arm_tran_start_handle_t *tran_handle,
    /*[out]*/ arm_correlator_t *current_correlator);
```

DESCRIPTION

arm_start_transaction() is used to indicate that a transaction is beginning execution.

Call *arm_start_transaction*() just prior to a transaction beginning execution. *arm_start_transaction*() signals the ARM library to start timing the transaction response time. There is one exception: when *arm_get_arrival_time*() is used to get a start time before *arm_start_transaction*() executes. See *arm_get_arrival_time*() to understand this usage.

arm_start_transaction() is also the means to pass to ARM the correlation token (called a "correlator" in ARM) from a caller – the "parent" – and to request a correlator that can be passed to transactions called by this transaction. The correlation token may be used to establish a calling hierarchy across processes and systems. A correlator contains a two-byte length field, a one-byte format ID, a one-byte flag field, plus it may contain other data that is used to uniquely identify an instance of a transaction. Applications do not need to understand correlator internals. The maximum length of a correlator is 512 (ARM_CORR_MAX_LENGTH) bytes. (In ARM 2.0 it was 168 bytes.)

PARAMETERS

app_handle A handle returned in an *out* parameter from an *arm_start_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are **arm_subbuffer_arrival_time_t**, **arm_subbuffer_metric_values_t**, **arm_subbuffer_tran_context_t**, and **arm_subbuffer_user_t**.

current_correlator

Pointer to a buffer into which the ARM implementation will store a correlator for the transaction instance, if any. The length of the buffer should be (at least) ARM_CORR_MAX_LENGTH.

If the pointer is null (ARM_CORR_NONE), the application is not requesting that a correlator be created.

If the pointer is not null, the application is requesting that a correlator be created. In this case the ARM implementation may (but need not) create a correlator in the buffer. It may not create a correlator if it does not support the function or if it is configured to not create a correlator.

After *arm_start_transaction*() completes, the application tests the length field using *arm_get_correlator_length*() to determine whether a correlator has been stored. If the length is still zero, no correlator has been stored. The ARM implementation must store zero in the length field if it does not generate a correlator.

flags Contains 32-bit flags. Three flags are defined:

ARM_FLAG_TRACE_REQUEST (0x00000001) is 1 if the application requests/suggests a trace.

ARM_FLAG_BIND_THREAD (0x00000002) is 1 if the started transaction is bound to this thread. Setting this flag is equivalent to immediately calling *arm_bind_thread*() after the *arm_start_transaction*() completes, except the timing is a little more accurate and the extra call is avoided.

ARM_FLAG_CORR_IN_PROCESS (0x00000004) is 1 if the application is stating that it will not send the correlator outside the current process. An ARM implementation may be able to optimize correlator handling if it knows this, because it may be able to avoid serialization to create the correlator.

parent_correlator
Pointer to the parent correlator, if any. The pointer may be null (ARM_CORR_NONE).

tran_handle Pointer to an **int64** into which the ARM library will store the value of the handle that will represent the transaction instance in all calls, up to and including the *arm_stop_transaction*() that indicates that the instance has completed executing. The scope of the handle is the process in which the *arm_start_transaction*() is executed.

There are no defined behaviors for the value returned – it is implementation-defined. Note that the returned value will always be a value that the application can use in future calls that take a handle [*arm_bind_thread*(), *arm_block_transaction*(), *arm_stop_transaction*(), *arm_unbind_thread*(), *arm_unblock_transaction*(), and *arm_update*()].

tran_id A transaction ID returned in an *out* parameter from an *arm_register_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is

returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_bind_thread(), *arm_block_transaction*(), *arm_get_arrival_time*(), *arm_get_correlator_length*(), *arm_register_transaction*(), *arm_start_application*(), *arm_stop_transaction*(), *arm_unbind_thread*(), *arm_unblock_transaction*(), *arm_update*()

arm_stop_application()

NAME

arm_stop_application() – stop application

SYNOPSIS

```
arm_error_t
arm_stop_application(
    /*[in]*/ const arm_app_start_handle_t app_handle,
    /*[opt in]*/ const arm_int32_t flags,         /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);   /*no current use*/
```

DESCRIPTION

arm_stop_application() indicates that the application instance has finished making ARM calls. It typically means that the instance is ending, such as just prior to the process exiting or a thread that represents an application instance terminating.

For any transactions that are still in-process [*arm_start_transaction*() executed without a matching *arm_stop_transaction*()], an implicit *arm_discard_transaction*() is executed.

If the *arm_start_application*() used the system address sub-buffer to indicate that the ARM calls would be about an application instance on a different system, *arm_stop_application*() indicates that no more calls about that application instance and its transactions will be made.

After executing *arm_stop_application*(), no further calls should be made for this application, including calls for transactions created by this application, until a new instance "session" is started using *arm_start_application*(). Data from any other calls that are made will be ignored. This function is the equivalent of the ARM 2.0 function *arm_end*().

PARAMETERS

app_handle The handle returned in an *out* parameter from an *arm_start_application*() call in the same process.

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_discard_transaction(), *arm_start_application*(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_stop_transaction()

NAME

arm_stop_transaction() – stop transaction

SYNOPSIS

```
arm_error_t
arm_stop_transaction(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[in]*/ const arm_tran_status_t tran_status,
    /*[opt in]*/ const arm_int32_t flags,       /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);
```

DESCRIPTION

arm_stop_transaction() signals the end of a transaction.

Call *arm_stop_transaction*() just after a transaction completes. *arm_start_transaction*() signals the ARM library to start timing the transaction response time. *arm_stop_transaction*() signals the ARM library to stop timing the transaction response time. It can be called from any thread in the process that executed the *arm_start_transaction*().

Implicit *arm_unbind_thread*() and *arm_unblock_transaction*() calls are made for any pending *arm_bind_thread*() and *arm_block_transaction*() calls, respectively, that have not been explicitly unbound or unblocked with *arm_unbind_thread*() and *arm_unblock_transaction*().

PARAMETERS

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are **arm_subbuffer_diag_detail_t** and **arm_subbuffer_metric_values_t**.

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

tran_status Indicates the status of the transaction. The following values are allowed:

0 = ARM_STATUS_GOOD = transaction completed successfully
1 = ARM_STATUS_ABORTED = transaction was aborted before it completed. For example, the user might have pressed “Stop” or “Back” on a browser while a transaction was in process, causing the transaction, as measured at the browser, to be aborted.
2 = ARM_STATUS_FAILED = transaction completed in error
3 = ARM_STATUS_UNKNOWN = transaction completed but the status was unknown. This would most likely occur if middleware or some other form of proxy instrumentation measured the transaction. This instrumentation may know enough to know when the transaction starts and stops, but does not understand the application-specific semantics sufficiently well to know whether the transaction was successful.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_bind_thread(), *arm_block_transaction*(), *arm_start_transaction*(), *arm_unbind_thread*(), *arm_unblock_transaction*()

arm_unbind_thread()

NAME

arm_unbind_thread() – unbind a thread

SYNOPSIS

```
arm_error_t
arm_unbind_thread(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[opt in]*/ const arm_int32_t flags,         /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);   /*no current use*/
```

DESCRIPTION

arm_unbind_thread() indicates that the thread from which it is called is no longer performing on behalf of the transaction identified by the start handle.

Call *arm_unbind_thread*() when a thread is no longer executing a transaction. The thread binding is useful for managing computing resources at a finer level of granularity than the process. It should be called when, for this transaction and this thread, either:

- *arm_bind_thread*() was previously called, or:
- The ARM_FLAG_BIND_THREAD flag was on in the *arm_start_transaction*() call.

arm_stop_transaction() is an implicit *arm_unbind_thread*() for any threads still bound to the transaction instance [*arm_bind_thread*() issued without a matching *arm_unbind_thread*()]. As long as the transaction is bound to the thread when the *arm_stop_transaction*() executes, there is no need nor any value in calling *arm_unbind_thread*() before calling *arm_stop_transaction*().

PARAMETERS

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_bind_thread(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_unblock_transaction()

NAME

arm_unblock_transaction() – unblock transaction

SYNOPSIS

```
arm_error_t
arm_unblock_transaction(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[in]*/ const arm_tran_block_handle_t block_handle,
    /*[opt in]*/ const arm_int32_t flags,          /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);    /*no current use*/
```

DESCRIPTION

arm_unblock_transaction() indicates that the suspension indicated by the *block_handle* for the transaction identified by the start handle is no longer waiting for a downstream transaction to complete.

Call *arm_unblock_transaction*() when a transaction is no longer blocked on an external event. It should be called when *arm_block_transaction*() was previously called and the blocking condition no longer exists. Knowledge of when a transaction is blocked can be useful for better understanding response times. It is useful to separate out this “blocked” time from the elapsed start-to-stop time. The unblocked call is paired with a block call for finer grained analysis.

arm_stop_transaction() is an implicit *arm_unblock_transaction*() for any blocking condition for the transaction instance that has not been cleared yet [*arm_block_transaction*() issued without a matching *arm_unblock_transaction*()]. It should only be called without calling *arm_unblock_transaction*() first when the blocking condition ends immediately prior to the transaction ending.

PARAMETERS

block_handle
: A handle returned in an *out* parameter from an *arm_block_transaction*() call in the same process.

buffer4
: Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. No sub-buffer types are currently valid with this function call, so the pointer should be null (ARM_BUF4_NONE).

flags
: Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle
: A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is

returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_block_transaction(), *arm_start_transaction*(), *arm_stop_transaction*()

arm_update_transaction()

NAME

arm_update_transaction() – get transaction status

SYNOPSIS

```
arm_error_t
arm_update_transaction(
    /*[in]*/ const arm_tran_start_handle_t tran_handle,
    /*[opt in]*/ const arm_int32_t flags,         /*no current use*/
    /*[opt in]*/ const arm_buffer4_t *buffer4);
```

DESCRIPTION

arm_update_transaction() signals that a transaction is still processing.

arm_update_transaction() is useful as a heartbeat. It is also used to pass additional data about a transaction. It can be called from any thread in the process that executed the *arm_start_transaction*().

PARAMETERS

buffer4 Pointer to the user data buffer, if any. If the pointer is null, there is no buffer. The sub-buffer that might be used is **arm_subbuffer_metric_values_t**.

flags Contains 32-bit flags. No values are currently defined. The field should be zero (ARM_FLAG_NONE).

tran_handle A handle returned in an *out* parameter from an *arm_start_transaction*() call in the same process.

RETURN VALUE

The returned code is a status that may indicate if an error was detected.

ERRORS

If the return code is negative, an error occurred. If the return code is not negative, an error may or may not have occurred – the determination of what is an error and whether an error code is returned is at the discretion of the ARM implementation. The application can test the return code if it wants to provide its own error logging.

SEE ALSO

arm_start_transaction()

9 Optional Buffer and Sub-Buffers

9.1 User Data Buffer

(**buffer4* in all calls)

Syntax

