Technical Standard

Application Response Measurement (ARM)

Issue 4.1 Version 1 – C Binding

THE Open GROUP
Making standards work®

Technical Standard

Application Response Measurement (ARM) Issue 4.1 Version 1 – C Binding

ISBN: 1-931624-74-7

Document Number: C071

Published by The Open Group, April 2007.

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

The Open Group, Thames Tower, 37-45 Station Road, Reading, Berkshire, RG1 1LX, United Kingdom

or by electronic mail to: ospecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX® certification.

Further information on The Open Group 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/certification.

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/bookstore.

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 4.1. 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.
- Italics in text denote variable names and function names. Italic strings are also used for emphasis.
- Normal font is used for the names of constants and literals.
- Syntax and code examples are shown in fixed width font.
- Sections marked with a revision number of the ARM standard in parentheses – e.g., (ARM 4.0) – describe a feature that exists since the introduction of the given revision. Features remain in effect in all higher revisions of the same major ARM release.

Trademarks

Acknowledgements

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

- Bill Furnas, Hewlett-Packard
- Mark Johnson, IBM
- Marcus Thoss, tang-IT Consulting GmbH
- Van Wiles, BMC

Referenced Documents

The following documents are referenced in this Technical Standard:

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 (C Binding)	Technical Standard, August 2004, Application Response Measurement (ARM) Issue 4.0 Version 2 – C Binding (C041), published by The Open Group.
ARM 4.0 (Java Binding)	Technical Standard, August 2004, Application Response Measurement (ARM) Issue 4.0 Version 2 – Java Binding (C042), published by The Open Group.
ARM 4.1 (Java Binding)	Technical Standard, April 2007, Application Response Measurement (ARM) Issue 4.1 Version 1 – Java Binding (C072), 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 1155	Structure and Identification of management Information for TCP/IP-based Internets, May 1990.
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.

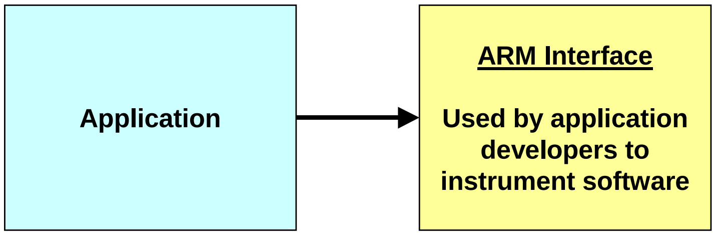

Figure 1: Application – ARM Interface

From the system administrator's perspective, ARM consists of the interfaces that applications load and call and the classes that implement these interfaces, 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.

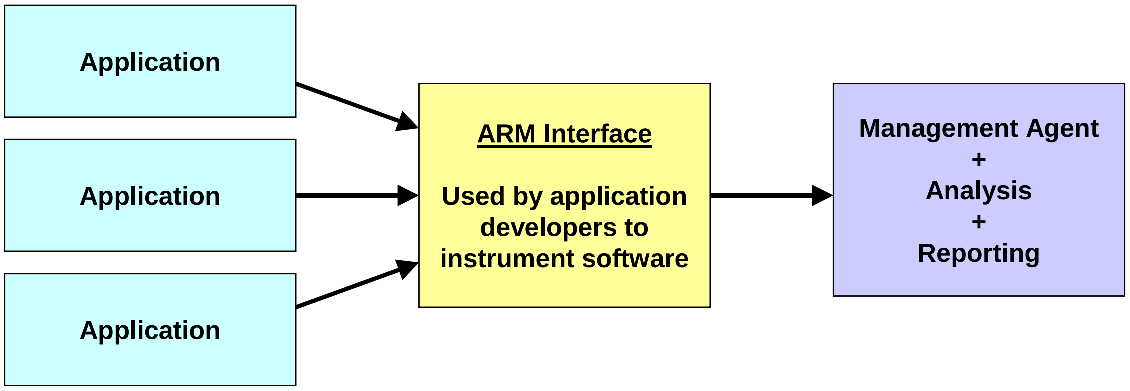

Figure 2: Application – ARM Management System Interaction

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 was developed to implement new capabilities, and to provide equivalent functions for both C and Java programs. ARM 4.0 added 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

ARM 4.1 has been developed to implement new capabilities while maintaining equivalent functions for both C and Java programs. This document describes the C program bindings. A companion document describes the Java program bindings. ARM 4.1 adds the following capabilities:

- An instrumentation control interface that an application may use to query an implementation to determine how much information about a transaction would be useful (from none to very detailed)
- Interfaces to improve ARM's usefulness in messaging and workflow environments
- More flexible mechanisms for reporting properties about a transaction; this is especially useful in instrumented middleware that may not know the properties about a transaction of an application that runs on the middleware until the transaction executes
- Additional flexibility to report when transactions begin executing

1.5 Compatibility between ARM C Binding Versions

ARM defines low-level programming interfaces to be used between program modules 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*. In cases in which the application uses ARM 2.0 and the implementation uses 1.0, the implementation ignores the 2.0 capabilities.
- ARM 4.0 and ARM 4.1 are interoperable. More specifically, any application instrumenting with ARM 4.0 can link to an ARM implementation using ARM 4.1, and *vice versa*. In cases in which the application uses ARM 4.1 and the implementation uses 4.0, the implementation ignores the 4.1 capabilities.
- 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 4.0/4.1 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, **<arm41.h>**. **<arm41.h>** includes the ARM 4.0 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.

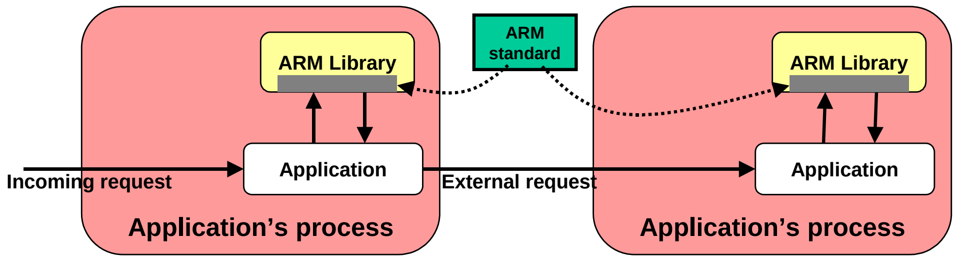

Figure 3: What ARM Looks Like to an Application

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.

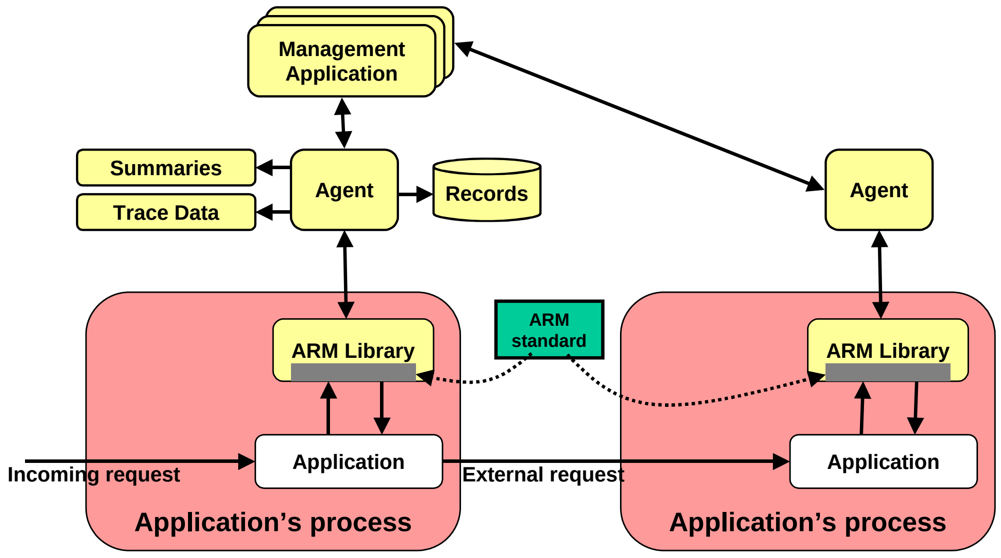

Figure 4: Behind the Scenes of the ARM Interface

1.7 Linking to an ARM 4.0/4.1 Implementation

The most common and most interoperable ways to link to an ARM 4.0/4.1 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 I5/OS (formerly 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.

1.7.1 64-Bit Compiler Restriction

To maintain compatibility between applications and ARM libraries on 64-bit platforms, the compiler must use automatic alignment of pointers on 64-bit platforms.

1.8 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 Getting Started Using ARM 4.0/4.1 C Bindings

ARM is designed to be a relatively simple interface with several optional capabilities. Many applications need only the essential elements, perhaps augmented by a small subset of the optional capabilities, to satisfy their instrumentation requirements.

In order to describe all of ARM's optional capabilities, this specification is by necessity large. Most users need to understand only a subset of the specification. This chapter describes three common sets of capabilities and directs the reader to the relevant sections in the specification. It is hoped this will assist users to quickly start using ARM.

2.1 Basic End-to-End Measurements

All applications must use or are recommended to use the following capabilities:

Capability	Suggested Reading	API Calls Used	Sub-Buffers Used
Registration and initialization	Chapter 5 Chapter 8	*arm_register_application* *arm_register_transaction* *arm_start_application* *arm_stop_application*	[all are optional] Application Identity Application Context Values Transaction Identity Transaction Context
Measure response time and status	Chapter 3	*arm_start_transaction* *arm_stop_transaction*	
[optional] Correlate transactions end-to-end	Section 4.1	arm_get_correlator_length (Additional parameters of *arm_start_transaction*)	
There is a code sample in Section 16.1.			

Table 1: ARM Capabilities for Basic End-to-End Measurements

2.2 Detailed Timing and Threading Measurements

The blocking and threading capabilities described below are useful with management software that monitors and/or manages how transactions interact with the operating system, such as workload management software.

The response time detail measurements are useful when there are transactions that begin executing prior to the time when all the identity context data for the transaction is known.

The best practice instrumentation uses the capabilities described in Table 2: in addition to the basic capabilities described in Table 1.

Capability	Suggested Reading	API Calls Used	Sub-Buffers Used
[optional] Measure time that transactions are blocked	Section 6.2	*arm_block_transaction* *arm_unblock_transaction*	[optional] Block Cause
[optional] Associate transactions to threads	Section 6.3	*arm_bind_thread* *arm_unbind_thread*	
[optional] Measure response times more accurately in certain situations	Section 6.1	*arm_get_arrival_time*	[use at least one of the following] Arrival Time Formatted Arrival Time UsecJan1970 Formatted Arrival Time Strings Preparation Time Preparation Statistics
There is a code sample in Section 16.2.			

Table 2: ARM Capabilities for Detailed Timing and Threading Measurements

2.3 Messaging

The message instrumentation capabilities described below are useful with transactions that are invoked via messages or that use messages to invoke other transactions. The best practice instrumentation uses the capabilities described in Table 3 in addition to the basic capabilities described in Table 1.

Capability	Suggested Reading	Macro Used	Sub-Buffers Used
[optional] Indicate asynchronous flows and independent transactions	Section 4.2	ARM_SET_CORRELATOR_FLAG	
[optional] Indicate message sent and received events	Section 4.2		[use at least one of the following] Message Received Event Message Sent Event
There is a code sample in Section 16.3.			

Table 3: ARM Capabilities for Messaging Instrumentation

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 4.0/4.1 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.

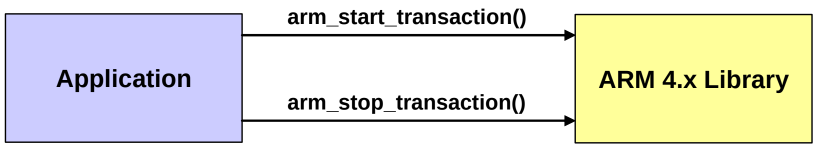

Figure 5: Measurement using Start/Stop

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.