```
typedef struct arm_buffer4
{
    arm_int32_t count;
    arm_subbuffer_t **subbuffer_array;
} arm_buffer4_t;

typedef struct arm_subbuffer {
    arm_subbuffer_format_t format;
    /* format-specific data fields are following here */
} arm_subbuffer_t;
```

Description

Many of the ARM function calls provide a way for the application and ARM implementation to exchange optional data, in addition to the required data in other function parameters. This buffer describes the format of the exchanged data.

Almost all functions define a **buffer4* parameter. When the value is not null, the value points to a buffer in the following format. It differs from the ARM 2.0 format in that the buffer contains pointers to sub-buffers, rather than inline contiguous data. The sub-buffers contain the meaningful data.

Each sub-buffer may be in the user data buffer once. If there is more than one instance of a sub-buffer in the buffer, all instances after the first will be ignored.

The buffer is aligned on a pointer boundary for the platform. The byte layout depends on the platform.

Format

(See Figure 15.)

- **Count of sub-buffer pointers:** An **int32** count of sub-buffers in the following array.
- **Array of pointers to sub-buffers:** A pointer to an array of sub-buffers. The array is aligned on a pointer boundary for the platform. A null pointer indicates that this element in the array is not used on this call; later elements in the array may be non-null.

Each sub-buffer is in the following format:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID.

  Known sub-buffer formats (all unassigned values in the non-negative range are reserved):

  ```
  3 = ARM_SUBBUFFER_USER = user
  4 = ARM_SUBBUFFER_ARRIVAL_TIME = arrival time
  5 = ARM_SUBBUFFER_METRIC_VALUES = metric values
  6 = ARM_SUBBUFFER_SYSTEM_ADDRESS = system
  7 = ARM_SUBBUFFER_DIAG_DETAIL = diagnostic details
  102 = ARM_SUBBUFFER_APP_IDENTITY = application identity properties
  103 = ARM_SUBBUFFER_APP_CONTEXT = application context properties
  104 = ARM_SUBBUFFER_TRAN_IDENTITY = transaction identity
        properties
  105 = ARM_SUBBUFFER_TRAN_CONTEXT = transaction context properties
  106 = ARM_SUBBUFFER_METRIC_BINDINGS = metric bindings
  107 = ARM_SUBBUFFER_CHARSET = character set encoding
  ```

  All negative values are available for implementation-specific purposes.

- **Other sub-buffer data:** There are two common patterns, each illustrated below.

  — The sub-buffer does not contain an array of elements. In this case, the data is inline immediately (subject to byte alignment requirements for the data type on this platform) following the format ID. See Figure 18.

  — The sub-buffer contains one or more arrays of elements. In this case, the sub-buffer is in the following format. See Figure 19.

    — Sub-buffer format ID

    — Non-array data, if any

    — One or more sets of array count/pointer pairs

    — Count of array elements

    — Pointer to an array of elements

    — Other non-array data, if any

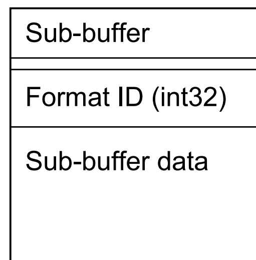

Figure 18: Format of Sub-Buffer that does not Contain an Array

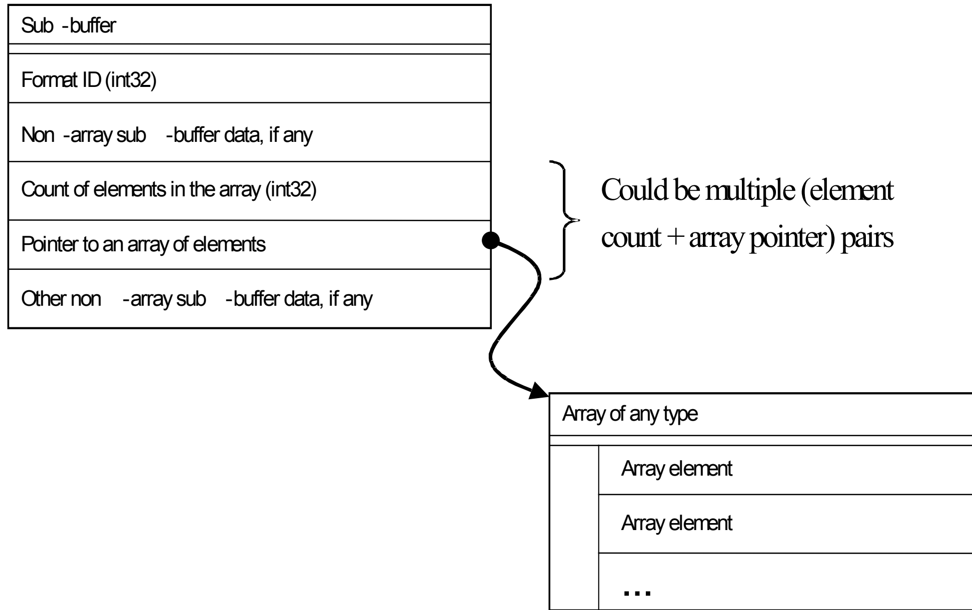

Figure 19: Format of Sub-Buffer that Contains One or More Arrays of Data

9.2 User Sub-Buffer

(*format*=3, ARM_SUBBUFFER_USER)

Syntax

```
typedef struct arm_subbuffer_user
{
    arm_subbuffer_t header;
    const arm_char_t *name;
    arm_boolean_t id_valid;
    arm_id_t id;
} arm_subbuffer_user_t;
```

Description

A user name and/or ID may be optionally associated with each transaction instance. A name is a null-terminated character string. An ID is a 16-byte binary value, such as a UUID. Either or both may be provided.

Format

The pattern is illustrated in Figure 18:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=3.
- **User name:** A null-terminated character string with a maximum length of 128 characters, including the termination character. A null value indicates no name is provided.
- **ID valid:** A boolean indicating whether the ID field contains a valid ID.
- **ID:** 16-byte ID, aligned on an 8-byte boundary, that is associated with and can be used as an alias for the user name.

9.3 Arrival Time Sub-Buffer

(*format*=4, ARM_SUBBUFFER_ARRIVAL_TIME)

Syntax

```
typedef struct arm_subbuffer_arrival_time
{
    arm_subbuffer_t header;
    arm_arrival_time_t opaque_time;
} arm_subbuffer_arrival_time_t;
```

Description

Some applications may start processing a transaction before all the context information that identifies the transaction is known. For example, it might by necessary to retrieve the context information as the first step in processing the transaction. For these cases, the application can call *arm_get_arrival_time*() to receive a timestamp value for the current time from the ARM implementation. This timestamp value is known as the "arrival time". When the transaction context data is all known, *arm_start_transaction*() is called, passing the optional arrival time value in a sub-buffer, to indicate when the transaction actually started executing. The arrival time is provided in this sub-buffer.

The arrival time field is a 64-bit integer, aligned on a 64-bit boundary, containing the value returned by the *arm_get_arrival_time*() function. The format of the data within the **int64** is implementation-defined.

Format

The pattern is illustrated in Figure 18:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=4.
- **Arrival time:** An **int64** value, aligned on an 8-byte boundary, containing an opaque time indicator generated by and recognizable by the ARM implementation.

9.4 Metric Values Sub-Buffer

(*format*=5, ARM_SUBBUFFER_METRIC_VALUES)

Syntax

```
typedef struct arm_subbuffer_metric_values
{
    arm_subbuffer_t header;
    arm_int32_t count;
    const arm_metric_t *metric_value_array;
} arm_subbuffer_metric_values_t;

typedef struct arm_metric
{
    arm_metric_slot_t slot;
    arm_metric_format_t format;
    arm_metric_usage_t usage;
    arm_boolean_t valid;
    union
    {
        arm_metric_counter32_t counter32;
        arm_metric_counter64_t counter64;
        arm_metric_cntrdivr32_t counterdivisor32;
        arm_metric_gauge32_t gauge32;
        arm_metric_gauge64_t gauge64;
        arm_metric_gaugedivr32_t gaugedivisor32;
        arm_metric_numericID32_t numericid32;
        arm_metric_numericID64_t numericid64;
        arm_metric_string32_t string32;
    } metric_u;
} arm_metric_t;
```

Description

The buffer is used to pass metric values on any of *arm_start_transaction*(), *arm_update_transaction*(), *arm_stop_transaction*(), and *arm_report_transaction*().

Format

The pattern is illustrated in Figure 19:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=5.
- **Count of metric value pointers:** An **int32** count of metric value pointers.
- **Pointer to an array of metric value structures:** The number of pointers in the array is specified by the previous *count* value.

Each structure is in the following format, and is aligned as required for the C compiler on the platform:

- **Slot number:** A single-byte slot number. Valid values are 0 to 6. The slot number must be the same as the corresponding entry in the metric bindings sub-buffer. Each slot number should be used at most once; if a slot number is reused, the first entry is used and all others are ignored.
- **Metric format:** A single-byte format indicator. Valid values are 1 to 10 and are the same as ARM 2.0. Only values 1 to 8 are valid in slots 0-5. Only value 10 is valid in slot 6. This is a carry-over from ARM 2.0. The format must be the same as the corresponding entry in the metric bindings sub-buffer.

```
1 = ARM_METRIC_FORMAT_COUNTER32 = int32 counter
2 = ARM_METRIC_FORMAT_COUNTER64 int64 counter
3 = ARM_METRIC_FORMAT_CNTRDIVR = int32 counter + int32 divisor
4 = ARM_METRIC_FORMAT_GAUGE32 = int32 gauge
5 = ARM_METRIC_FORMAT_GAUGE64 = int64 gauge
6 = ARM_METRIC_FORMAT_GAUGEDIVR32 = int32 gauge + int32 divisor
7 = ARM_METRIC_FORMAT_NUMERICID32 = int32 numeric ID
8 = ARM_METRIC_FORMAT_NUMERICID64 = int64 numeric ID
9 = (DEPRECATED)
10 = ARM_METRIC_FORMAT_STRING32 = string (null-terminated) of a
maximum length of 32 characters (33 including the null
termination character)
```

- **Usage:** An **int16** indicating how the metric is used. The usage indicator must be the same as the corresponding entry in the metric bindings sub-buffer.
  - 0 = ARM_METRIC_USE_GENERAL = no usage is declared
  - 1 = ARM_METRIC_USE_TRAN_SIZE = the metric indicates the transaction size (e.g., the size of a file or the number of jobs in a network backup operation)
  - 2 = ARM_METRIC_USE_TRAN_STATUS = the metric is a status code (numeric ID) or text message (string). It would typically be used with *arm_stop_transaction*() or *arm_report_transaction*() to provide additional details about a transaction status of Failed.
  - 3:32767 = Reserved.
  - -32768:-1 = available for implementation-defined purposes.
- **Valid flag:** A boolean that indicates whether the data in the metric is currently valid.
- **Metric value:** A C union containing the metric value. The data type matches the metric format indicator (above).

```
1 = ARM_METRIC_FORMAT_COUNTER32 = int32 counter
2 = ARM_METRIC_FORMAT_COUNTER64 int64 counter
3 = ARM_METRIC_FORMAT_CNTRDIVR = int32 counter + int32 divisor
4 = ARM_METRIC_FORMAT_GAUGE32 = int32 gauge
5 = ARM_METRIC_FORMAT_GAUGE64 = int64 gauge
6 = ARM_METRIC_FORMAT_GAUGEDIVR32 = int32 gauge + int32 divisor
7 = ARM_METRIC_FORMAT_NUMERICID32 = int32 numeric ID
8 = ARM_METRIC_FORMAT_NUMERICID64 = int64 numeric ID
```

```
9 = (DEPRECATED)
10 = ARM_METRIC_FORMAT_STRING32 = string (null-terminated) of a
maximum length of 32 characters (33 including the null
termination character)
```

9.5 System Address Sub-Buffer

(*format*=6, ARM_SUBBUFFER_SYSTEM_ADDRESS)

Syntax

```
typedef struct arm_subbuffer_system_address
{
    arm_subbuffer_t header;
    arm_int16_t address_format;
    arm_int16_t address_length;
    const arm_uint8_t *address;
    arm_boolean_t id_valid;
    arm_id_t id;
} arm_subbuffer_system_address_t;
```

Description

The system address sub-buffer is used with *arm_start_application*() when the transactions that will be reported execute on a different system than the one on which they will be reported.

- If no system sub-buffer is provided on *arm_start_application*(), all transactions reported by this application instance execute in the current process.
- If a system sub-buffer is provided on *arm_start_application*(), all transactions execute in a different process.
- If a system sub-buffer is provided in which the system address length is zero, or the system address pointer is null, the system is the "local" system, as determined by the ARM implementation.
- If a system sub-buffer is provided in which there is a non-null system address and length, the system may be the local system or a remote system. Interpretation of what is local *versus* remote is at the discretion of the ARM implementation.

Format

The pattern is illustrated in Figure 18:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=6.
- **Format of the system address:** An **int16** format of the system address field. The following formats are defined:
  - — 0 = Reserved.
  - — 1 = ARM_SYSADDR_FORMAT_IPV4 = IPv4

    Bytes 0:3 = 4-byte IP address

— 2 = ARM_SYSADDR_FORMAT_IPV4PORT = IPv4 + port number

  Bytes 0:3 = 4-byte IP address
  Bytes 4:5 = 2-byte IP port number

— 3 = ARM_SYSADDR_FORMAT_IPV6 = IPv6

  Bytes 0:15 = 16-byte IP address

— 4 = ARM_SYSADDR_FORMAT_IPV6PORT = IPv6 + port number

  Bytes 0:15 = 16-byte IP address
  Bytes 16:17 = 2-byte IP port number

— 5 = ARM_SYSADDR_FORMAT_SNA = SNA

  Bytes 0:7 = EBCDIC-encoded network ID
  Bytes 8:15 = EBCDIC-encoded network accessible unit (control point or LU)

— 6 = ARM_SYSADDR_FORMAT_X25 = X.25

  Bytes 0:15 = The X.25 address (also referred to as an X.121 address). This is up to 16 ASCII character digits ranging from 0-9.

— 7 = ARM_SYSADDR_FORMAT_HOSTNAME = hostname (characters, not null-terminated)

  Bytes 0:?? = hostname

— 8 = ARM_SYSADDR_FORMAT_UUID = Universally-unique ID

  Bytes 0:15 = UUID in binary. This is useful for applications that define their system by a UUID rather than a network address or hostname or some other address form.

— 9:32767 = Reserved.

— –32768: –1: = Available for implementation-defined 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 deemed small.

- **Length of system address:** An **int16** length of system address in bytes.
  - There is no maximum length.
  - A length of zero refers to the local system.
- **System address:** A byte array containing the system address.

- The byte array is the length specified by the “Length of system address” field. Note that it could have a length of zero bytes, or be a null pointer, indicating that it is the local system.

- **ID valid:** A boolean indicating whether the system definition ID field contains a valid ID.
- **System address ID:** A 16-byte character array containing an ID that can be used as a synonym for this system address.
  - The ID is an optional identifier that is mapped to the other fields. If the value is all zeros, the ID is not being provided.

9.6 Diagnostic Detail Sub-Buffer

(*format*=7, ARM_SUBBUFFER_DIAG_DETAIL)

Syntax

```
typedef struct arm_subbuffer_diag_detail
{
    arm_subbuffer_t header;
    const arm_char_t *diag_detail;
} arm_subbuffer_diag_detail_t;
```

Description

When a transaction completion is reported with *arm_stop_transaction*() or *arm_report_transaction*(), and the transaction status is not ARM_STATUS_GOOD, the application may provide a character string containing any arbitrary diagnostic data. The string may not be longer than 4096 characters, including the null-termination character. For example, the application might provide the SQL query text for a failing database transaction.

Format

The pattern is illustrated in Figure 18:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=7.
- **Value:** A **char*** to a null-terminated string containing the diagnostic data. Each string has a maximum length of 4096 characters, including the termination character. If the pointer is null, it is equivalent to not providing the sub-buffer at all.

9.7 Application Identity Sub-Buffer

(*format*=102, ARM_SUBBUFFER_APP_IDENTITY)

Syntax

```
typedef struct arm_subbuffer_app_identity
{
    arm_subbuffer_t header;
    arm_int32_t identity_property_count;
    const arm_property_t *identity_property_array;
    arm_int32_t context_name_count;
    const arm_char_t **context_name_array;
} arm_subbuffer_app_identity_t;

typedef struct arm_property
{
    const arm_char_t *name;
    const arm_char_t *value;
} arm_property_t;
```

Description

Applications are identified by a name and an optional set of identity attribute *name*=*value* pairs. Application instances are further identified by an optional set of context *name*=*value* pairs. The optional context property names are provided in this sub-buffer on the *arm_register_application*() call. The optional context property values are provided on the *arm_start_application*() call. The sub-buffer is ignored if it is passed on any other call.

The names of identity and context properties can be any string, with one exception. Strings beginning with the four characters "ARM:" are reserved for the ARM specification. The specification will define names with known semantics using this prefix. One name format is currently defined. Any name beginning with the eight-character prefix "ARM:CIM:" represents a name defined using the DMTF CIM (Distributed Management Task Force Common Information Model) naming rules. For example, "ARM:CIM:CIM_SoftwareElement.Name" indicates that the property value has the semantics of the *Name* property of the *CIM_SoftwareElement* class. It is anticipated that additional naming semantics are likely to be added in the future.

Format

The pattern is illustrated in Figure 19:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=102.
- **Count of property values:** An **int32** count of property values in the following array.
- **Pointer to an array of identity properties:** An array of C structures containing the property names and values.

The array index of a property value in the value array is bound to the property name at the same index in the name array. Moving the (*name*,*value*) pair to a different index does not affect the identity of the application. For example, if an application registers once with a name *A* and a value *X* in array indices 0 and once with the same name and value in array indices 1, the registered identity has not changed.

If a name is repeated in the array, the name and its corresponding value are ignored, and the first instance of the name and value in the array is used. The implementation may return an error code but is not obliged to do so.

Each structure is aligned as required for the C compiler on the platform. Each structure contains:

— **Name:** A **char*** to a null-terminated string representing the *name* part of the *name*=*value* pair. Each string has a maximum length of 128 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored.

— **Value:** A **char*** to a null-terminated string representing the *value* part of the *name*=*value* pair. Each string has a maximum length of 256 characters, including the termination character. If any pointer is null, the *name*=*value* pair is ignored.

- **Pointer to an array of context property names:** An array of character pointers to the context property names.

  If a name is repeated in the array, the name and its corresponding value (in the application context sub-buffer) are ignored, and the first instance of the name in the array (and its corresponding value) is used. The implementation may return an error code but is not obliged to do so.

  Each pointer in the array is aligned as required for the C compiler on the platform. Each array element contains:

  — **Name:** A **char*** to a null-terminated string representing the *name* part of the *name*=*value* pair. Each string has a maximum length of 128 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored. The values are provided in the application context values sub-buffer.

9.8 Application Context Values Sub-Buffer

(*format*=103, ARM_SUBBUFFER_APP_CONTEXT)

Syntax

```
typedef struct arm_subbuffer_app_context
{
    arm_subbuffer_t header;
    arm_int32_t context_value_count;
    const arm_char_t **context_value_array;
} arm_subbuffer_app_context_t;
```

Description

Applications are identified by a name and an optional set of identity attribute *name*=*value* pairs. Application instances are further identified by an optional set of context *name*=*value* pairs. These properties could indicate something about the runtime instance of the application, such as the instance identifier and the name of a configuration file used.

The optional context property names are provided in the application identity sub-buffer on the *arm_register_application*() call. The optional context property values are provided in this sub-buffer on the *arm_start_application*() call. The sub-buffer is ignored if it is passed on any other call.

Format

The pattern is illustrated in Figure 19:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=103.
- **Count of property values:** An **int32** count of property values in the following array.
- **Pointer to an array of property values:** An array of pointers to character strings containing the property values. The values in the array are position-sensitive; each must align with the corresponding context property name in the application identity sub-buffer. Each pointer is aligned as required for the C compiler on the platform. Each array element contains:
    - — **Value:** A **char*** to a null-terminated string representing the *value* part of the *name*=*value* pair. Each string has a maximum length of 256 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored.

9.9 Transaction Identity Sub-Buffer

(*format*=104, ARM_SUBBUFFER_TRAN_IDENTITY)

Syntax