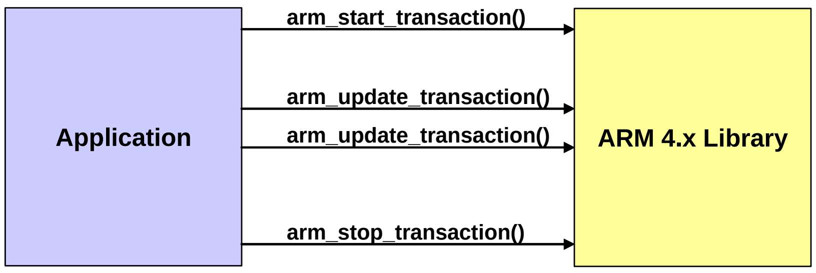

Figure 6: Application using Heartbeats

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.

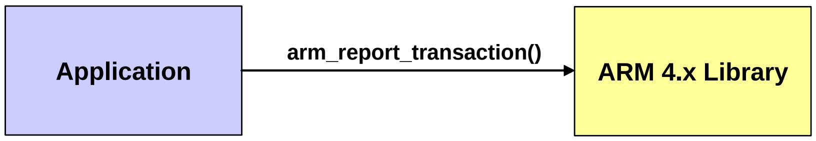

Figure 7: Measurement by the Application

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 uses inline synchronous *arm_start_transaction*() and *arm_stop_transaction*() calls. The calls are made at the moment the real transaction starts and stops, though there is an optional feature to indicate an earlier start time. If they are not, the timings will not be accurate. If the application finds this inconvenient or impractical, it must perform the measurements itself and report them 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. A typical reason why ARM reporting through a different system might be necessary is that the originating system does not support ARM facilities.

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.

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

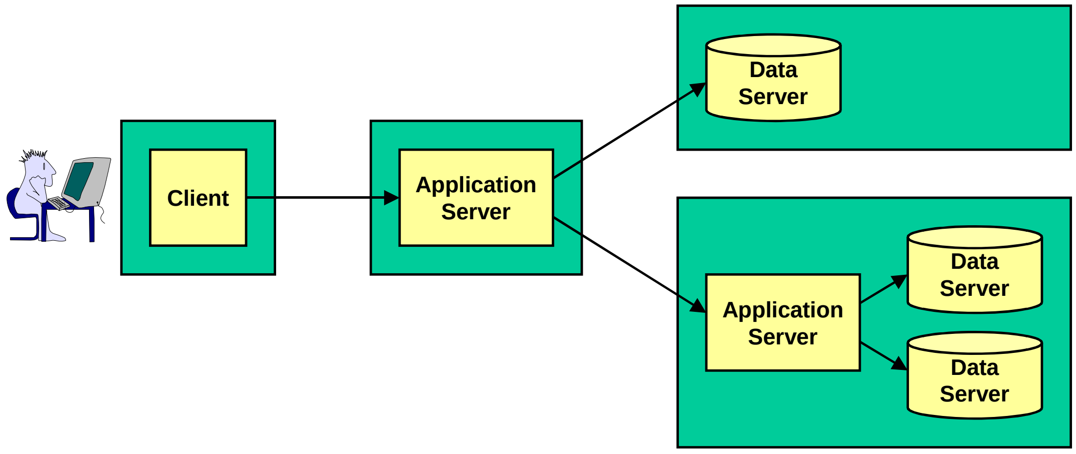

Figure 8: A Common Distributed Application Architecture

4.1 Distributed Transactions with Synchronous Flows

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.

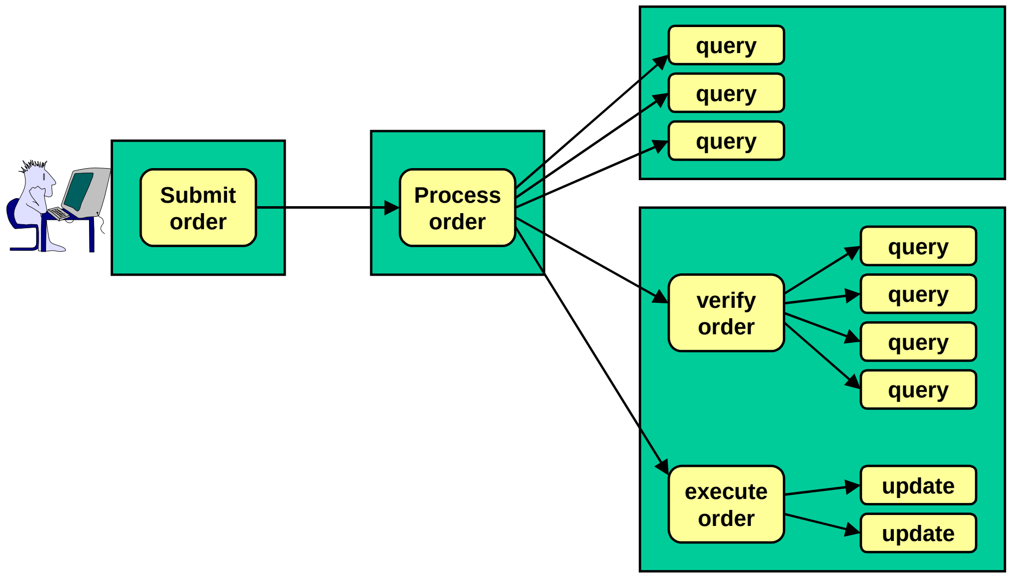

Figure 9: An Example of a Distributed Transaction

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. The correlator format is known to the implementation that creates it, 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.

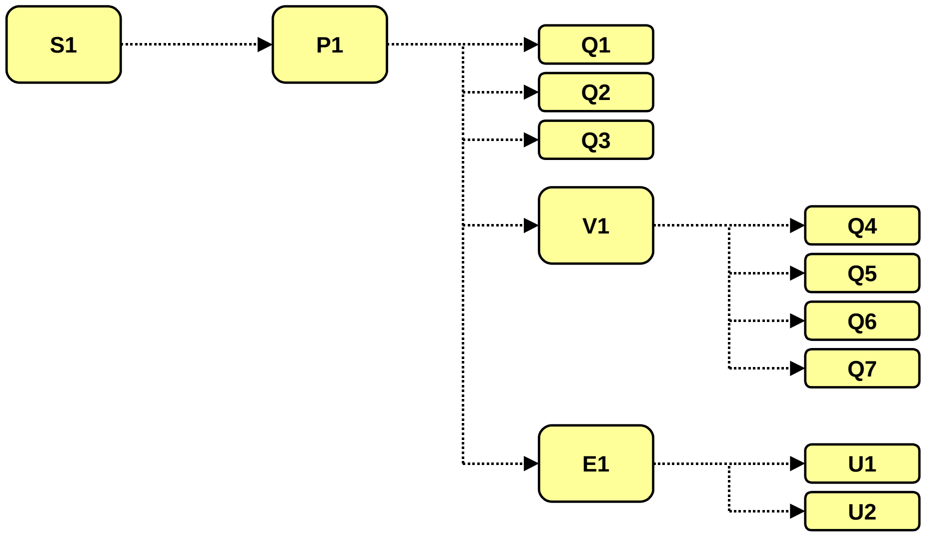

Figure 10: Distributed Transactions that Appear Unrelated

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 is 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.

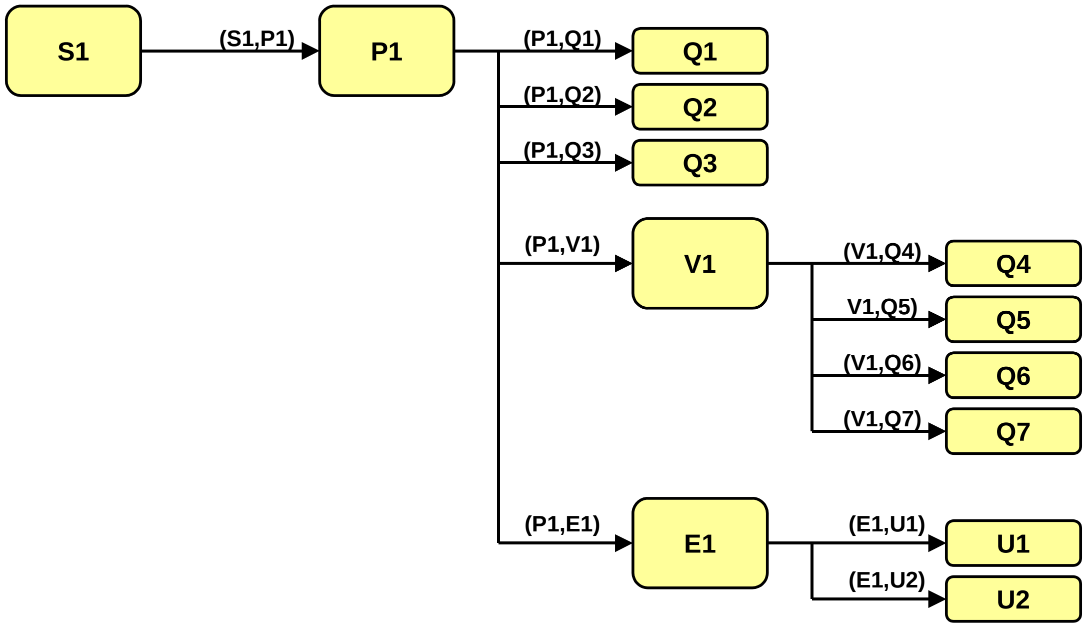

Figure 11: A Distributed Transaction Calling Hierarchy

4.2 Distributed Transactions with Asynchronous Flows (ARM 4.1)

The discussion in Section 4.1 implies synchronous flows between all the component transactions of the distributed transactions. Asynchronous flows using a queuing system are another widely used model for distributed transactions.

An asynchronous flow is one that is accomplished using an asynchronous mechanism, such as a messaging protocol OR if the relationship between the transactions is asynchronous in nature. In the latter case, a parent transaction might initiate a service using a synchronous protocol, but if the parent continues processing other logic in parallel with the invoked service then the overall relationship between the transactions is asynchronous in nature.

In the examples from Figure 12 through Figure 16 the yellow boxes with a single letter in them represent transaction instances. The transactions communicate with each other using queues.

- Figure 12 represents a simple request/response flow. The business logic of A and B are the same as they would be with synchronous flows, but the communication mechanisms are message queues.
- Figure 13 represents a simple request/response flow from the perspective of B or C, each using message queues for communication. From the perspective of A there are two parallel flows that are in a race condition – it is not possible for A to predict in advance whether B or C will return a response first.
- Figure 14: represents a daisy chain flow in which each component passes the results of the transaction on to another component. The last component returns a message to A, which initiated the first flow. Also shown is an optional independent flow (to D) that is spawned by the main flow but which is independent after it is spawned.
- Figure 15 represents a sequential workflow in which no results are returned to the originating transaction (A). This is a common semantic with batch jobs.
- Figure 16 represents a workflow similar to the one in Figure 15, but with parallel paths that converge. C represents a synchronization point, waiting for input from both B and D before proceeding to execute.

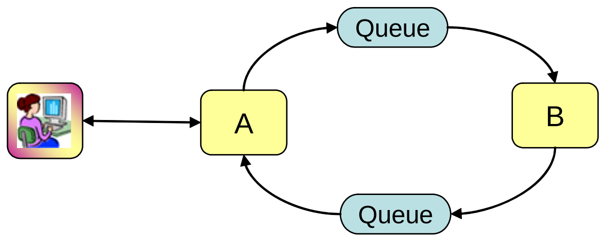

Figure 12: Request/Response with Asynchronous Flows

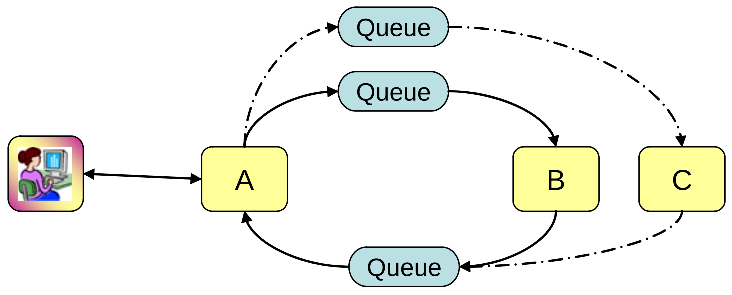

Figure 13: Request/Response with Parallel Asynchronous Flows

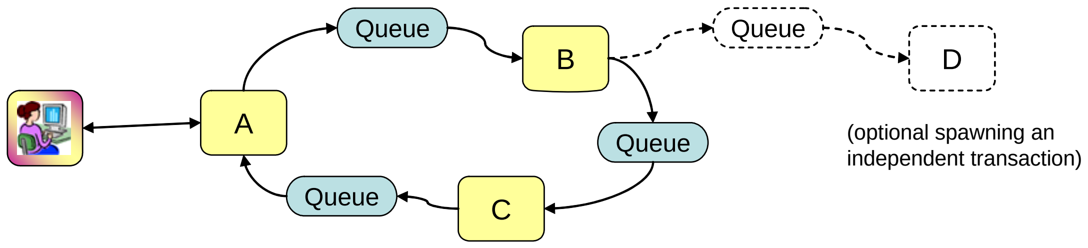

Figure 14: Asynchronous "Daisy-Chain" Asynchronous Flows

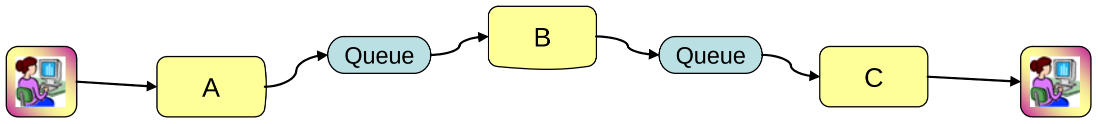

Figure 15: Workflow with Sequential Asynchronous Flows

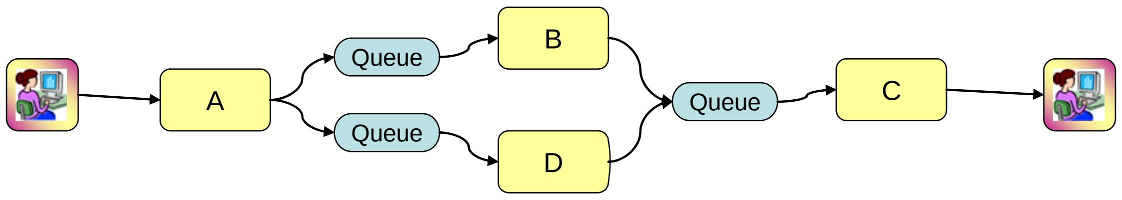

Figure 16: Workflow with Parallel Asynchronous Flows

In each of these five cases it is reasonable for a management application to wish to construct a call graph along the lines of the one depicted in Figure 11. The basic mechanism that is used is the same – creating (parent correlator, current correlator) pairs and then analyzing them to construct the call graph. ARM 4.1 adds extensions that applications can use to make it clearer that the flows are asynchronous and to understand when a transaction is represented by a circular flow, such as is shown in Figure 14:, or a one-way flow, such as is shown in Figure 15.

ARM 4.1 adds three features that enable applications to indicate conditions that describe the relationship between a parent transaction and transactions that it invokes:

- Section 4.2.1 describes the Asynchronous Flow flag in the correlator that indicates that there is an asynchronous relationship between a parent transaction and transactions that it invokes. This flag can be set with the *ARM_SET_CORRELATOR_FLAG*() macro.
- Section 4.2.2 describes the Independent Transaction flag in the correlator that indicates that an invoked transaction does not influence its parent transaction. This flag can be set with the *ARM_SET_CORRELATOR_FLAG*() macro.
- Section 4.2.3 describes message event sub-buffers that can be used to provide additional details about messages that have been exchanged.

4.2.1 Indicating Asynchronous Flows

Setting the Asynchronous Flow flag to TRUE indicates that the control flow from the parent was accomplished via an asynchronous mechanism, such as a messaging protocol OR that the transaction flow is asynchronous in nature. For example, it would be appropriate to set the Asynchronous Flow flag in the following situation even though synchronous protocols are being used:

- A parent uses a synchronous web service call to send a message to a child that starts a transaction. The response to the web service means "I've received your request", NOT "I've completed your request".
- The parent transaction continues executing business logic.
- At some later point in time the parent transaction may issue another synchronous web service call to retrieve the results.

Another way to think about it is that if there is a race condition between the parent and the child transaction such that it is unknown whether the child transaction will complete before the parent transaction continues executing, it is appropriate to use the Asynchronous Flow flag. The circumstance in which it is inappropriate to use the Asynchronous Flow flag is when the caller makes a synchronous call using a synchronous protocol and blocks waiting for the child transaction to complete. It would be appropriate to use the Asynchronous Flow flag if a message is sent using a message queuing protocol even if the application immediately blocks to await a response.

When the flag is FALSE, the ARM agent cannot presume anything about the nature of parent-child interaction.

When used, this flag needs to be set prior to specification of a parent correlator on an *arm_start_transaction*(), *arm_generate_correlator*(), or *arm_report_transaction*() call. The flag could be set by the child recipient of the correlator, but the expectation is that the flag is set most frequently by the parent prior to the control transfer protocol invocation. Note that the

Independent Transaction (IT) flag, described below, has a semantic dependency on this flag, which also potentially influences where this flag is set.

4.2.2 Indicating Independent Flows

The Independent Transaction flag indicates that the invoked transaction will not influence the parent transaction in any way, and in many cases, the invoked transaction would be best thought of and modeled as a new business transaction.

Setting the Independent Transaction flag to TRUE indicates that ARM-reported program logic execution in the child recipient of this correlator reflects transaction work that, although initiated by this parent, is performed independently of the parent's ARM-reported transaction work in scope and purpose. More specifically, setting the flag to TRUE indicates that neither the parent transaction's response time nor status is dependent on the child transaction's execution, nor is there any expected use of the same transaction context.

It logically follows that the child transaction is executing asynchronously to the parent transaction and therefore the Independent Transaction flag may be set to TRUE only if the Asynchronous Flow flag is also set to TRUE. If the Asynchronous Flow flag is FALSE, the Independent Transaction flag is ignored.

Asynchronous Flow Flag	Independent Transaction Flag	Interpretation
FALSE	FALSE	No assumptions about the parent/child interaction can be made.
FALSE	TRUE	The Independent Transaction flag is ignored. No assumptions about the parent/child interaction can be made.
TRUE	FALSE	The child transaction is executing asynchronously to the parent transaction but not independently.
TRUE	TRUE	The child transaction is executing asynchronously to the parent transaction and is also independent of the parent transaction.

Table 4: Correlator Flags in Asynchronous Flows

When used, this flag needs to be set prior to specification of a parent correlator on an *arm_start_transaction*(), *arm_generate_correlator*(), or *arm_report_transaction*() call. The flag could be set by the child recipient of the correlator, but the expectation is that the flag is set most frequently by the parent prior to the control transfer protocol invocation.

The Asynchronous Flow and Independent Transaction flags are interpreted only within the scope of a single parent-child control transfer, and are exclusively manipulated by applications. The ARM agent never sets these flags. The application uses the *ARM_SET_CORRELATOR_FLAG*() macro to set the flags. All correlators constructed by the ARM agent and returned as output on *arm_start_transaction*() and *arm_generate_correlator*() calls are constructed with the Asynchronous Flow and Independent Transaction flags set to FALSE. Note that the flags are never inherited from parent correlators to child correlators, unlike the Agent Trace and Application Trace flags (bit positions 0 and 1, respectively), which often are inherited from the parent correlator.

4.2.3 Event Flows (ARM 4.1)

Asynchronous flows between programs are initiated, controlled, and terminated through the exchange of messages. It can be useful to programs that analyze asynchronous flows and how they impact applications to understand the underlying event flows.

There are two types of events:

- Message Received

  The Message Received Event sub-buffer is optionally used to describe one or more messages that have been received that have an effect on the execution state of a transaction. The cases of interest are the following:

  — A message causes the transaction to initiate.

  — A message causes the transaction to unblock.

  — A message is received that terminates an asynchronous transaction or a step in an asynchronous transaction.

  — A message is received that is related to the transaction, though it may not change the blocked or running state of the transaction.

- Message Sent

  The Message Sent Event sub-buffer is optionally used to describe one or more messages that have been sent that have an effect on the execution state of a transaction. The cases of interest are the following:

  — A message is sent that causes the transaction to block.

  — A message is sent that initiates or terminates an asynchronous transaction or a step in an asynchronous transaction.

  — A message is sent that is part of an exchange between this transaction and another transaction.

With either a Message Received or Message Sent event an application can indicate with the End-Of-Flow flag that the current transaction is the last in a series of transactions that together represent one logical flow. For example, if there are three transactions (A,B,C) that together perform some service, and if the flow between them is A invokes B, which invokes C, and neither B nor C return status or other data to the transaction that invoked it, then C represents the end of the flow A→B→C.

The event data is provided in the Message Received Event sub-buffer (see Section 13.8) and the Message Sent Event sub-buffer (see Section 13.9), respectively. The data may be provided on any of the following API calls: *arm_start_transaction*(), *arm_update_transaction*(), *arm_stop_transaction*(), *arm_block_transaction*(), *arm_unblock_transaction*().

5 Describing Applications and Transactions

ARM 4.1 uses two types of properties to describe applications and transactions: “identity” properties (for applications and transactions), and “context” properties (for applications and transactions). Each property consists of a name string and a value string.

They differ based on when the names and/or values are set, as shown in Table 5.

Type of Property	Same for all Instances	May Vary per Instance
Identity Property (applications and transactions)	Name Value	
Context Property (applications and transactions)	Name	Value

Table 5: Descriptive Property Types

When deciding which property type to use, 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 for both the names and the values and apply to all transaction instances.
- Processing of context property values may occur for every transaction instance. This increases overhead but it is a good practice if each transaction “flavor”, where a flavor represents a different combination of properties, is a slight variation on the same type of transaction. An implementation that does not process the context properties could still provide useful reports if the performance characteristics of each flavor are similar.

5.1 Identity Properties

An identity property is a property that has the same name and value for all instances of an application or transaction.

- The *application name* and *transaction name* parameters are mandatory and provide the most basic identification. Many applications and transactions are satisfactorily identified through the use of only the *name* parameter.
- In addition, there can be up to 20 identity properties of the pattern (name,value) pair, in which both name and value are separate character strings.
- Transactions can also have a URI property that is part of the identity when it is provided.

Identity property names and values are provided when an application or transaction is registered.

5.2 Context Properties

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 can also have a URI property and a user name property that can be used to help define the context.

Context property names are provided when an application or transaction is registered.

Context property values are provided when an application or transaction instance starts.

6 Transaction Response Time Elements

The use of ARM implies a model in which the total response time of a transaction can be sub-divided into finer grained elements, as shown in Figure 17:.

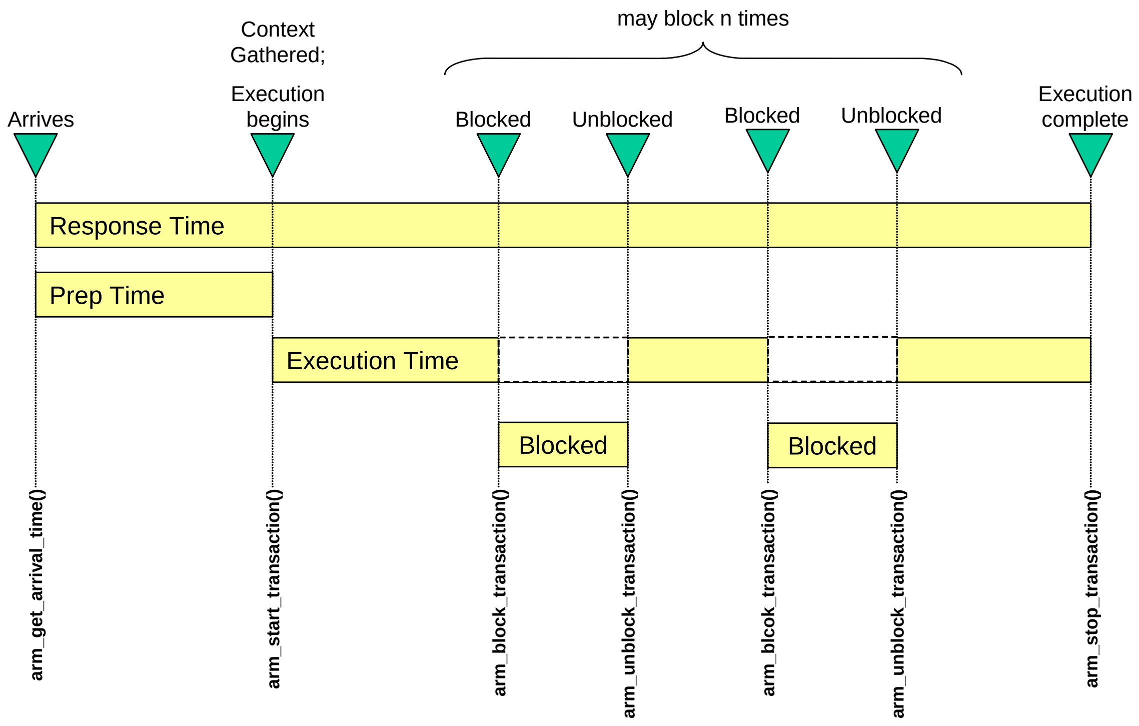

Figure 17: Response Time Elements

- The processing begins when the transaction "arrives"; i.e., when the message (including synchronous RPC invocations) that invokes the transaction is retrieved by the application. The message could have been delayed substantially prior to retrieval, such as a message sitting in a queue.
- In some environments the application first gathers context related to the transaction before it can be processed and/or the *arm_start_transaction*() call made. For example, instrumentation in the Apache web server makes a JNDI call to retrieve some context properties passed on *arm_start_transaction*(). This is shown in the example as Preparation Time. However, because this Prep Time is part of the response time, the application captures a timestamp as soon as it starts executing. It will later provide the timestamp as a parameter on *arm_start_transaction*().
- The transaction begins executing. The application calls *arm_start_transaction*() to indicate the fact. For environments that do not have any Prep Time, the actual time that

arm_start_transaction() is called is used as the time that the response time measurement starts.