```
typedef struct arm_subbuffer_tran_identity
{
    arm_subbuffer_t header;
    arm_int32_t identity_property_count;
    const arm_property_t *identity_property_array;
    arm_int32_t context_name_count;
    const arm_char_t **context_name_array;
    const arm_char_t *uri;
} arm_subbuffer_tran_identity_t;

typedef struct arm_property
{
    const arm_char_t *name;
    const arm_char_t *value;
} arm_property_t;
```

Description

Transactions are identified by a name, an optional URI value, and an optional set of attribute *name*=*value* pairs. The URI and optional *name*=*value* pairs are provided in this sub-buffer on the *arm_register_transaction*() call. The sub-buffer is ignored if it is passed on any other call. The identity is scoped to a single application.

The names of identity and context properties can be any string, with one exception. Strings beginning with the four characters "ARM:" are reserved for the ARM specification. The specification will define names with known semantics using this prefix. One name format is currently defined. Any name beginning with the eight-character prefix "ARM:CIM:" represents a name defined using the DMTF CIM (Distributed Management Task Force Common Information Model) naming rules. For example, "ARM:CIM:CIM_SoftwareElement.Name" indicates that the property value has the semantics of the *Name* property of the *CIM_SoftwareElement* class. It is anticipated that additional naming semantics are likely to be added in the future.

Format

The pattern is illusttrated in Figure 19:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=104.
- **Count of elements in the identity property names and values array:** An **int32** count of elements in the following array.

- **Pointer to an array of identity property names and values:** An array of C structures containing the property names and values.

  The array index of a property value in the value array is bound to the property name at the same index in the name array. Moving the (*name*,*value*) pair to a different index does not affect the identity of the transaction. For example, if an application is registered once with a name *A* and a value *X* in array indices 0 and once with the same name and value in array indices 1, the registered identity has not changed.

  If a name is repeated in the array, the name and its corresponding value (in the transaction context sub-buffer) are ignored, and the first instance of the name in the array (and its corresponding value) is used. The implementation may return an error code but is not obliged to do so.

  Each structure is aligned as required for the C compiler on the platform. The pointer may be null. Each structure contains:

  — **Name:** A **char*** to a null-terminated string representing the *name* part of the *name*=*value* pair. Each string has a maximum length of 128 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored.

  — **Value:** A **char*** to a null-terminated string representing the *value* part of the *name*=*value* pair. Each string has a maximum length of 256 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored.

- **Count of elements in the context property names array:** An **int32** count of elements in the following array.

- **Pointer to an array of context property names:** An array of strings, each containing a context property name. If any pointer is null or the string is the null string, the name is ignored. Each array element is a **char*** to a null-terminated string representing the *name* part of the *name*=*value* pair. Each string has a maximum length of 128 characters, including the termination character. The name is passed on *arm_register_transaction*(). If any pointer is null when *arm_register_transaction*() executes, the corresponding value is ignored on all future calls using the context property values sub-buffer.

  If a name is repeated in the array, the name and its corresponding value (in the transaction context sub-buffer) are ignored, and the first instance of the name in the array (and its corresponding value) is used. The implementation may return an error code but is not obliged to do so.

  Each pointer in the array is aligned as required for the C compiler on the platform. Each array element contains:

  — **Name:** A **char*** to a null-terminated string representing the *name* part of the *name*=*value* pair. Each string has a maximum length of 128 characters, including the termination character. If any pointer is null or the string is the null string, the *name*=*value* pair is ignored.

- **Pointer to URI:** Pointer to a string containing a URI, with a maximum of 4096 characters, including the termination character. The string is null terminated. The pointer may be null.

9.10 Transaction Context Sub-Buffer

(*format*=105, ARM_SUBBUFFER_TRAN_CONTEXT)

Syntax

```
typedef struct arm_subbuffer_tran_context
{
    arm_subbuffer_t header;
    arm_int32_t context_value_count;
    const arm_char_t **context_value_array;
    const arm_char_t *uri;
} arm_subbuffer_tran_context_t;
```

Description

In addition to the identity properties, a transaction may be described with additional context properties. Context properties differ from identity properties in that although the name is provided when the transaction is registered [*arm_register_transaction*()], the values are provided when a transaction is measured [*arm_start_transaction*() or *arm_report_transaction*()]. The value of an identity property never changes, whereas the value of a context property may change every time a transaction executes.

Context properties are of two forms. There may be one URI value and up to twenty *name*=*value* pairs. No name may duplicate the name of a transaction identity property.

Format

The pattern is illustrated in Figure 19:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=105.
- **Count of elements in the context property values array:** An **int32** count of elements in the following array.
- **Pointer to an array of context property values:** An array of strings, each containing a context property value. The pointer may be null. Each array element is a **char*** to a null-terminated string representing the *value* part of the *name*=*value* pair. Each string has a maximum length of 256 characters, including the termination character. The values in the array are position-sensitive; they must align with the context property names in the transaction identity sub-buffer. When *arm_start_transaction*() or *arm_report_transaction*() execute, any pointer may be null, which indicates that no value is provided at this time. If the name passed to *arm_register_transaction*() was null, this value is ignored.
- **Pointer to URI:** Pointer to a string containing a URI, with a maximum of 4096 characters, including the termination character. The string is null terminated. The pointer may be null.

If a URI is provided in both the transaction identity sub-buffer and in the transaction context sub-buffer, the URI in the transaction identity sub-buffer must be the same as the URI provided in the transaction context sub-buffer, or a truncated subset.

9.11 Metric Bindings Sub-Buffer

(*format*=106, ARM_SUBBUFFER_METRIC_BINDINGS)

Syntax

```
typedef struct arm_subbuffer_metric_bindings
{
    arm_subbuffer_t header;
    arm_int32_t count;
    const arm_metric_binding_t *metric_binding_array;
} arm_subbuffer_metric_bindings_t;

typedef struct arm_metric_binding
{
    arm_metric_slot_t slot;
    arm_id_t id;
} arm_metric_binding_t;
```

Description

Applications may provide additional data about a transaction when the transaction starts, while it is executing, and/or after it has stopped. This additional data may serve several purposes, such as indicating the size of a transaction (e.g., the number of bytes in a file for a file transfer transaction), the state of the system (e.g., the number of queued up transactions when this transaction arrived), or an error code. The metadata describing each metric is provided with the *arm_register_metric*() function.

Each transaction definition may define zero to seven metrics for which data values may be provided on *arm_start_transaction*(), *arm_update_transaction*(), *arm_stop_transaction*(), or *arm_report_transaction*(). Each metric is assigned to an array slot numbered 0 to 6 (they were numbered 1 to 7 in ARM 2.0). This sub-buffer is passed on the *arm_register_transaction*() function to indicate which metrics are assigned to which slots.

The combination of this sub-buffer plus the *arm_register_metric*() function replaces ARM 2.0 *format* = 101. Unlike ARM 2.0, the metric definitions do not influence the transaction identity. Any properties besides the transaction name that affect identity are provided in the transaction identity properties (*format*=104) sub-buffer.

Format

The pattern is illustrated in Figure 19.

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=106.
- **Count of metric bindings:** An **int32** count of metric values in the following array.

- **Pointer to an array of metric bindings:** An array of C structures containing the metric bindings. Each structure is aligned as required for the C compiler on the platform. Each structure contains:
  - **Slot number:** A byte slot number. Valid values are 0 to 6 (replacing the ARM 2.0 numbering standard). If a slot number is repeated, the first time it appears is the only one processed; all others are ignored.
  - **ID:** A 16-byte array for the ID of this metric definition. The ID must have been previously returned as an *out* parameter from *arm_register_metric*().

9.12 Character Set Encoding Sub-Buffer

(*format*=107, ARM_SUBBUFFER_CHARSET)

Syntax

```
typedef struct arm_subbuffer_charset
{
    arm_subbuffer_t header;
    arm_charset_t charset;  /* one of the IANA MIBenum values */
    arm_int32_t   flags;
} arm_subbuffer_charset_t;
```

Description

Applications may specify on *arm_register_application*() the character set encoding for all strings passed by the application. An ARM library must support certain encodings, depending on the platform (see Table 1). The application may always use one of the mandatory encodings. The **<arm4.h>** file defines names of the form ARM_CHARSET_* for all the mandatory encodings. The application may optionally ask the ARM library if another encoding is supported using *arm_is_charset_supported*().

When the application registers with *arm_register_application*(), it may optionally provide the character set encoding sub-buffer. In the sub-buffer, the application specifies the MIBenum value of a character set encoding that has been assigned by the IANA (Internet Assigned Numbers Authority – see www.iana.org). The MIBenum value in the sub-buffer should only be either a mandatory encoding for the platform or a MIBenum value for which support has been verified using *arm_is_charset_supported*().

The list of supported encodings must restrict itself to those that do not contain any embedded null bytes. Exceptions are permitted for character sets that have fixed-length characters (e.g., two bytes) and that do not allow a character of all zeros (e.g., 0x0000). These include UTF-16LE, UTF-16BE, and UTF-16 (MIBenum values 1013, 1014, 1015). For these encodings, there will be a convention that a character of all zeros is the null-termination character.

For any encodings that are of a fixed size greater than one byte, such as UTF-16 (all characters are two bytes in length), the characters must be aligned on the natural boundary for the system (e.g., two-byte characters on a 16-bit boundary, etc.).

If an ARM implementation supports characters that are longer than one byte, the application and implementation are both responsible to cast (virtually, at least) the characters to wide characters, but the function signature does not change. For example, if UTF-16 is supported, the application must write pairs of characters into the **char*** array (there would always be an even number of characters), and the implementation must process the data in pairs (looking for 0x0000 as a termination character rather than 0x00, for example).

Format

The pattern is illustrated in Figure 18:

- **Sub-buffer format ID:** An **int32** sub-buffer format ID, *value*=107.
- **Character set ID (MIBenum):** An **int32** containing a MIBenum value assigned by the IANA. Some common encodings are listed in Table 1.
- **Flags:** An **int32** containing bit flags. The field is currently reserved for future use.

10 <arm4.h> Header File for Compiling

```
/* ---------------------------------------------------------------- */
/* arm4.h - ARM4 standard header file                               */
/*                                                                  */
/* This header file defines all defines, typedefs, structures,      */
/* and API functions visible for an application which uses an ARM  */
/* agent. All compiler/platform specifics are handled in a          */
/* separate header file named <arm4os.h>.                           */
/*                                                                  */
/* NOTE: The ARM4 C language binding differs completely from        */
/* ARM1 and ARM2 bindings.                                          */
/* ---------------------------------------------------------------- */

#ifndef ARM4_H_INCLUDED
#define ARM4_H_INCLUDED

#ifndef ARM4OS_H_INCLUDED
#include "arm4os.h"
#endif /* ARM4OS_H_INCLUDED */

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

/* ---------------------------------------------------------------- */
/* ---------------------- defines section ------------------------ */
/* ---------------------------------------------------------------- */

/* Boolean values */
#define ARM_FALSE                                   0
#define ARM_TRUE                                    1

/* Transaction status */
#define ARM_STATUS_GOOD                             0
#define ARM_STATUS_ABORTED                          1
#define ARM_STATUS_FAILED                           2
#define ARM_STATUS_UNKNOWN                          3

/* -------------- reserved error codes range --------------------- */

#define ARM_ERROR_CODE_RESERVED_MIN            -20999
#define ARM_ERROR_CODE_RESERVED_MAX            -20000

/* --------------- known sub-buffer formats ---------------------- */

#define ARM_SUBBUFFER_USER                          3
#define ARM_SUBBUFFER_ARRIVAL_TIME                  4
#define ARM_SUBBUFFER_METRIC_VALUES                 5
```

```
#define ARM_SUBBUFFER_SYSTEM_ADDRESS             6
#define ARM_SUBBUFFER_DIAG_DETAIL                7

#define ARM_SUBBUFFER_APP_IDENTITY             102
#define ARM_SUBBUFFER_APP_CONTEXT              103
#define ARM_SUBBUFFER_TRAN_IDENTITY            104
#define ARM_SUBBUFFER_TRAN_CONTEXT             105
#define ARM_SUBBUFFER_METRIC_BINDINGS          106
#define ARM_SUBBUFFER_CHARSET                  107