- The transaction may block one or more times waiting for an external event, such as a database call or a call to another application program. It indicates the beginning and end of each period when it is blocked with *arm_block_transaction*() and *arm_unblock_transaction*(), respectively.
- The transaction completes executing and indicates the same to ARM with *arm_stop_transaction*(). The actual time that *arm_stop_transaction*() is called is used as the time that the response time measurement ends.

6.1 Arrival and Preparation Time

ARM 4.0 specified one way to indicate the duration of the Prep Time. With ARM 4.1, there are three ways to indicate the duration of the Prep Time.

6.1.1 Opaque Timestamp (ARM 4.0)

Use *arm_get_arrival_time*() as shown in the example in Figure 17:. *arm_get_arrival_time*() returns an opaque timestamp that the application provides with *arm_start_transaction*() in the Arrival Time sub-buffer (see Section 13.3).

6.1.2 Formatted Timestamp (ARM 4.1)

Capture the arrival time using some unspecified means, convert it into a format that ARM recognizes, and provide it with *arm_start_transaction*() in either the Formatted Arrival Time MsecJan1970 sub-buffer (see Section 13.10) or the Formatted Arrival Time Strings sub-buffer (see Section 13.11).

6.1.3 Measured Prep Time (ARM 4.1)

Measure the Prep Time (or a mean over several transaction instances) as a duration and provide it with *arm_start_transaction*() in the Preparation Time or Preparation Statistics sub-buffer (see Sections 13.12 and 13.13).

6.2 Blocked Time

A blocking condition is one in which an application suspends execution of a transaction while awaiting the completion of an external event, such as the completion of a database query or other service request. If an application makes a service request but is able to continue executing other logic, it is not in a blocked condition. After completing the other logic it may enter a blocked condition if it is still waiting for the previous service request to complete before continuing.

The application indicates that it is blocked and unblocked using *arm_block_transaction*() and *arm_unblock_transaction*(). The moment at which either call is made is considered the time when the blocked condition begins or ends.

Beginning with ARM 4.1, the application may optionally provide information indicating whether the blocking event is a synchronous or an asynchronous event, plus some additional information describing the cause of the blocking condition. This data is passed in the Block Cause sub-buffer (see Section 13.7) on *arm_block_transaction*().

6.3 Thread Binding

Independent of blocking conditions, it can be useful to know which threads are executing which transactions. The thread binding could be useful for managing computing resources at a finer level of granularity than a process.

This information is not readily apparent to the operating system because it does not inspect the context of each thread and does not know the association of a thread to a transaction measured with ARM. The application does know the binding and can indicate the same to ARM. There are two ways to indicate the binding:

- *arm_bind_thread*() indicates that the thread from which it is called is performing on behalf of the transaction identified by the start handle. 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.
- The *ARM_FLAG_BIND_THREAD* flag of *arm_start_transaction*() *may be* set to 1 if the started transaction is bound to the executing 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.

There can be any number of threads simultaneously bound to the same transaction.

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

7 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 whether 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.
- It can be useful to know additional context information about a transaction, especially if it fails. For example, instrumentation about a servlet might provide the following diagnostic properties when there is a failure: query string, remote host, remote address, remote user, protocol, request attributes, request header, request parameters, etc.

ARM provides four ways for applications to provide these types of data. The use of either is *optional*.

- One is the identity and context properties described in Chapter 5.
- One is a set of short numeric or character strings named "metrics" in ARM. They are described in Section 7.1.
- One is a long character string, named "diagnostic detail". It is described in Section 7.2.

- One is a set of properties, named “diagnostic properties”. They are described in Section 7.3.

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.

7.1 Metrics

Metrics are provided for transaction instances.

- Each metric has a format and name. These are provided when a transaction is registered and do not change afterwards.
- The metric values may vary per transaction instance.
- The metric values are provided at any or all of when a transaction starts, stops, or is updated between the start and stop.

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

7.1.1 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) RFC 1155 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*)

7.1.2 Gauges

A gauge value can go up and down, and it can be positive or negative. This is the IETF RFC 1155 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*)

7.1.3 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*)

7.1.4 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*)

7.2 Diagnostic Detail

Diagnostic detail is provided for transaction instances.

- The diagnostic detail is in the form of a long null-terminated character string.
- The diagnostic detail string is provided when a transaction instance stops.

- There are no constraints on the contents of the 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 application may provide either Diagnostic Detail OR Diagnostic Properties for any transaction instance, but not both.

7.3 Diagnostic Properties (ARM 4.1)

Diagnostic properties are provided for transaction instances.

- Each property consists of a name and a value, both strings.
- Both names and values are provided when a transaction instance stops.
- Both names and values may vary per transaction instance. They are particularly useful in some middleware environments in which the property names used by hosted applications are not known when the middleware starts executing.

The application may provide either Diagnostic Detail OR Diagnostic Properties for any transaction instance, but not both.

8 API Overview

8.1 Overall API Structure

Figure 18: 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). The valid sub-buffers vary depending on which call is being made.

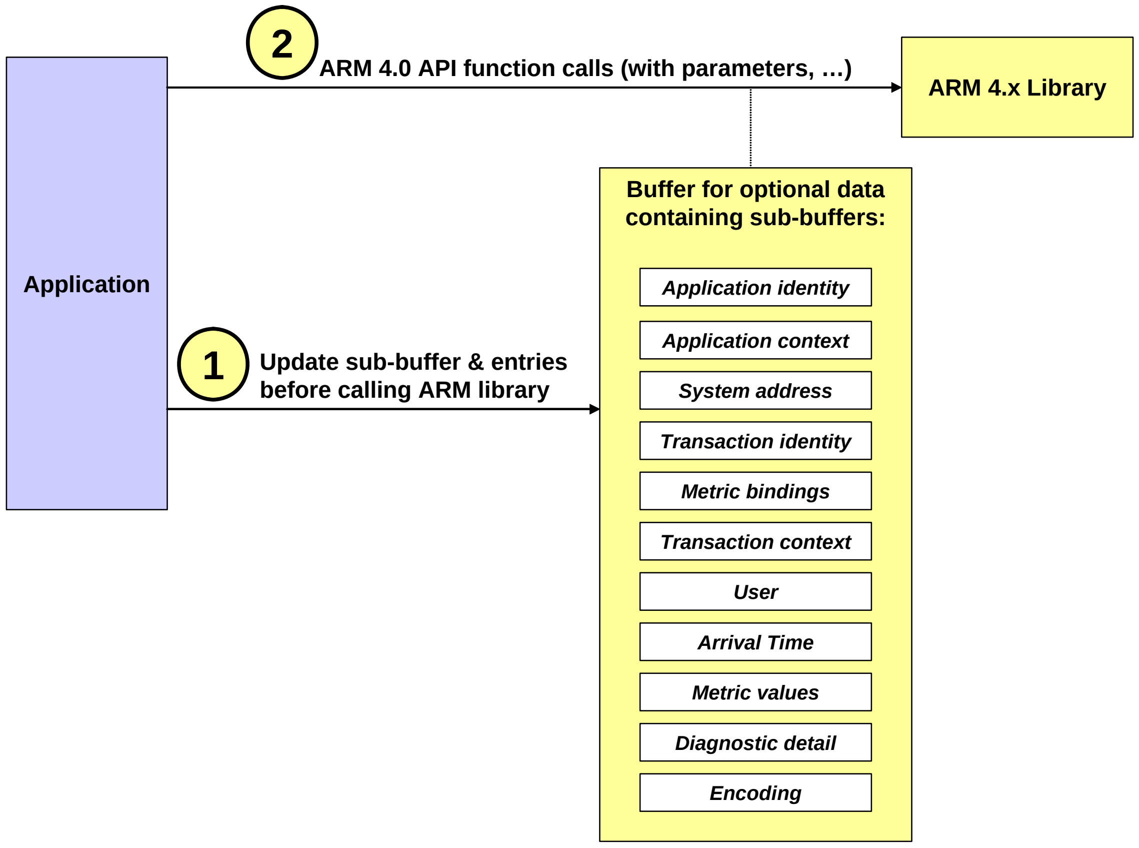

Figure 18: Overall API Structure

8.2 Structure of Optional Buffer and Sub-Buffers

Figure 19 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.

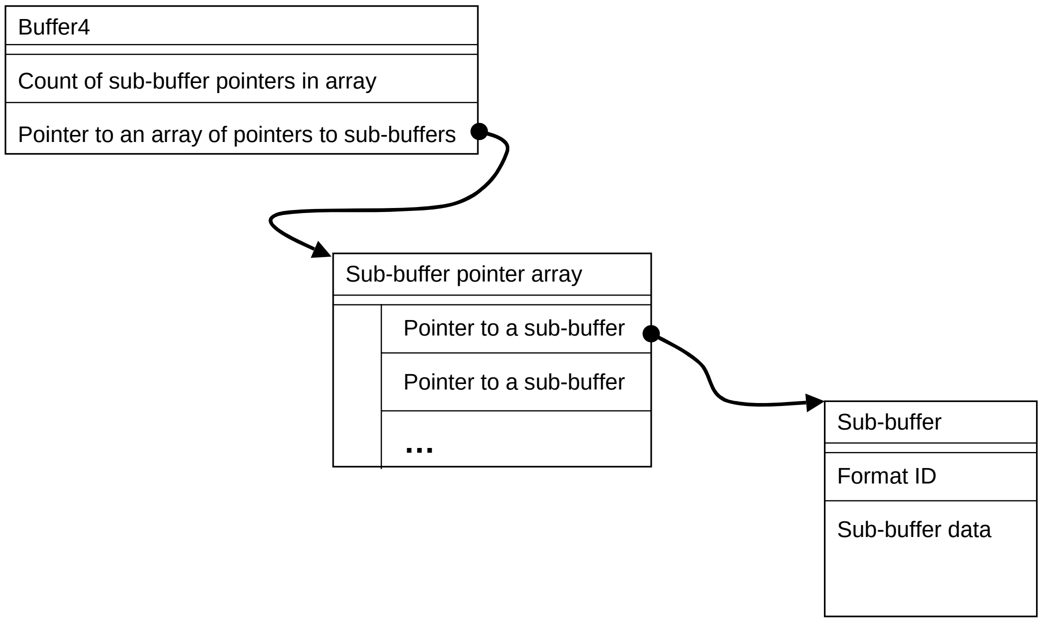

Figure 19: Structure of Optional Buffer and Sub-Buffers

8.3 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.

8.4 Byte Order Markers in Character Strings

Applications must not pass 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.

8.5 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 20 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.

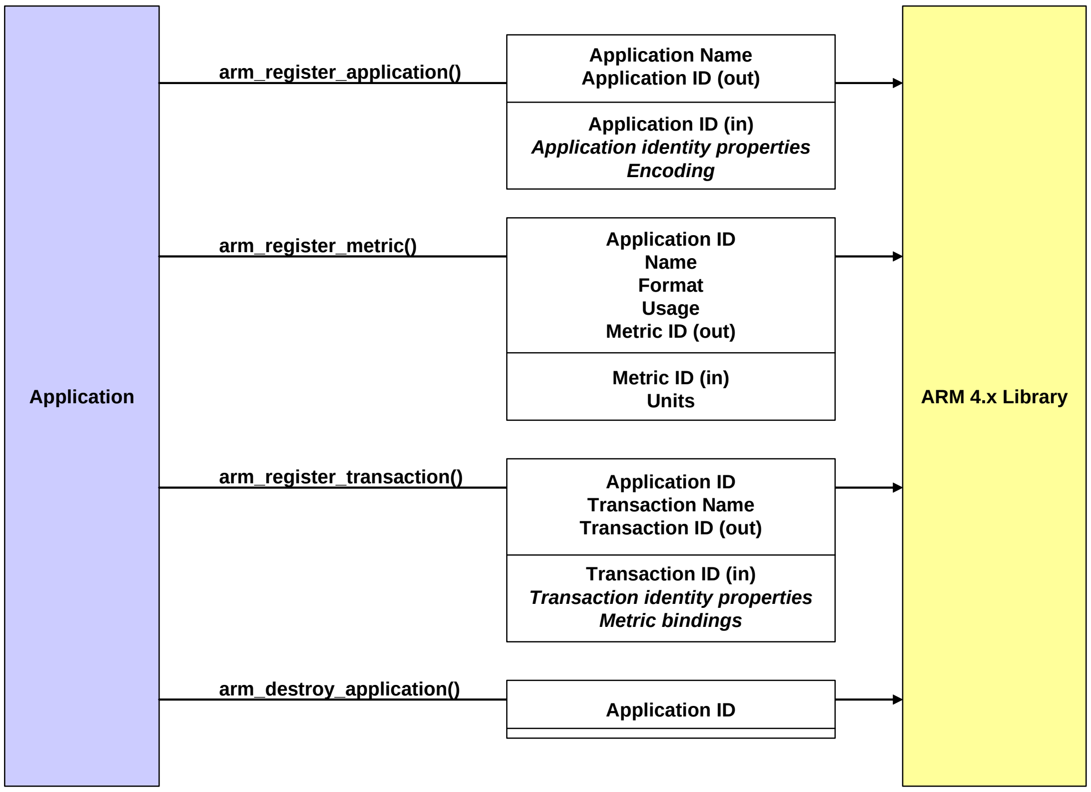

Figure 20: API Functions to Register Metadata

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 optionally 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.

8.6 Overview of API Functions for Application Starts/Stops

Figure 21: lists the functions used to indicate when an instance of an application has started or is stopping. They are performed after the metadata has been registered. The application instance must be started before transactions can be measured.

- 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.

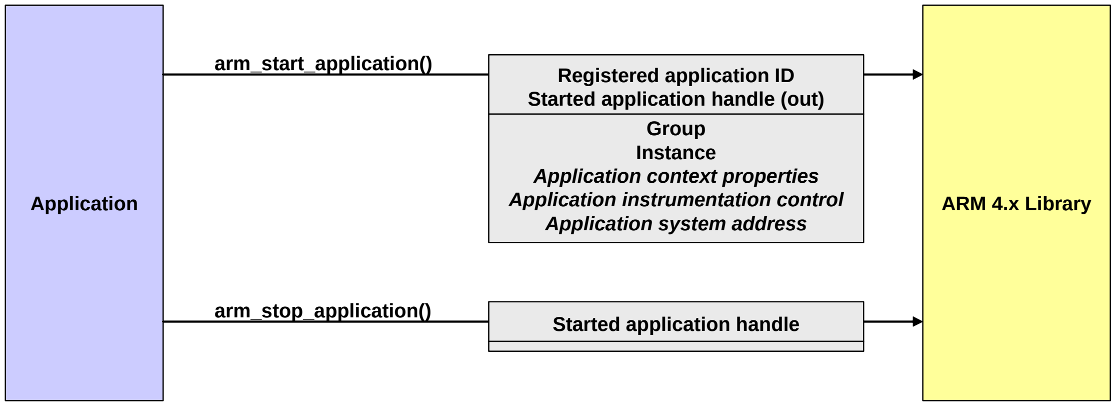
Application
arm_start_application()
Registered application ID
Started application handle (out)
Group
Instance
Application context properties
Application instrumentation control
Application system address
arm_stop_application()
Started application handle
ARM 4.x Library

Figure 21: API Functions for Application Starts and Stops

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).

8.7 Overview of Common API Functions to Measure Transactions

Figure 22 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.

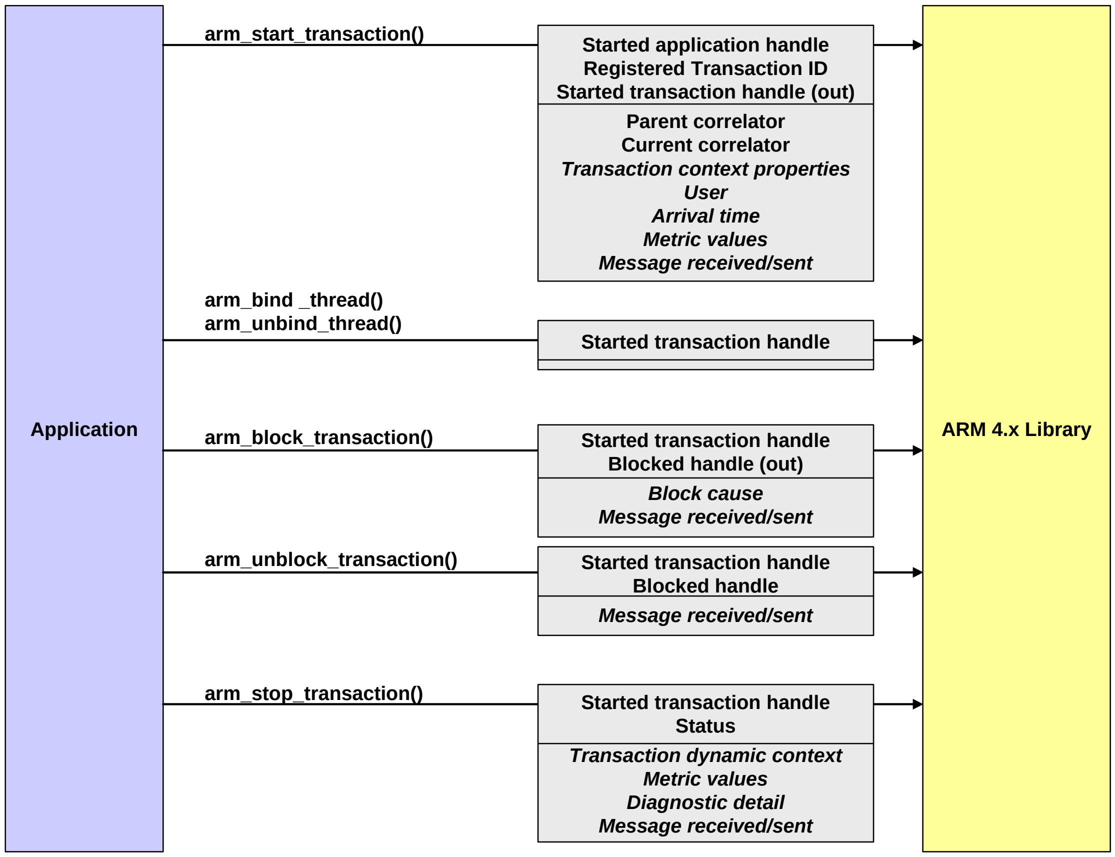

Figure 22: Most Frequently Used API Functions to Measure Transactions

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.

8.8 Overview of Other API Functions

In addition to the functions in Figure 22, 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. It is also used for a special purpose – to enable an application and an ARM implementation to exchange information indicating the level of instrumentation that the implementation prefers.

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, as a way to provide message event sub-buffers, and as a way to pass metric or diagnostic data while a transaction executes. This was described in Section 3.1.

8.9 Allowable Sub-Buffer Use per API Function

Table 6: Sub-Buffer Usage per API Function

API Function	**Allowable Sub-Buffers**
arm_bind_thread()	
arm_block_transaction()	Block Cause (ARM 4.1) Message Received Event (ARM 4.1) Message Sent Event (ARM 4.1)
arm_destroy_application()	
arm_discard_transaction()	
arm_generate_correlator()	User Transaction Context Application Control (ARM 4.1) Transaction ID Control (ARM 4.1) Transaction Instance Control (ARM 4.1)
arm_get_arrival_time()	
arm_get_correlator_flags()	
arm_get_correlator_length()	
arm_get_error_message()	
arm_is_charset_supported()	
arm_register_application()	Application Identity Character Set Encoding
arm_register_metric()	
arm_register_transaction()	Transaction Identity Metric Bindings
arm_report_transaction()	User Metric Values Diagnostic Detail Transaction Context Diagnostic Properties (ARM 4.1)
arm_start_application()	System Address Application Context Values Application Control (ARM 4.1)

API Function	Allowable Sub-Buffers
arm_start_transaction()	User Arrival Time Metric Values Message Received Event (ARM 4.1) Message Sent Event (ARM 4.1) Formatted Arrival Time MsecJan1970 (ARM 4.1) Formatted Arrival Time Strings (ARM 4.1) Preparation Time (ARM 4.1) Preparation Statistics (ARM 4.1) Transaction Context
arm_stop_application()	
arm_stop_transaction()	Metric Values Diagnostic Detail Message Received Event (ARM 4.1) Message Sent Event (ARM 4.1) Diagnostic Properties (ARM 4.1)
arm_unbind_thread()	
arm_unblock_transaction()	Message Received Event (ARM 4.1) Message Sent Event (ARM 4.1)
arm_update_transaction()	Metric Values Message Received Event (ARM 4.1) Message Sent Event (ARM 4.1)

8.10 Processing Multiple Values of the Same Metric

Additional semantics are defined when using *arm_start_transaction*(), *arm_update_transaction*(), and *arm_stop_transaction*() 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.

8.10.1 Counters

If a counter is used, its initial value must be set at the time of the *arm_start_transaction*() call. The difference between the value when the *arm_start_transaction*() executes and when the *arm_stop_transaction*() executes (or the value in the last *arm_update_transaction*() if no metric value is passed on *arm_stop_transaction*()) is the value attributed to this transaction. Similarly, the difference between successive *arm_update_transaction*()s, or from the *arm_start_transaction*() to the first *arm_update_transaction*(), or from the last *arm_update_transaction*() to the *arm_stop_transaction*(), 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 *arm_start_transaction*() and to some value at *arm_stop_transaction*() (or the last *arm_update_transaction*()). In this case, the application probably measured the value for this transaction and provided that value in the *arm_stop_transaction*(). The application always sets the value to zero at the *arm_start_transaction*() so the value at *arm_stop_transaction*() reflects both the difference from the *arm_start_transaction*() 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 *arm_start_transaction*(), *x*2 at its corresponding *arm_stop_transaction*(), *x*2 at the next *arm_start_transaction*(), and *x*3 at its corresponding *arm_stop_transaction*(). 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 *arm_start_transaction*(), *x*2 at *arm_stop_transaction*(), *x*3 (not equal to *x*2) at the next *arm_start_transaction*(), and *x*4 at its *arm_stop_transaction*(). In this case, the application is probably keeping a rolling counter as in the previous example. But in this case the measurement represents a value affected by other users or transaction classes, so the value often changes from one *arm_stop_transaction*() to the next *arm_start_transaction*() for the same transaction class.

8.10.2 Gauges

Gauges can be set before *arm_start_transaction*(), *arm_update_transaction*(), and *arm_stop_transaction*() calls. This creates the potential for different interpretations. If several values are provided for a transaction [e.g., one at *arm_start_transaction*(), one at each *arm_update_transaction*(), and one at *arm_stop_transaction*()], 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 *arm_start_transaction*() and *arm_stop_transaction*() calls.