/* -------------------- metric defines --------------------------- */

#define ARM_METRIC_FORMAT_RESERVED               0
#define ARM_METRIC_FORMAT_COUNTER32              1
#define ARM_METRIC_FORMAT_COUNTER64              2
#define ARM_METRIC_FORMAT_CNTRDIVR32             3
#define ARM_METRIC_FORMAT_GAUGE32                4
#define ARM_METRIC_FORMAT_GAUGE64                5
#define ARM_METRIC_FORMAT_GAUGEDIVR32            6
#define ARM_METRIC_FORMAT_NUMERICID32            7
#define ARM_METRIC_FORMAT_NUMERICID64            8
/* format 9 (string8) is deprecated */
#define ARM_METRIC_FORMAT_STRING32              10

#define ARM_METRIC_USE_GENERAL                   0
#define ARM_METRIC_USE_TRAN_SIZE                 1
#define ARM_METRIC_USE_TRAN_STATUS               2

#define ARM_METRIC_MIN_ARRAY_INDEX               0
#define ARM_METRIC_MAX_ARRAY_INDEX               6
#define ARM_METRIC_MAX_COUNT                     7

#define ARM_METRIC_STRING32_MAX_CHARS           31
#define ARM_METRIC_STRING32_MAX_LENGTH
    (ARM_METRIC_STRING32_MAX_CHARS*3+1)

/* ------------------- misc string defines ----------------------- */

#define ARM_NAME_MAX_CHARS                     127
#define ARM_NAME_MAX_LENGTH(ARM_NAME_MAX_CHARS*3+1)

#define ARM_DIAG_DETAIL_MAX_CHARS             4095
#define ARM_DIAG_DETAIL_MAX_LENGTH(ARM_DIAG_DETAIL_MAX_CHARS*3+1)

#define ARM_MSG_BUFFER_CHARS                   255
#define ARM_MSG_BUFFER_LENGTH(ARM_MSG_BUFFER_CHARS*3+1)

/* ------------------- properties defines ------------------------ */

#define ARM_PROPERTY_MIN_ARRAY_INDEX             0
#define ARM_PROPERTY_MAX_ARRAY_INDEX            19
#define ARM_PROPERTY_MAX_COUNT                  20

#define ARM_PROPERTY_NAME_MAX_CHARS(ARM_NAME_MAX_CHARS)
```

```
#define ARM_PROPERTY_NAME_MAX_LENGTH(ARM_PROPERTY_NAME_MAX_CHARS*3+1)
#define ARM_PROPERTY_VALUE_MAX_CHARS          255
#define ARM_PROPERTY_VALUE_MAX_LENGTH(ARM_PROPERTY_VALUE_MAX_CHARS*3+1)

#define ARM_PROPERTY_URI_MAX_CHARS           4095
#define ARM_PROPERTY_URI_MAX_LENGTH(ARM_PROPERTY_URI_MAX_CHARS*3+1)

/* -------------- system address format values ------------------- */

#define ARM_SYSADDR_FORMAT_RESERVED             0
#define ARM_SYSADDR_FORMAT_IPV4                 1
#define ARM_SYSADDR_FORMAT_IPV4PORT             2
#define ARM_SYSADDR_FORMAT_IPV6                 3
#define ARM_SYSADDR_FORMAT_IPV6PORT             4
#define ARM_SYSADDR_FORMAT_SNA                  5
#define ARM_SYSADDR_FORMAT_X25                  6
#define ARM_SYSADDR_FORMAT_HOSTNAME             7
#define ARM_SYSADDR_FORMAT_UUID                 8

/* ------------------ mandatory charsets ------------------------ */

/* IANA charset MIBenum numbers (http://www.iana.org/) */
#define ARM_CHARSET_ASCII                       3  /* mandatory */
#define ARM_CHARSET_UTF8                      106  /* mandatory */
#define ARM_CHARSET_UTF16BE                  1013
#define ARM_CHARSET_UTF16LE                  1014
    /* mandatory on Windows */
#define ARM_CHARSET_UTF16                    1015
#define ARM_CHARSET_IBM037                   2028
    /* mandatory on iSeries */
#define ARM_CHARSET_IBM1047                  2102
    /* mandatory on zSeries */

/* ------------- flags to be passed on API calls ----------------- */

/* Use ARM_FLAG_NONE instead of zero to be more readable.          */
#define ARM_FLAG_NONE                   (0x00000000)

/* ARM_FLAG_TRACE_REQUEST could be used in the following calls to  */
/* request a trace:                                                */
/*   - arm_generate_correlator()                                   */
/*   - arm_start_transaction()                                     */
/*   - arm_report_transaction()                                    */
/* NOTE: The agent need not support instance tracing, so to be     */
/* sure check the generated correlator using the                   */
/* arm_get_correlator_flags() function.                            */
#define ARM_FLAG_TRACE_REQUEST          (0x00000001)

/* ARM_FLAG_BIND_THREAD could be used on arm_start_transaction()   */
/* call to do an implicit arm_bind_thread().                       */
#define ARM_FLAG_BIND_THREAD            (0x00000002)

/* ARM_FLAG_CORR_IN_PROCESS indicates that a correlator will only  */
/* be used within the process it was created. So an ARM            */
```

```
/* implementation may optimize the generation of a correlator      */
/* for that special usage. This flag can be passed to:             */
/*   - arm_generate_correlator()                                   */
/*   - arm_start_transaction()                                     */
/* NOTE: The agent need not support in-process correlation at all. */
#define ARM_FLAG_CORR_IN_PROCESS     (0x00000004)

/* --------------- correlator defines ---------------------------- */

#define ARM_CORR_MAX_LENGTH                    512
    /* total max length */

/* Correlator interface flag numbers. See                           */
/* arm_get_correlator_flags().                                      */
#define ARM_CORR_FLAGNUM_APP_TRACE               1
#define ARM_CORR_FLAGNUM_AGENT_TRACE             2


/* Use if no correlator should be provided (e.g., in               */
/* arm_start_transaction().                                        */
#define ARM_CORR_NONE ((arm_correlator_t *) NULL)

/* --- current time for arm_report_transaction() stop time ------- */

#define ARM_USE_CURRENT_TIME ((arm_stop_time_t)-1)

/* ------------------ misc defines ------------------------------- */

/* Use ARM_BUF4_NONE instead of a NULL to be more readable.        */
#define ARM_BUF4_NONE ((arm_buffer4_t*) NULL)

/* Use ARM_ID_NONE instead of a NULL to be more readable.          */
#define ARM_ID_NONE ((arm_id_t *) NULL)

/* ------------------------------------------------------------- */
/* --------------- basic typedef section ------------------------- */
/* ------------------------------------------------------------- */

/* Generic data types */
/* ARM4_*INT* defines are set in the <arm4os.h> header file.       */
/* They are platform/compiler-specific.                            */
typedef ARM4_CHAR arm_char_t;

typedef ARM4_INT8 arm_int8_t;
typedef ARM4_UINT8 arm_uint8_t;
    /* used to define an opaque byte array */

typedef ARM4_INT16 arm_int16_t;
typedef ARM4_UINT16 arm_uint16_t;

typedef ARM4_INT32 arm_int32_t;
typedef ARM4_UINT32 arm_uint32_t;

typedef ARM4_INT64 arm_int64_t;
```

```
typedef ARM4_UINT64 arm_uint64_t;

/* ARM-specific simple types */
typedef arm_int32_t arm_boolean_t;
typedef arm_int32_t arm_error_t;

typedef arm_int64_t arm_arrival_time_t;  /* opaque arrival time */
typedef arm_int64_t arm_stop_time_t;  /* stop time in milli secs */
typedef arm_int64_t arm_response_time_t;
    /* response time in nano secs */

typedef arm_int32_t arm_tran_status_t;  /* ARM_TRAN_STATUS_* values */
typedef arm_int32_t arm_charset_t;  /* IANA MIBenum values */
typedef arm_int32_t arm_sysaddr_format_t;  /* ARM_SYSADDR_* values */

/* ARM string buffer types */
typedef arm_char_t arm_message_buffer_t[ARM_MSG_BUFFER_LENGTH];

/* subbuffer types */
typedef arm_int32_t arm_subbuffer_format_t;

/* metric types */
typedef arm_uint8_t arm_metric_format_t;
typedef arm_uint8_t arm_metric_slot_t;
typedef arm_int16_t arm_metric_usage_t;

/* handle types */
typedef arm_int64_t arm_app_start_handle_t;
typedef arm_int64_t arm_tran_start_handle_t;
typedef arm_int64_t arm_tran_block_handle_t;

/* correlator types */
typedef arm_int16_t arm_correlator_length_t;

/* --------------------------------------------------------------- */
/* ---------------- compound typedefs section -------------------- */
/* --------------------------------------------------------------- */

/* All IDs are 16 bytes on an 8-byte boundary. */
typedef struct arm_id
{
    union
    {
        arm_uint8_t  uint8[16];
        arm_uint32_t uint32[4];
        arm_uint64_t uint64[2];
    } id_u;
} arm_id_t;

/* Correlator */
typedef struct arm_correlator
{
    arm_uint8_t opaque[ARM_CORR_MAX_LENGTH];
} arm_correlator_t;
```

```
/* User-defined metrics */
typedef arm_int32_t arm_metric_counter32_t;
typedef arm_int64_t arm_metric_counter64_t;
typedef arm_int32_t arm_metric_divisor32_t;
typedef arm_int32_t arm_metric_gauge32_t;
typedef arm_int64_t arm_metric_gauge64_t;
typedef arm_int32_t arm_metric_numericID32_t;
typedef arm_int64_t arm_metric_numericID64_t;
typedef const arm_char_t *arm_metric_string32_t;
typedef struct arm_metric_cntrdivr32
{
    arm_metric_counter32_t counter;
    arm_metric_divisor32_t divisor;
} arm_metric_cntrdivr32_t;
typedef struct arm_metric_gaugedivr32
{
    arm_metric_gauge32_t gauge;
    arm_metric_divisor32_t divisor;
} arm_metric_gaugedivr32_t;

typedef struct arm_metric
{
    arm_metric_slot_t slot;
    arm_metric_format_t format;
    arm_metric_usage_t usage;
    arm_boolean_t valid;
    union
    {
        arm_metric_counter32_t counter32;
        arm_metric_counter64_t counter64;
        arm_metric_cntrdivr32_t counterdivisor32;
        arm_metric_gauge32_t gauge32;
        arm_metric_gauge64_t gauge64;
        arm_metric_gaugedivr32_t gaugedivisor32;
        arm_metric_numericID32_t numericid32;
        arm_metric_numericID64_t numericid64;
        arm_metric_string32_t string32;
    } metric_u;
} arm_metric_t;

typedef struct arm_metric_binding
{
    arm_metric_slot_t slot;
    arm_id_t id;
} arm_metric_binding_t;

typedef struct arm_property
{
    const arm_char_t *name;
    const arm_char_t *value;
} arm_property_t;
```

```
/* ------------------------------------------------------------- */
/* -------------- sub-buffer typedefs section ------------------ */
/* ------------------------------------------------------------- */

typedef struct arm_subbuffer {
    arm_subbuffer_format_t format;
    /* Format-specific data fields follow here. */
} arm_subbuffer_t;