- The minimum value for a transaction will be the smallest valid value passed at any time between and including the *arm_start_transaction*() and *arm_stop_transaction*() calls.

- The mean value for a transaction will be the mean of all valid values passed at any time between and including the *arm_start_transaction*() and *arm_stop_transaction*() calls. All valid values will be weighted equally each time a *arm_start_transaction*(), *arm_update_transaction*(), or *arm_stop_transaction*() 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 *arm_start_transaction*(), *arm_update_transaction*(), or *arm_stop_transaction*() executes.

- The last value for a transaction will be the last valid value passed whenever any *arm_start_transaction*(), *arm_update_transaction*(), or *arm_stop_transaction*() executes.

8.10.3 Numeric IDs

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

8.10.4 Strings

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

9 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.

9.1 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.

There is a special case that is described in the 'Special Note' on page 53 in the description of *arm_generate_correlator*(). Because of this special case, implementations are recommended to not use the return code –1012 with *arm_generate_correlator*() unless it has the semantic related to the null or invalid *current_correlator* parameter that is described in the 'Special Note'.

10 Instrumentation Control (ARM 4.1)

ARM is designed to be a high performing interface so applications and middleware can invoke it as often as there is an interesting event to report. The ARM implementation determines whether to process the passed data and the manner of processing. In most cases this is entirely satisfactory and the application does not include conditional logic to determine when to call ARM nor how much data to provide with each call.

Certain applications or middleware may provide fine-grained instrumentation, both in terms of the number of instrumentation points and the depth of data available at each instrumentation point. IBM WebSphere is an example. Beginning with ARM 4.1, ARM provides a mechanism that enables an application or middleware program to query the ARM implementation for the desired instrumentation granularity.

The mechanism works as shown in Figure 23:. Its use is optional for both instrumented applications and ARM implementations, and there is a handshake so an application knows whether to depend on the capability. There are three control scopes: application-wide for all transactions, application-wide for all transactions of a specified ID, and each individual transaction instance. There is a code sample using all three scopes in Figure 16.4.

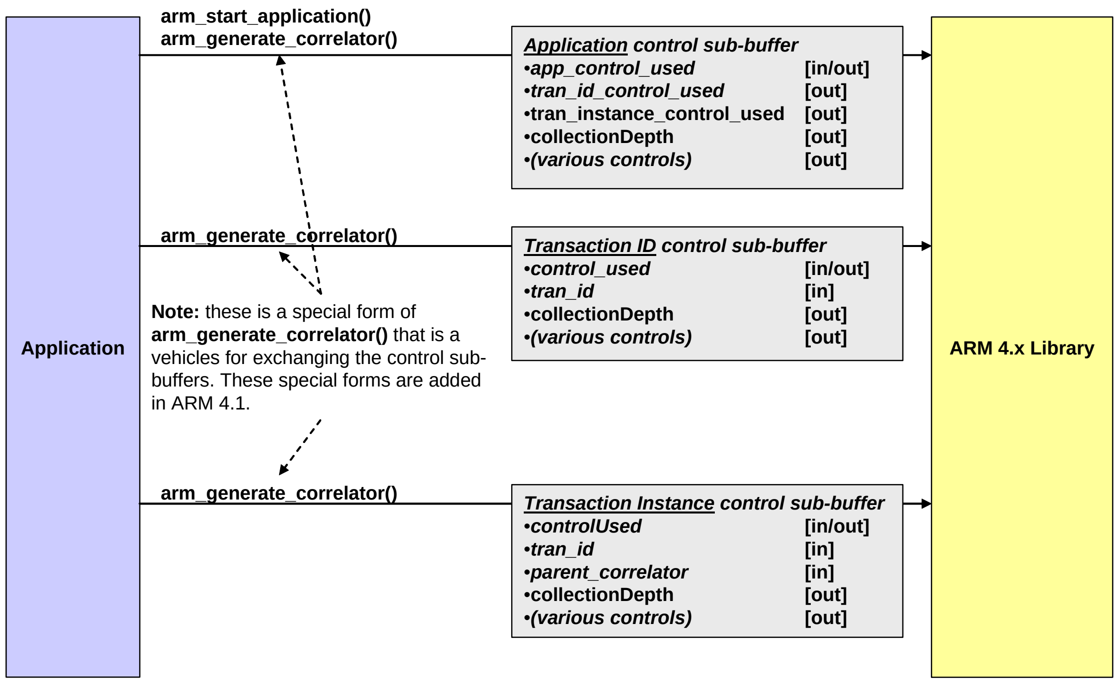
Application
arm_start_application()
arm_generate_correlator()
Application control sub-buffer
•app_control_used [in/out]
•tran_id_control_used [out]
•tran_instance_control_used [out]
•collectionDepth [out]
•(various controls) [out]
arm_generate_correlator()
Transaction ID control sub-buffer
•control_used [in/out]
•tran_id [in]
•collectionDepth [out]
•(various controls) [out]
Note: these is a special form of arm_generate_correlator() that is a vehicles for exchanging the control sub-buffers. These special forms are added in ARM 4.1.
arm_generate_correlator()
Transaction Instance control sub-buffer
•controlUsed [in/out]
•tran_id [in]
•parent_correlator [in]
•collectionDepth [out]
•(various controls) [out]
ARM 4.x Library

Figure 23: Instrumentation Control APIs

10.1 Scope: Application-Wide for all Transactions

An application determines whether instrumentation control is being used, and if so, establishes the instrumentation level for the entire application, by passing the Application Control sub-buffer when it starts, using *arm_start_application*(). It sets the *app_control_used* flag to False and inspects this flag upon return from the API call.

- If the flag is still False, then the ARM implementation is not using instrumentation control and the application uses its default instrumentation settings.
- If the flag is True, then the ARM implementation is using instrumentation control and the other control settings have been set to meaningful values. Three important settings are:

  — *tran_id_control_used*, which indicates whether the control settings may vary per transaction ID

  — *tran_instance_control_used*, which indicates whether the control settings may vary per transaction instance

  — *collection_depth*, which sets the default level (None, Process, Container, Maximum) for all transactions in the process

The application may pass the Application Control sub-buffer periodically after starting to see if any control settings have changed. For example, the settings might be re-queried every 1000 transactions or every 15 minutes. This is a good practice, especially if the instrumentation level has previously been set to None. The check is done using a special form of *arm_generate_correlator*(), with the current correlator set to NULL to indicate that no correlator is to be generated.

10.2 Scope: All instances of a Registered Transaction ID

If the ARM implementation returned *tran_id_control_used*=True in the Application Control sub-buffer, then the application is invited to request the instrumentation controls for every instance of a registered transaction ID. It does this by passing the Transaction ID Control sub-buffer using the special form of *arm_generate_correlator*(), with the current correlator set to NULL to indicate that no correlator is to be generated. These settings override the application-wide settings. It is a good practice to periodically check to see if any control settings have changed, especially if the instrumentation level has previously been set to None. For example, the settings might be re-queried every 1000 transactions or every 15 minutes.

The *control_used* setting in the Transaction ID Control sub-buffer provides the same handshake that *app_control_used* provides in the Application Control sub-buffer. The application sets the flag to False and tests the value upon return. If the value is still False, then the implementation has provided no guidance and the application uses its current defaults for transactions of this type. If the value has been changed to True, then the other values set by the ARM implementation are used.

The settings are valid for every instance of the registered transaction ID unless the value is overridden by a setting in the Transaction Instance Control sub-buffer.

10.3 Scope: One Transaction Instance

If the ARM implementation returned *tran_instance_control_used*=True in the Application Control sub-buffer, then the application is invited to request the instrumentation controls for each instance of a transaction. It does this by passing the Transaction Instance Control sub-buffer using the special form of *arm_generate_correlator*(), with the current correlator set to NULL to indicate that no correlator is to be generated, prior to calling *arm_start_transaction*().

The *control_used* setting in the Transaction Instance Control sub-buffer provides the same handshake that *app_control_used* provides in the Application Control sub-buffer. The application sets the flag to False and tests the value upon return. If the value is still False, then the implementation has provided no guidance and the application uses its current defaults for transactions of this type. If the value has been changed to True, then the other values set by the ARM implementation are used.

The settings are valid for exactly one instance of a transaction. They do not carry over to other instances or change the defaults set with the Application Control or Transaction ID Control sub-buffers.

11 API Macros

This chapter describes macros that are provided with the specification to encapsulate details that should be transparent to programmers using ARM.

ARM_SET_CORRELATOR_FLAG()

NAME

ARM_SET_CORRELATOR_FLAG – set value of flag

SYNOPSIS

```
ARM_SET_CORRELATOR_FLAG(corr, flag_num, boolean_value)
```

DESCRIPTION

The macro sets the Boolean value of one of the flags in the correlator pointed to by *corr*. If the application intends to set both the Asynchronous Flow and Independent Transaction flags, it would use the macro twice. Since this is a macro, compile-time type checking might not apply. Two flags can be set with the macro:

- Asynchronous Flow (see Section 4.2.1)
- Independent Transaction (see Section 4.2.2)

PARAMETERS

corr Pointer to a correlator. The type of the pointer must be const **arm_correlator_t ***.

flag_num Enumerated value indicating the flag to be changed. The following values are allowed:

3 = ARM_CORR_FLAGNUM_ASYNCH = mark asynchronous control flow; see Section 4.2.1

4 = ARM_CORR_FLAGNUM_INDEPENDENT = mark child transaction as independent; see Section 4.2.2

boolean_value
The value to set the flag to. Must be ARM_TRUE or ARM_FALSE.

12 The API Functions

This chapter defines the ARM 4.1 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 optional 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 whether 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,
    /*[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.

A description of the blocking cause can be provided in the optional Block Cause sub-buffer.

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 optional buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_block_cause_t
arm_subbuffer_message_rcvd_event_t
arm_subbuffer_message_sent_event_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 whether 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 that 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 optional 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 whether 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 optional 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 whether 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 for two very different purposes:

1. It is used to generate a correlator for use with *arm_report_transaction*().
2. It is used (beginning with ARM 4.1) to exchange control information that indicates the level of instrumentation for this transaction that is desired by the ARM implementation.

Generating a Correlator

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 is no need for a start handle, and there is 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.

Controlling the Level of Instrumentation for a Transaction

There is a special use of *arm_generate_correlator*() that is not intuitive because it is unrelated to generating a correlator. It is used because this allowed a new capability to be added without adding any new function calls, which would have impacted backwards-compatibility with ARM libraries that support ARM 4.0 but not ARM 4.1 or later ARM 4.x versions. The use is described in Chapter 10. Further details are available in the descriptions of the Application Control sub-buffer (see Section 13.21) and Transaction Instance Control sub-buffer (see Section 13.22).

When used for this special purpose, the meaningful parameters are:

```
app_handle
buffer4 → arm_subbuffer_app_control_t,
    arm_subbuffer_tran_id_control_t, or
    arm_subbuffer_tran_instance_control_t
current_correlator = NULL
```

This special form is recognized by setting the *current_correlator* pointer to null, which renders the call meaningless for the purpose of generating a correlator.

Note: Some ARM 4.0 implementations check for and do not expect to find a null current correlator value in *arm_generate_correlator*(), and these implementations set the return code to –1012 if they are called with a null or invalid *current_correlator* pointer. If the application is using *arm_generate_correlator*() for purposes of instrumentation control and has therefore set *current_correlator* to null, and the return value from *arm_generate_correlator*() = -1012, it is likely that the implementation does not recognize the use of *arm_generate_correlator*() for instrumentation control, and the application should proceed accordingly. The suggested course of action is to discontinue using *arm_generate_correlator*() for instrumentation control.

PARAMETERS

app_handle

The value returned from an *arm_start_application*() call in the same process. The ARM implementation may use this handle to access application instance data that may become part of the correlator; e.g., the **arm_subbuffer_system_address_t**.

buffer4 Pointer to the optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_app_control_t
arm_subbuffer_tran_context_t
arm_subbuffer_tran_id_control_t
arm_subbuffer_tran_instance_control_t
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) 512 (ARM_CORR_MAX_LENGTH). The value must be non-null if *arm_generate_correlator*() is being used to generate a correlator. If *arm_generate_correlator*() is being used to pass the Application Control, Transaction ID Control, or Transaction Instance Control sub-buffer, then *current_correlator* must be set to null.

flags Contains 32-bit flags.

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

0x00000004 (ARM_FLAG_CORR_IN_PROCESS) is set to 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 (ARM_CORR_NONE). If *arm_generate_correlator*() is being used for purposes of instrumentation control, *parent_correlator* is ignored.

tran_id A transaction ID returned in an *out* parameter from an *arm_register_transaction*() call in the same process. If *arm_generate_correlator*() is being used for purposes of instrumentation control, *tran_id* is ignored.

RETURN VALUE

The returned code is a status that may indicate whether 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 **arm_int64_t** 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 whether 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 (ARM_CORR_FLAGNUM_APP_TRACE) = Application trace flag

  2 (ARM_CORR_FLAGNUM_AGENT_TRACE) = Agent trace flag

  3 (ARM_CORR_FLAGNUM_ASYNCH) = mark asynchronous control flow; see Section 4.2.1

  4 (ARM_CORR_FLAGNUM_INDEPENDENT) = mark child transaction as independent; see Section 4.2.2

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 whether 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_start_transaction*()

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 **arm_correlator_length_t** (a 16-bit integer) 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 512 (ARM_CORR_MAX_LENGTH) bytes.

RETURN VALUE

The returned code is a status that may indicate whether 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_start_transaction*()

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 contains 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 (i.e., 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 whether 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_get_error_message(), *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 7 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 & Linux	Microsoft Windows	IBM I5/OS	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 7: 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 whether 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_get_error_message(), *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. A name should be chosen that is unique, so generic names that might be used by a different development team, such as “Payroll Application”, should not be used. The name should not contain trailing blank characters or consist of only blank characters. If the application has a copyrighted product name, the copyrighted name would be a good choice.

buffer4 Pointer to the optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffer formats that might be used are:

arm_subbuffer_app_identity_t
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 128-bit ID (16 bytes) 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 (ARM_ID_NONE), 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. 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 whether 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_destroy_application(), *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 optional 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:

1 (ARM_METRIC_FORMAT_COUNTER32) = **arm_int32_t** counter

2 (ARM_METRIC_FORMAT_COUNTER64) = **arm_int64_t** counter

3 (ARM_METRIC_FORMAT_CNTRDIVR) = **arm_int32_t** counter + **arm_int32_t** divisor

4 (ARM_METRIC_FORMAT_GAUGE32) = **arm_int32_t** gauge

5 (ARM_METRIC_FORMAT_GAUGE64) = **arm_int64_t** gauge

6 (ARM_METRIC_FORMAT_GAUGEDIVR32) = **arm_int32_t** gauge + **arm_int32_t** divisor

7 (ARM_METRIC_FORMAT_NUMERICID32) = **arm_int32_t** numeric ID

8 (ARM_METRIC_FORMAT_NUMERICID64) = **arm_int64_t** 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 128-bit ID (16 bytes) 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 (ARM_ID_NONE), 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. The name should not contain trailing blank characters or consist of only blank characters.

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. 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 whether 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_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 optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_metric_bindings_t
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*(), *arm_report_transaction*(), or *arm_generate_correlator*().

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 128-bit ID (16 bytes) 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 (ARM_ID_NONE), 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. 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. The name should not contain trailing blank characters or consist of only blank characters.

RETURN VALUE

The returned code is a status that may indicate whether 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.

If the transaction represented by the *arm_report_transaction*() call is the correlation parent of another transaction, *arm_generate_correlator*() must be used to get a parent correlator, prior to invoking the child transaction (because it must be passed to the child). In this case, the sequence is the following:

1. *arm_generate_correlator*() to get a correlator for this transaction.
2. Invoke the child transaction, passing the correlator returned in step (1) to the child.
3. When this transaction is complete, invoke *arm_report_transaction*() to report the statistics about the transaction.

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 optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_diag_detail_t
arm_subbuffer_diag_properties_t
arm_subbuffer_metric_values_t
arm_subbuffer_tran_context_t
arm_subbuffer_user_t

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

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

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

response_time
Response time in nanoseconds.

stop_time An **arm_int64_t** 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, –1 (ARM_USE_CURRENT_TIME), 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_ABORTE*D*) = 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 whether 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_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.

ARM 4.1 added a new capability for controlling the level of instrumentation that the ARM implementation prefers. The capability is described in Chapter 10. It uses the Application Control sub-buffer described in Section 13.21.

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. If the System Address sub-buffer is provided, it is meaningless to use *arm_start_transaction*() or *arm_stop_transaction*(), or any of the API calls that are used after *arm_start_transaction*() and before *arm_stop_transaction*().

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. Application instances for a given software product that are started for a common runtime purpose are typically very good candidates for using the same group name. For example, identical replica instances of a product started across multiple processes or servers to address a specific transaction workload objective can be, advantageously to the ARM agent, commonly identified by the group name. The maximum length of the string is 256 (ARM_PROPERTY_VALUE_MAX_CHARS) characters, including the termination character. A null pointer indicates that there is no group.

app_handle Pointer to an **arm_int64_t** into which the ARM library will store the value of the handle that will represent the application instance in all calls, up to and including the *arm_stop_application*() that indicates that the instance has completed executing. The scope of the handle is the process in which the *arm_start_application*() is executed. 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. A null pointer indicates that there is no instance value.

buffer4 Pointer to the optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffer formats that might be used are:

arm_subbuffer_app_context_t
arm_subbuffer_app_control_t
arm_subbuffer_system_address_t

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

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

If a System Address 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 Address 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 whether 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_stop_application*()

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 optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_arrival_time_t
arm_subbuffer_prep_stats_t
arm_subbuffer_prep_time_t
arm_subbuffer_formatted_arrival_time_msecJan1970_t
arm_subbuffer_formatted_arrival_time_strings_t
arm_subbuffer_message_rcvd_event_t
arm_subbuffer_message_sent_event_t
arm_subbuffer_metric_values_t
arm_subbuffer_tran_context_t
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) 512 (ARM_CORR_MAX_LENGTH) bytes.

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:

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

0x00000002 (ARM_FLAG_BIND_THREAD) is set to 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.

0x00000004 (ARM_FLAG_CORR_IN_PROCESS) is set to 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 **arm_int64_t** 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_arm_update_transaction*()].

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 whether 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_transaction*()

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 an *out* parameter from an *arm_start_application*() call in the same process.
buffer4	Pointer to the optional 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 whether 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 optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_diag_detail_t
arm_subbuffer_diag_properties_t
arm_subbuffer_message_rcvd_event_t
arm_subbuffer_message_sent_event_t
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 whether 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.
- 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 optional 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 whether 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 optional buffer, if any. If the pointer is null, there is no buffer. The sub-buffers that may be used are:

arm_subbuffer_message_rcvd_event_t
arm_subbuffer_message_sent_event_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 whether 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 optional buffer, if any. If the pointer is null (ARM_BUF4_NONE), there is no buffer. The sub-buffers that might be used are:

arm_subbuffer_message_rcvd_event_t
arm_subbuffer_message_sent_event_t
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 whether 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_stop_transaction*()

13 Optional Buffer and Sub-Buffers

13.1 Optional 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 optional 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 19.)

- **Count of sub-buffer pointers:** An **arm_int32_t** count of sub-buffers in the following array.
- **Array of pointers to sub-buffers:** A pointer to an array of pointers to 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 **arm_int32_t** sub-buffer format ID.

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

ID	arm41.h Constant, if Applicable	Usage Description
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
8	ARM_SUBBUFFER_BLOCK_CAUSE	Block Cause (ARM 4.1)
9	ARM_SUBBUFFER_MESSAGE_RCVD_ EVENT	Message Received Event (ARM 4.1)
10	ARM_SUBBUFFER_MESSAGE_SENT_ EVENT	Message Sent Event (ARM 4.1)
11	ARM_SUBBUFFER_FORMATTED_ ARRIVAL_TIME_USECJAN1970	Formatted Arrival Time UsecJan1970 (ARM 4.1)
12	ARM_SUBBUFFER_FORMATTED_ ARRIVAL_TIME_STRINGS	Formatted Arrival Time Strings (ARM 4.1)
13	ARM_SUBBUFFER_PREP_TIME	Preparation Time (ARM 4.1)
14	ARM_SUBBUFFER_PREP_STATS	Preparation Statistics (ARM 4.1)
15	ARM_SUBBUFFER_DIAG_PROPERTIES	Diagnostic Properties (ARM 4.1)
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

ID	arm41.h Constant, if Applicable	Usage Description
108	ARM_SUBBUFFER_APP_CONTROL	Application Control (ARM 4.1)
109	ARM_SUBBUFFER_TRAN_ID_CONTROL	Transaction ID Control (ARM 4.1)
110	ARM_SUBBUFFER_TRAN_INSTANCE_ CONTROL	Transaction Instance Control (ARM 4.1)
30000 – 30999		Reserved for IBM

Table 8: Sub-Buffer Formats

All negative values are available for implementation-specific purposes. Some negative values are known to be in use or planned for future use so users of the ARM interface are advised to avoid them to avoid conflicts:

Value Range	User
–30999 : –30000	IBM

Table 9: Known Implementation-Specific Sub-Buffer Format IDs

- **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 24.

— The sub-buffer contains one or more arrays of elements. In this case, the sub-buffer is in the following format. See Figure 25.

— 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

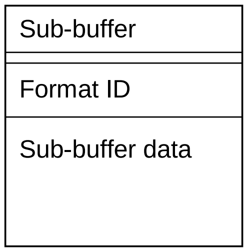

Figure 24: Format of Sub-Buffer that does not Contain an Array

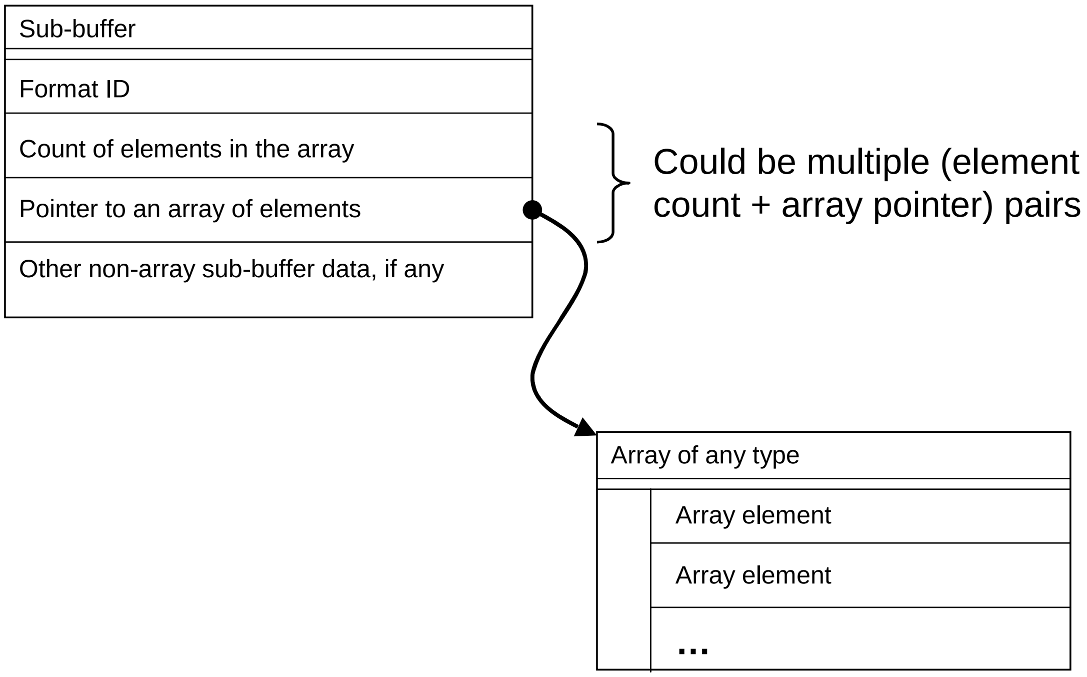
Sub-buffer
Format ID
Count of elements in the array
Pointer to an array of elements
Other non-array sub-buffer data, if any
Could be multiple (element count + array pointer) pairs
Array of any type
Array element
Array element
...