/* This macro could be used avoid a compiler warning if you      */
/* direct one of the following arm_subbuffer_*_t structure       */
/* pointers to a function accepting sub-buffer pointers. Any     */
/* sub-buffer is passed to the ARM API call as a                 */
/* (arm_subbuffer_t *) pointer. Use this macro if you pass a     */
/* "real" subbuffer to an API function. Note for the special     */
/* ARM SDK subbuffers the ARM_SDKSB() macro has to be used.      */
#define ARM_SB(x)  (&((x).header))

/* The user data buffer */
typedef struct arm_buffer4
{
    arm_int32_t count;
    arm_subbuffer_t **subbuffer_array;
} arm_buffer4_t;

typedef struct arm_subbuffer_charset
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_CHARSET */

    arm_charset_t charset;  /* One of the IANA MIBenum values */
    arm_int32_t flags;
} arm_subbuffer_charset_t;

typedef struct arm_subbuffer_app_identity
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_APP_IDENTITY */

    arm_int32_t identity_property_count;
    const arm_property_t *identity_property_array;
    arm_int32_t context_name_count;
    const arm_char_t **context_name_array;
} arm_subbuffer_app_identity_t;

typedef struct arm_subbuffer_app_context
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_APP_CONTEXT */

    arm_int32_t context_value_count;
    const arm_char_t **context_value_array;
} arm_subbuffer_app_context_t;

typedef struct arm_subbuffer_tran_identity
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_TRAN_IDENTITY */
```

```
    arm_int32_t identity_property_count;
    const arm_property_t *identity_property_array;
    arm_int32_t context_name_count;
    const arm_char_t **context_name_array;
    const arm_char_t *uri;
} arm_subbuffer_tran_identity_t;

typedef struct arm_subbuffer_tran_context
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_TRAN_CONTEXT */

    arm_int32_t context_value_count;
    const arm_char_t **context_value_array;
    const arm_char_t *uri;
} arm_subbuffer_tran_context_t;

typedef struct arm_subbuffer_arrival_time
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_ARRIVAL_TIME */

    arm_arrival_time_t opaque_time;
} arm_subbuffer_arrival_time_t;

typedef struct arm_subbuffer_metric_bindings
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_METRIC_BINDINGS */

    arm_int32_t count;
    const arm_metric_binding_t *metric_binding_array;
} arm_subbuffer_metric_bindings_t;

typedef struct arm_subbuffer_metric_values
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_METRIC_VALUES */

    arm_int32_t count;
    const arm_metric_t *metric_value_array;
} arm_subbuffer_metric_values_t;

typedef struct arm_subbuffer_user
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_USER */

    const arm_char_t *name;
    arm_boolean_t id_valid;
    arm_id_t id;
} arm_subbuffer_user_t;

typedef struct arm_subbuffer_system_address
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_SYSTEM_ADDRESS */

    arm_int16_t address_format;
```

```
    arm_int16_t address_length;
    const arm_uint8_t *address;
    arm_boolean_t id_valid;
    arm_id_t id;
} arm_subbuffer_system_address_t;

typedef struct arm_subbuffer_diag_detail
{
    arm_subbuffer_t header;  /* ARM_SUBBUFFER_DIAG_DETAIL */

    const arm_char_t *diag_detail;
} arm_subbuffer_diag_detail_t;

/* ----------------------------------------------------------------- */
/* ------------------ ARM4 API section --------------------------- */
/* ----------------------------------------------------------------- */

/* register metadata API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_register_application(
    const arm_char_t *app_name,
    const arm_id_t *input_app_id,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_id_t *output_app_id);

ARM4_API_DYNAMIC(arm_error_t)
arm_destroy_application(
    const arm_id_t *app_id,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

ARM4_API_DYNAMIC(arm_error_t)
arm_register_transaction(
    const arm_id_t *app_id,
    const arm_char_t *tran_name,
    const arm_id_t *input_tran_id,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_id_t *output_tran_id);

ARM4_API_DYNAMIC(arm_error_t)
arm_register_metric(
    const arm_id_t *app_id,
    const arm_char_t *metric_name,
    const arm_metric_format_t metric_format,
    const arm_metric_usage_t metric_usage,
    const arm_char_t *metric_unit,
    const arm_id_t *input_metric_id,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_id_t *output_metric_id);
```

```
/* application instance API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_start_application(
    const arm_id_t *app_id,
    const arm_char_t *app_group,
    const arm_char_t *app_instance,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_app_start_handle_t *app_handle);

ARM4_API_DYNAMIC(arm_error_t)
arm_stop_application(
    const arm_app_start_handle_t app_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

/* transaction instance API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_start_transaction(
    const arm_app_start_handle_t app_handle,
    const arm_id_t *tran_id,
    const arm_correlator_t *parent_correlator,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_tran_start_handle_t *tran_handle,
    arm_correlator_t *current_correlator);

ARM4_API_DYNAMIC(arm_error_t)
arm_stop_transaction(
    const arm_tran_start_handle_t tran_handle,
    const arm_tran_status_t tran_status,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

ARM4_API_DYNAMIC(arm_error_t)
arm_update_transaction(
    const arm_tran_start_handle_t tran_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

ARM4_API_DYNAMIC(arm_error_t)
arm_discard_transaction(
    const arm_tran_start_handle_t tran_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

ARM4_API_DYNAMIC(arm_error_t)
arm_block_transaction(
    const arm_tran_start_handle_t tran_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_tran_block_handle_t *block_handle);
```

```
ARM4_API_DYNAMIC(arm_error_t)
arm_unblock_transaction(
    const arm_tran_start_handle_t tran_handle,
    const arm_tran_block_handle_t block_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

/* thread support API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_bind_thread(
    const arm_tran_start_handle_t tran_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

ARM4_API_DYNAMIC(arm_error_t)
arm_unbind_thread(
    const arm_tran_start_handle_t tran_handle,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

/* report transaction data API function */
ARM4_API_DYNAMIC(arm_error_t)
arm_report_transaction(
    const arm_app_start_handle_t app_handle,
    const arm_id_t *tran_id,
    const arm_tran_status_t tran_status,
    const arm_response_time_t response_time,
    const arm_stop_time_t stop_time,
    const arm_correlator_t *parent_correlator,
    const arm_correlator_t *current_correlator,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4);

/* correlator API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_generate_correlator(
    const arm_app_start_handle_t app_handle,
    const arm_id_t *tran_id,
    const arm_correlator_t *parent_correlator,
    const arm_int32_t flags,
    const arm_buffer4_t *buffer4,
    arm_correlator_t *current_correlator);

ARM4_API_DYNAMIC(arm_error_t)
arm_get_correlator_length(
    const arm_correlator_t *correlator,
    arm_correlator_length_t *length);

ARM4_API_DYNAMIC(arm_error_t)
arm_get_correlator_flags(
    const arm_correlator_t *correlator,
    const arm_int32_t corr_flag_num,
    arm_boolean_t *flag);
```

```
/* miscellaneous API functions */
ARM4_API_DYNAMIC(arm_error_t)
arm_get_arrival_time(
    arm_arrival_time_t *opaque_time);

ARM4_API_DYNAMIC(arm_error_t)
arm_get_error_message(
    const arm_charset_t charset,
    const arm_error_t code,
    arm_message_buffer_t msg);

ARM4_API_DYNAMIC(arm_error_t)
arm_is_charset_supported(
    const arm_charset_t charset,
    arm_boolean_t *supported);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#ifndef ARM4DYN_H_INCLUDED
#include "arm4dyn.h"
#endif /* !ARM4DYN_H_INCLUDED */

#endif /* ARM4_H_INCLUDED */
```

11 <arm4os.h> Header File for Compiling

Within the **<arm4.h>** header file, macros are used to separate compiler and operating system specifics from the ARM API. These macros are defined in the **<arm4os.h>** header file. The standard **<arm4.h>** header file includes the **<arm4os.h>** header file. This section describes the minimum set of macros that must be defined in the **<arm4os.h>** header. For a reference implementation of these macros, the **<arm4os.h>** header file found in the ARM4 SDK can be downloaded from The Open Group web site at www.opengroup.org/tech/management/arm/.

The following macros define the required ARM4 data types, and each must be defined in **<arm4.h>**, or a header file accessible to it, in order to satisfy all platform and compiler-specific data types. The **typedef** statement is used with an appropriate preprocessor **#define**.

For example, the **arm_int64_t** type is defined as follows:

```
typedef ARM4_INT64 arm_int64_t;
```

The ARM4_INT64 macro must contain the correct native 64-bit integer type for the platform and compiler used.

ARM4_CHAR — Defines the type of a character (mostly this is a "**char**").

ARM4_INT8 — Defines the type of an 8-bit wide signed integer value.

ARM4_UINT8 — Defines the type of an 8-bit wide unsigned integer value.

ARM4_INT16 — Defines the type of a 16-bit wide signed integer value.

ARM4_UINT16 — Defines the type of a 16-bit wide unsigned integer value.

ARM4_INT32 — Defines the type of a 32-bit wide signed integer value.

ARM4_UINT32 — Defines the type of a 32-bit wide unsigned integer value.

ARM4_INT64 — Defines the type of a 64-bit wide signed integer value.

ARM4_UINT64 — Defines the type of a 64-bit wide unsigned integer value.

ARM4_API_DYNAMIC(return_type)

The ARM4_API_DYNAMIC mcro is used to place compiler and/or operating system-specific keywords before each prototype of the ARM4 API calls that are placed inside a shared library. On some systems the keywords have to be placed before the prototype, and on other systems the keywords have to be placed between the return type and the function name. Therefore the macro takes the return type as its argument.

A Application Instrumentation Sample

```
/* ---------------------------------------------------------------- */
/* ------- ARM4 C API Example: Automated Teller Machine ----------- */
/* ---------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "arm4sdk.h"

/* ---------------------------------------------------------------- */
/* Example: ARMed Automated Teller Machine simulation               */
/*                                                                  */
/* Description:                                                     */
/*                                                                  */
/* This ATM example demonstrates a more complex ARMed application   */
/* using the new ARM4 C API. It uses ARM correlators, ARM           */
/* properties, ARM user, ARM metrics, diagnostic detail for error   */
/* reporting, and the arrival time feature.                         */
/*                                                                  */
/* 1. Register any ARM class such as applications, transactions,   */
/*    or metrics using arm_register_*() calls.                      */
/* 2. Start an application instance using the                       */
/*    arm_start_application() call.                                 */
/* 3. Start and stop any transaction using arm_start_transaction()  */
/*    and arm_stop_transaction() calls.                             */
/* 4. Stop the ARM application instance using the                   */
/*    arm_stop_application() call.*/
/* 5. Destroy all registered ARM classes using the                  */
/*    arm_destroy_application() call.                               */
/* ---------------------------------------------------------------- */

/* ---------------------------------------------------------------- */
/* ------------------- atm data structures ------------------------ */
/* ---------------------------------------------------------------- */

typedef struct atm_customer
{
    int no;
    char name[32];
    int pin;
    double balance;
    char currency[8];
} atm_customer_t;

atm_customer_t customers[] = {
    {1, "Mueller", 1234, 150.0, "EUR"},
```

```
    {2, "Miller", 5678, 100.0, "GBP"},
    {3, "Meyer", 4711, 200.0, "USD"}
};
int max_customers = sizeof(customers)/sizeof(atm_customer_t);

enum atm_errors
{
    ERR_NONE,
    ERR_INVALID_NO,
    ERR_INVALID_PIN,
    ERR_OVERDRAWN,
    ERR_MAX
};
const char *errors[ERR_MAX] = {
    "no error",
    "invalid customer number",
    "invalid pin number",
    "amount overdrawn"
};
int max_errors = sizeof(errors)/sizeof(const char*);

/* --------------------------------------------------------------- */
/* -------------------- arm data structures --------------------- */
/* --------------------------------------------------------------- */

typedef struct atm_arm
{
    /* application class ID */
    arm_id_t app_id;

    /* transaction class IDs */
    arm_id_t tran_atm_id;
    arm_id_t tran_check_customer_id;
    arm_id_t tran_withdraw_id;

    /* metric class IDs */
    arm_id_t metric_customer_no_id;
    arm_id_t metric_customer_pin_id;
    arm_id_t metric_amount_id;
    arm_id_t metric_old_balance_id;
    arm_id_t metric_new_balance_id;

    /* application start handle */
    arm_app_start_handle_t app_handle;

    /* transaction start handles */
    arm_tran_start_handle_t tran_atm_handle;
    arm_tran_start_handle_t tran_check_customer_handle;
    arm_tran_start_handle_t tran_withdraw_handle;

    /* correlator storage for parent transactions */
    arm_correlator_t corr_atm;
    arm_correlator_t corr_check_customer;
} atm_arm_t;
```

```
/* ------------------------------------------------------------------ */
/* ----------------------- prototypes ---------------------------- */
/* ------------------------------------------------------------------ */

/* support functions for getting response from the atm user */
static void get_string(const char *prompt, char *buf, int len);
static int get_int(const char *prompt);
static double get_amount(const char *prompt);
static void error(int err);

/* arm initialize and freeing functions */
static void atm_arm_init(atm_arm_t *arm);
static void atm_arm_free(atm_arm_t *arm);

/* functions of the atm */
static void atm(atm_arm_t *arm);
static void check_customer(atm_arm_t *arm);
static void withdraw(int idx, atm_arm_t *arm);

/* ------------------------------------------------------------------ */
/* --------------- support input/output functions ----------------- */
/* ------------------------------------------------------------------ */

static void get_string(const char *prompt, char *buf, int len)
{
    char *ptr;
    fputs(prompt, stdout);
    fflush(stdout);
    fgets(buf, len, stdin);
    if((ptr = strchr(buf, '\n')) != NULL)
        *ptr = '\0';
}

static int get_int(const char *prompt)
{
    char buf[32];
    get_string(prompt, buf, sizeof(buf));
    return atoi(buf);
}

static double get_amount(const char *prompt)
{
    char buf[32];
    get_string(prompt, buf, sizeof(buf));
    return atof(buf);
}

static void error(int err)
{
    printf("\nerror: %s\n", errors[err]);
}
```

```
/* ---------------------------------------------------------------- */
/* --------------- arm init & free functions ---------------------- */
/* ---------------------------------------------------------------- */

static void atm_arm_init(atm_arm_t *arm)
{
    armsdk_buffer4_t buffer4;
    armsdk_subbuffer_metric_bindings_t sb_mbindings;
    armsdk_subbuffer_tran_identity_t sb_tran_identity;

    /* Register application class with arm. */
    arm_register_application("ATM - Automated Teller Machine",
        ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE, &arm->app_id);

    /* Register metrics classes with arm. */
    arm_register_metric(&arm->app_id, "Customer-No.",
        ARM_METRIC_FORMAT_NUMERICID32, ARM_METRIC_USE_GENERAL,
        "ID", ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
        &arm->metric_customer_no_id);
    arm_register_metric(&arm->app_id, "Customer-Pin",
        ARM_METRIC_FORMAT_NUMERICID32, ARM_METRIC_USE_GENERAL,
        "ID",  ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
        &arm->metric_customer_pin_id);

    arm_register_metric(&arm->app_id, "Amount",
        ARM_METRIC_FORMAT_COUNTER32, ARM_METRIC_USE_GENERAL,
        "money",  ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
        &arm->metric_amount_id);
    arm_register_metric(&arm->app_id, "Old balance",
        ARM_METRIC_FORMAT_GAUGE32, ARM_METRIC_USE_GENERAL,
        "money",  ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
        &arm->metric_old_balance_id);
    arm_register_metric(&arm->app_id, "New balance",
        ARM_METRIC_FORMAT_GAUGE32, ARM_METRIC_USE_GENERAL,
        "money", ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
        &arm->metric_new_balance_id);

    /* Register transaction classes with arm. */
    arm_register_transaction(&arm->app_id, "ATMTran", ARM_ID_NONE,
        ARM_FLAG_NONE, ARM_BUF4_NONE, &arm->tran_atm_id);

    armsdk_metric_binding_initialize(&sb_mbindings);
    armsdk_metric_binding_set(&sb_mbindings, 0,
        &arm->metric_customer_no_id);
    armsdk_metric_binding_set(&sb_mbindings, 1,
        &arm->metric_customer_pin_id);

    armsdk_buffer4_initialize(&buffer4);
    armsdk_buffer4_subbuffer_set(&buffer4, ARMSDK_SB(&sb_mbindings));

    arm_register_transaction(&arm->app_id, "Check Customer",
        ARM_ID_NONE, ARM_FLAG_NONE, &buffer4.header,
        &arm->tran_check_customer_id);
```

```
    armsdk_metric_binding_initialize(&sb_mbindings);
    armsdk_metric_binding_set(&sb_mbindings, 0,
        &arm->metric_amount_id);
    armsdk_metric_binding_set(&sb_mbindings, 1,
        &arm->metric_old_balance_id);
    armsdk_metric_binding_set(&sb_mbindings, 2,
        &arm->metric_new_balance_id);

    armsdk_tran_identity_initialize(&sb_tran_identity);
    armsdk_tran_identity_context_name_set(&sb_tran_identity, 0,
        "Currency");

    armsdk_buffer4_initialize(&buffer4);
    armsdk_buffer4_subbuffer_set(&buffer4, ARMSDK_SB(&sb_mbindings));
    armsdk_buffer4_subbuffer_set(&buffer4,
        ARMSDK_SB(&sb_tran_identity));

    arm_register_transaction(&arm->app_id, "Withdraw", ARM_ID_NONE,
        ARM_FLAG_NONE, &buffer4.header, &arm->tran_withdraw_id);

    /* Now start our arm application. */
    arm_start_application(&arm->app_id, "Examples", NULL,
        ARM_FLAG_NONE, ARM_BUF4_NONE, &arm->app_handle);
}

static void atm_arm_free(atm_arm_t *arm)
{
    /* Stop the application instance. */
    arm_stop_application(arm->app_handle, ARM_FLAG_NONE,
        ARM_BUF4_NONE);

    /* Destroy all registered metadata. */
    arm_destroy_application(&arm->app_id, ARM_FLAG_NONE,
        ARM_BUF4_NONE);
}

static void atm(atm_arm_t *arm)
{
    /* Now start the atm transaction. */
    arm_start_transaction(arm->app_handle, &arm->tran_atm_id,
        ARM_CORR_NONE, ARM_FLAG_CORR_IN_PROCESS, ARM_BUF4_NONE,
        &arm->tran_atm_handle, &arm->corr_atm);

    printf("atm:\n"
        "----\n");
    check_customer(arm);
    printf("bye.\n\n");

    /* Stop the measurement for the atm transaction and commit */
    /* it to ARM. */
    arm_stop_transaction(arm->tran_atm_handle, ARM_STATUS_GOOD,
        ARM_FLAG_NONE, ARM_BUF4_NONE);
}
```

```
static void check_customer(atm_arm_t *arm)
{
    armsdk_buffer4_t buffer4;
    armsdk_subbuffer_metric_values_t sb_mvalues;
    arm_subbuffer_diag_detail_t sb_diag_detail;
    arm_subbuffer_user_t sb_user;
    arm_subbuffer_arrival_time_t sb_arrival;
    arm_arrival_time_t arrival;

    int idx=-1;
    int err = ERR_INVALID_NO;
    int i;
    int no;
    int pin;

    /* The real check_customer transaction starts here. But    */
    /* currently we don't know the customer (arm user) so get  */
    /* a time stamp from arm and pass it later on              */
    /* arm_start_transaction() when we know the customer name! */
    arm_get_arrival_time(&arrival);

    no = get_int("enter your customer no.: ");
    pin = get_int("enter your pin no.: ");
    for(i=0; i<max_customers; ++i)
    {
        if(customers[i].no == no)
    {
        if(customers[i].pin == pin)
    {
        idx = i;
        err = ERR_NONE;
    } else
        err = ERR_INVALID_PIN;
        break;
    }
}

    armsdk_buffer4_initialize(&buffer4);

    armsdk_arrival_time_initialize(&sb_arrival);
    armsdk_arrival_time_set(&sb_arrival, arrival);
    armsdk_buffer4_subbuffer_set(&buffer4, ARM_SB(&sb_arrival));

    if(idx >= 0)
    {
        armsdk_user_initialize(&sb_user);
        armsdk_user_set(&sb_user, customers[idx].name);
        armsdk_buffer4_subbuffer_set(&buffer4, ARM_SB(&sb_user));
    }

    /* Now start the check_customer transaction. */
    arm_start_transaction(arm->app_handle,
        &arm->tran_check_customer_id, &arm->corr_atm,
        ARM_FLAG_CORR_IN_PROCESS, &buffer4.header,
```

```
        &arm->tran_check_customer_handle,
        &arm->corr_check_customer);

    if(idx >= 0)
        withdraw(idx, arm);

    armsdk_buffer4_initialize(&buffer4);

    armsdk_metric_initialize(&sb_mvalues);
    armsdk_metric_numericid32_set(&sb_mvalues, 0,
        ARM_TRUE, ARM_METRIC_USE_GENERAL, no);
    armsdk_metric_numericid32_set(&sb_mvalues, 1,
        ARM_TRUE, ARM_METRIC_USE_GENERAL, pin);
    armsdk_buffer4_subbuffer_set(&buffer4, ARMSDK_SB(&sb_mvalues));

    if(err > ERR_NONE)
    {
        armsdk_diag_detail_initialize(&sb_diag_detail);
        armsdk_diag_detail_set(&sb_diag_detail, errors[err]);
        armsdk_buffer4_subbuffer_set(&buffer4,
            ARM_SB(&sb_diag_detail));

        error(err);
    }

    /* Stop the measurement for the check_customer transaction. */
    arm_stop_transaction(arm->tran_check_customer_handle,
        (idx >= 0) ? ARM_STATUS_GOOD : ARM_STATUS_FAILED,
            ARM_FLAG_NONE, &buffer4.header);
}

static void withdraw(int idx, atm_arm_t *arm)
{
    armsdk_buffer4_t buffer4;
    armsdk_subbuffer_metric_values_t sb_mvalues;
    armsdk_subbuffer_tran_context_t sb_tran_context;
    arm_subbuffer_diag_detail_t sb_diag_detail;

    int err = ERR_NONE;
    double amount = 0.0;
    double new_balance;
    double old_balance;

    char amount_prompt[80];

    old_balance = new_balance = customers[idx].balance;

    armsdk_tran_context_initialize(&sb_tran_context);
    armsdk_tran_context_set(&sb_tran_context, 0,
        customers[idx].currency);

    armsdk_buffer4_initialize(&buffer4);
    armsdk_buffer4_subbuffer_set(&buffer4,
        ARMSDK_SB(&sb_tran_context));
```

```
    /* Now start the check_customer transaction. */
    arm_start_transaction(arm->app_handle, &arm->tran_withdraw_id,
        &arm->corr_check_customer, ARM_FLAG_NONE, &buffer4.header,
        &arm->tran_withdraw_handle, ARM_CORR_NONE);

    printf(" Hello Mr./Mrs./Miss %s\n"
           " your balance is %.2f %s\n",
        customers[idx].name, customers[idx].balance,
            customers[idx].currency);

    snprintf(amount_prompt, sizeof(amount_prompt),
        " enter amount to withdraw (in %s): ",
        customers[idx].currency);
    amount = get_amount(amount_prompt);
    if(amount > old_balance)
        err = ERR_OVERDRAWN;
    else
        new_balance = old_balance - amount;

    armsdk_metric_initialize(&sb_mvalues);
    armsdk_metric_counter32_set(&sb_mvalues, 0,
        ARM_TRUE, ARM_METRIC_USE_GENERAL,
        (arm_int32_t) (amount * 100.0));
    armsdk_metric_gauge32_set(&sb_mvalues, 1,
        ARM_TRUE, ARM_METRIC_USE_GENERAL,
        (arm_int32_t) (old_balance * 100.0));
    armsdk_metric_gauge32_set(&sb_mvalues, 2,
        ARM_TRUE, ARM_METRIC_USE_GENERAL,
        (arm_int32_t) (new_balance * 100.0));

    armsdk_buffer4_initialize(&buffer4);
    armsdk_buffer4_subbuffer_set(&buffer4, ARMSDK_SB(&sb_mvalues));

    if(err > ERR_NONE)
    {
        armsdk_diag_detail_initialize(&sb_diag_detail);
        armsdk_diag_detail_set(&sb_diag_detail, errors[err]);
        armsdk_buffer4_subbuffer_set(&buffer4,
            ARM_SB(&sb_diag_detail));

        error(err);
    } else
    {
        printf(" your new balance is %.2f %s\n",
            new_balance, customers[idx].currency);
    }

    /* Stop the measurement for the check_customer transaction. */
    arm_stop_transaction(arm->tran_withdraw_handle,
        (err == 0) ? ARM_STATUS_GOOD : ARM_STATUS_FAILED,
            ARM_FLAG_NONE, &buffer4.header);
}
```

```
int main(int argc, char *argv[])
{
    int rc = 0;
    atm_arm_t data;

    atm_arm_init(&data);
    /* access atm */
    atm(&data);
    atm_arm_free(&data);

    return rc;
}
```

B Information for Implementers

This appendix contains information useful to creators of ARM implementations, and analysis and reporting programs that process ARM data. Applications using ARM to measure transactions do not use any of this information.

Reserved Values

Some fields have been reserved for future use.

Field in <arm4.h>	Functions that Use the Field	Reserved Values
arm_tran_status_t	*arm_stop_transaction*() *arm_report_transaction*()	All negative values, except those between –999 and –1.
arm_error_t	Most functions	-20000:-20999

Byte Ordering in Correlators

Correlators are passed from application to application. The transfer may occur within a single system or a single process, or it may occur across a network. The recipient and sender of a correlator may run on different machines with different architectures, and the conventions for ordering bytes in data fields, such as integers and arrays, may be different.

If all the programs that touch a correlator were written in Java, the JVM (Java Virtual Machine) would ensure that the same ordering conventions are followed and no order would need to be specified. However, correlators are meant to be passed between applications using any version of ARM (both C and Java) and running on any platform, including both big-endian and little-endian platforms. Because big-endian and little-endian platforms order bytes differently, the specification needs to explicitly state the required ordering, in order to make the correlators interchangeable.

Recognizing this fact, ARM is designed expressly to permit correlators to be exchanged between any application using ARM and any ARM implementation, regardless of how it is written. For example, an application using ARM 4.0 Java Bindings may receive a (parent) correlator from an application using ARM 2.0 (for C programs), and it may send its correlator to an application using ARM 3.0 for Java programs. To permit these types of exchanges, ARM specifies the ordering of bytes within the correlator.

All correlator fields, and the correlator itself, are sent in network byte order. Network byte order is a standard described as follows. The most significant bit is the first bit sent, and the least significant bit is the last bit sent. For example, a 32-bit integer field would be sent with the most significant byte first, and the least significant byte would be the fourth byte sent.

Byte 0	Byte 1	Byte 2	Byte 3
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Bit 0 is the most significant bit			Bit 31 is the least significant bit

Limits on Interoperability between ARM Implementations

There is one limit on interoperability. In ARM 2.0 and 3.0, the maximum length of a correlator is 168 bytes. ARM 4.0 has changed the maximum to 512 bytes. If ARM 4.0 implementations restrict themselves to correlators of no more than 168 bytes, then the correlators are fully interchangeable with any other version of ARM. If an ARM 4.0 implementation uses a correlator that is more than 168 bytes long, it can only be successfully interchanged with another ARM 4.0 implementation.

Avoiding Interference between ARM Implementations

Each ARM implementation should be installed and configured in a way that avoids interfering with other ARM implementations. This does not mean that all ARM implementations that are installed on a system will receive calls from every application that uses ARM. Only one implementation will receive the calls from the applications in each process. The selection of the implementation is generally dependent on how the system administrator installs and configures the implementations (see Section 1.7), unless the application has been statically linked to one implementation.

Correlator Formats

ARM specifies formatting constraints that all correlators must adhere to. These are described in the following section. In addition, different versions of ARM have defined three specific correlator formats. ARM 4.0 has not defined any formats, and, in general, has taken the approach of making correlators as opaque as possible.

ARM Correlator Format Constraints

These constraints apply to all formats.

Table 2: ARM Correlator Format Constraints

Position	Length	Contents
Bytes 0:1	2 bytes	Length of the correlator, including these two bytes. Valid lengths are 4 <= length <= 512. Lengths shorter than four bytes are not permitted because all correlators must have the four bytes defined in this table. Note that a correlator that is longer than 168 bytes could not be passed to and used by an application using ARM 2.0 or ARM 3.0, because the maximum size in those versions was 168 bytes. Some correlator formats impose shorter length restrictions. In particular,

Position	Length	Contents
		formats 1, 2, and 127 have a maximum of 168 bytes.
Byte 2	1 byte	Correlator format The range 0:127 (unsigned) is reserved by the ARM specification. Six values have been assigned: 1 – Defined in ARM 2.0 2 – Defined in ARM 3.0 28 – Reserved for Hewlett-Packard 103 – Reserved for IBM 122 – Reserved for tang-IT 127 – Defined in ARM 3.0 The range 128:255 (unsigned) is available for use by ARM implementers. Known used values include: 128 – Hewlett-Packard 203 – IBM 204 – IBM
Byte 3	1 byte	Flags All eight-bit flags are reserved by the ARM specification. Two flags are defined in positions 0:1 (the highest order bits), as in *ab*000000, where *a* and *b* are bit flags. *a* = 1 if a trace of this transaction is requested by the agent that generated the correlator. This is transparent to the applications. *b* = 1 if the application indicates that this transaction is of particular importance, such as a test transaction, and therefore worthy of being traced. There are no requirements for how these flags are handled, if at all. By convention, if the flags are turned on in a correlator, the setting is copied into the correlators for child transactions. However, local policy or the ARM implementation may override this convention. The usage scenario that led to their creation was to enable a trace of selected transactions throughout an enterprise. A selective trace would yield much useful information without being a significant burden on the systems processing the transaction. For example, a client could be experiencing response time problems. The agent on the client could turn on the trace flag (bit 0) in the correlators that it generates. When this correlator is passed, as the parent correlator, to the ARM implementation on the server, the ARM implementation could turn on the trace flag in the correlators that it generates. The process could continue recursively. What has resulted is a trace of all the transactions associated with the client experiencing the response time problem, but only those transactions. If there are 1,000 clients in the enterprise running this application, 0.1% of all transactions are traced, which is a minimal load on the systems. The value of a surgical trace like this was considered great enough to justify including it in the ARM specification.

Index

<arm4.h> 4, 91, 93
<arm4os.h>..................................105
API functions...........................25, 32
API overview................................22
API structure 22
application context values sub-buffer 83
application identity sub-buffer 81
application instrumentation 106
ARM
  compatibility between versions....4
  evolution of.................................3
  usage of.......................................1
ARM 4.0
  new capabilities............................3
arm_bind_thread() 33
arm_block_transaction() 34
arm_destroy_application() 36
arm_discard_transaction().............37
arm_generate_correlator() 38
arm_get_arrival_time() 40
arm_get_correlator_flags()............41
arm_get_correlator_length()..........42
arm_get_error_message() 43
arm_is_charset_supported() 45
arm_register_application() 47
arm_register_metric() 49
arm_register_transaction() 52
arm_report_transaction()...............54
arm_start_application() 56
arm_start_transaction() 58
arm_stop_application() 61
arm_stop_transaction() 63
arm_unbind_thread() 65
arm_unblock_transaction() 66
arm_update_transaction()..............68
arrival time sub-buffer....................73
byte order markers..........................30
C Bindings...................................4, 9
calling hierarchy............................15
character set encoding sub-buffer...91
context properties 25
correlator 14, 58
  format constraints 116
correlator formats........................116
correlators
  byte ordering...........................115
counters 17
diagnostic data...............................16
diagnostic detail sub-buffer...........80
distributed transaction 13
error handling................................31
gauges 17
  conventions...............................20
GMT...55
heartbeat.................................11, 29
identification information 16
identity properties 25
interoperability limits..................116
JVM.....................................56, 115
libarm4 6
linked library 6
management agent...........................2
measurement functions 18
measurement information 16
metadata
  registration of............................26
metric bindings sub-buffer............89
metric values sub-buffer................74
metrics..16
  categories of.............................17
multiple values 19
numeric IDs..................................18
optional buffer...............................24
optional buffers 69
parent..58
programming options 11
reserved error codes 31
reserved values............................115
response time................................16
response times 11
SDK..7
shared library...................................6
status 16

values of 16
stop time 16
strings 18
sub-buffers 19
system address sub-buffer 77
thread-safe 25
transaction context sub-buffer 87
transaction identity sub-buffer 84
transaction measurement 2
transaction relationships 13
transactions
measurement of 28
user data buffer 69
user sub-buffer 72