Figure 25: Format of Sub-Buffer that Contains One or More Arrays of Data

13.2 User

(*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 24:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=3 (ARM_SUBBUFFER_USER).
- **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 (16 bytes) that is associated with and can be used as an alias for the user name.

13.3 Arrival Time

(*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. This scenario is described in Chapter 6.

When the application can call ARM when the transaction starts but not call *arm_start_transaction*() because it does not yet have all the data it needs, 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 the Arrival Time sub-buffer, to indicate when the transaction actually started executing. This scenario is described in Section 6.1.1.

The arrival time field is a 64-bit integer containing the value returned by the *arm_get_arrival_time*() function. The format of the data within the **arm_int64_t** is implementation-defined.

There are five sub-buffers (listed below) that are used for essentially the same purpose – to provide a more accurate start time than can be achieved with *arm_start_transaction*() alone. If more than one of these sub-buffers is passed, the ARM implementation will select which one to use, such as a heuristic that it will use the first sub-buffer in the list that it recognizes and ignore all others. Under no circumstances should the ARM implementation use data from more than one of the sub-buffers.

- Arrival Time sub-buffer (which contains an opaque formatted time)
- Formatted Arrival Time MsecJan1970 sub-buffer
- Formatted Arrival Time Strings sub-buffer
- Preparation Time sub-buffer
- Preparation Statistics sub-buffer

Format

The pattern is illustrated in Figure 24:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=4 (ARM_SUBBUFFER_ARRIVAL_TIME).
- **Arrival time:** An **arm_int64_t** value containing an opaque time indicator generated by and recognizable by the ARM implementation.

13.4 Metric Values

(*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 25:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=5 (ARM_SUBBUFFER_METRIC_VALUES).
- **Count of metric value pointers:** An **arm_int32_t** 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 re-used, 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	**arm_int32_t** counter
2	ARM_METRIC_FORMAT_COUNTER64	**arm_int64_t** counter
3	ARM_METRIC_FORMAT_CNTRDIVR	**arm_int32_t** counter + **arm_int32_t** divisor
4	ARM_METRIC_FORMAT_GAUGE32	**arm_int32_t** gauge
5	ARM_METRIC_FORMAT_GAUGE64	**arm_int64_t** gauge
6	ARM_METRIC_FORMAT_GAUGEDIVR32	**arm_int32_t** gauge + **arm_int32_t** divisor
7	ARM_METRIC_FORMAT_NUMERICID32	**arm_int32_t** numeric ID
8	ARM_METRIC_FORMAT_NUMERICID64	**arm_int64_t** numeric ID
9	(DEPRECATED)	
10	ARM_METRIC_FORMAT_STRING32	**arm_char_t*** null-terminated string of a maximum length of 32 characters (33 including the null termination character)

Table 10: Metric Formats

— **Usage:** An **arm_metric_usage_t** indicating how the metric is used. The usage must be the same as the usage parameter passed on the *arm_register_metric*() call that registered the metric ID with the same slot number that is 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.

Table 11: Metric Usage Indicators

— **Valid flag:** A boolean that indicates whether the data in the "Metric value" field is currently valid.

— **Metric value:** A C union containing the metric value. The data type matches the metric format indicator (above).

13.5 System Address

(*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 Address sub-buffer is provided on *arm_start_application*(), all transactions reported by this application instance execute in the current process.
- If a System Address sub-buffer is provided on *arm_start_application*(), all transactions execute in a different process.
- If a System Address 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 Address 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 24:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=6 (ARM_SUBBUFFER_SYSTEM_ADDRESS).
- **Format of the system address:** An **arm_int16_t** format of the system address field. The following formats are defined:

Format	Format Name	Details
0	Reserved	
1	ARM_SYSADDR_FORMAT_IPV4	Bytes 0:3 = 4-byte IP address

Format	Format Name	Details
2	ARM_SYSADDR_FORMAT_IPV4PORT	Bytes 0:3 = 4-byte IP address Bytes 4:5 = 2-byte IP port number
3	ARM_SYSADDR_FORMAT_IPV6	Bytes 0:15 = 16-byte IP address
4	ARM_SYSADDR_FORMAT_IPV6PORT	Bytes 0:15 = 16-byte IP address Bytes 16:17 = 2-byte IP port number
5	ARM_SYSADDR_FORMAT_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	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	Bytes 0:?? = hostname (characters, not null-terminated), where ?? is determined based on the **arm_int16_t** *address_length* field.
8	ARM_SYSADDR_FORMAT_UUID	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.

Table 12: System Address Formats

- **Length of system address:** An **arm_int16_t** length of system address in bytes.

  — There is no maximum length.

  — A length of zero refers to the local system.

- **System address:** A pointer to 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 address 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 the tuple of (format, length, system address). If the “System Address” format is a UUID, and a “System Address ID” is provided, it is a coincidence if the values are identical. Nothing precludes or requires that they be the same.

— 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.

13.6 Diagnostic Detail

(*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 24:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=7 (ARM_SUBBUFFER_DIAG_DETAIL).
- **Value:** An **arm_char_t*** 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.

13.7 Block Cause

[Added in ARM 4.1]

(*format*=8, ARM_SUBBUFFER_BLOCK_CAUSE)

Syntax

```
typedef struct arm_subbuffer_block_cause
{
    arm_subbuffer_t   header;
    arm_block_cause_t cause;
    arm_int32_t       extended_cause ;
    const arm_char_t *description;
} arm_subbuffer_block_cause_t;
```

Description

Applications may indicate that they are blocked waiting on an external event using the *arm_block_transaction*() function. When the application becomes unblocked it indicates the same using *arm_unblock_transaction*(). Calling *arm_stop_transaction*() implicitly executes an *arm_unblock_transaction*() for every blocking condition whose end has not been explicitly indicated using *arm_unblock_transaction*().

Calling *arm_block_transaction*() with no further information does not indicate the cause of the blocking condition. The Block Cause sub-buffer provides a means to indicate the cause of the blocking condition. Its use is optional. It is valid on the following calls:

- *arm_block_transaction*()

In the case of application servers that loop on queued message requests, it is important to distinguish between the *application* being blocked and a *message* being blocked (or simply not available yet). *arm_block_transaction*() and arm*_unblock_transaction*() are used only within the scope of paired *arm_start_transaction*() and *arm_stop_transaction*() calls that indicate the time that work is being processed. If the application server is idle waiting for work to process it would not report this time using *arm_block_transaction*() and *arm_unblock_transaction*().

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=8 (ARM_SUBBUFFER_BLOCK_CAUSE).
- **Block cause**: An **arm_block_cause_t** value that indicates the cause of the blocking condition. All non-negative values are defined by or reserved by this specification.

Value	#define Name	Description
1	ARM_BLOCK_CAUSE_SYNCHRONOUS_ EVENT	A synchronous call, such as an RPC (Remote Procedure Call) has been made and the application is awaiting a response before proceeding.
2	ARM_BLOCK_CAUSE_ASYNCHRONOUS_ EVENT	The application has invoked an asynchronous transaction and/or is involved in a conversation that consists of exchanging messages. It is currently waiting for another message or prior message response or some asynchronous event before proceeding with the execution of the current transaction.
All other values		Reserved for future use. All implementations should accept these values and ignore them if the meaning is unknown. This enables compatibility in future versions of the ARM standard if some of these values are defined.

Table 13: Block Causes

- **Extended Block Cause:** This is an optional value that can be used to further qualify the type of blocking cause within the two possible blocking types: synchronous and asynchronous.

Value	#define Name	Description
All non-negative values	N/A	Reserved by the standard. It is anticipated that ranges will be assigned to organizations that instrument their software.
Negative values	N/A	The extended blocking cause is not defined by the standard. There are no restrictions on who uses these values or what they mean.

Table 14: Extended Block Causes

- **Description:** This is an optional null-terminated character string with a maximum length of 128 characters, including the termination character, which describes the block cause. A null value indicates that no description is provided.

13.8 Message Received Event

[Added in ARM 4.1]

(*format*=9, ARM_SUBBUFFER_MESSAGE_RCVD_EVENT)

Syntax

```
typedef struct arm_subbuffer_message_rcvd_event
{
    arm_subbuffer_t                  header;
    arm_boolean_t                    end_of_flow;
    arm_int32_t                      event_count;
    const arm_message_rcvd_event_t *message_event_array;
} arm_subbuffer_message_rcvd_event_t;

typedef struct arm_message_rcvd_event
{
    const arm_correlator_t *received_correlator;
    const arm_char_t       *description;
} arm_message_rcvd_event_t;
```

Description

The Message Received Event sub-buffer is optionally used to indicate that one or more messages have been received that have an effect on the execution state of a transaction. Some cases of interest are the following. For examples, see Section 4.2.

- A message causes the transaction to initiate.
- A message causes the transaction to unblock.
- A message is received that terminates an asynchronous transaction or a step in an asynchronous transaction.
- A message has been received during the processing of a transaction that does not fall into one of the categories above.

The Message Received Event sub-buffer can be provided on the following function calls:

- *arm_start_transaction*()
- *arm_update_transaction*()
- *arm_stop_transaction*()
- *arm_block_transaction*()
- *arm_unblock_transaction*()

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=9 (ARM_SUBBUFFER_MESSAGE_RCVD_EVENT).

- **End of Flow indicator:** An **arm_boolean_t** that if true indicates that the currently executing transaction represents the last processing step in a compound transaction that started in another process. The arm*_stop_transaction*() of the currently executing transaction can be considered the stop time for the entire compound transaction. After this flag has been set to True, the value for the transaction instance is True; the flag is ignored in any other Message Received Event sub-buffers associated with this transaction instance.

- **Count of message received event structures:** An **arm_int32_t** count of message received event structures. The maximum value is 32. The minimum value is 0, which might occur if the purpose of passing the sub-buffer is to set the End of Flow indicator, even though there are no new message events to report.

- **Pointer to an array of message received event structures:**

  — **Received Correlator:** An optional pointer to a buffer in which the correlator associated with a received message may optionally be stored. The correlator is useful for indicating the transaction and message originator to which this message event relates. A null pointer indicates that no correlator is provided.

  A further restriction is that there can be no more than 32 received correlators associated with a transaction, including the parent correlator parameter of *arm_start_transaction*(), even if multiple API calls are made. For example, if a correlator is passed as a parameter on *arm_start_transaction*() only 31 other correlators could be passed in any combination of calls using this sub-buffer. There are no constraints on the uniqueness of the received correlators or any requirements that either the application or the ARM implementation test the contents of the correlator to see if it is unique.

  There are now two ways to provide parent correlators: this field and the *parent_correlator* parameter of *arm_start_transaction*(). Two considerations affecting which to use are the following:

  — The contents of a parent correlator often influence the contents of a transaction's current correlator. For example, information about the originating user transaction (the root parent of the call graph) may be copied from the parent correlator to each child correlator at each node in the call graph, so they are available at all the nodes. Because a current correlator is created when *arm_start_transaction*() executes, only parent correlators available at that time influence the contents of the current correlator.

  — Applications that use ARM 4.1 can call ARM libraries that implement either ARM 4.0 and/or ARM 4.1. Because ARM 4.0 implementations do not recognize the Message Received Event sub-buffer, a correlator in the *parent_correlator* parameter of *arm_start_transaction*() will be recognized and may be used, whereas a correlator in the Message Received Event sub-buffer will not be recognized or used.

The recommended practice for correlators in *arm_start_transaction*() is the following. Correlators received after *arm_start_transaction*() executes can only be passed in the Message Received Event sub-buffer.

— If there is one parent correlator available when *arm_start_transaction*() executes, pass the correlator in the *parent_correlator* parameter of *arm_start_transaction*().

— If there are multiple parent correlators available when *arm_start_transaction*() executes, and one of them is considered the primary parent, pass that one in the *parent_correlator* parameter of *arm_start_transaction*() and the other(s) in the Message Received Event sub-buffer.

— If there are multiple parent correlators available when *arm_start_transaction*() executes and none are considered the most important, pass all the correlators in the Message Received Event sub-buffer.

— **Description:** This is an optional null-terminated character string with a maximum length of 128 characters, including the termination character, which describes the message event. A null value indicates no message event description is provided.

13.9 Message Sent Event

[Added in ARM 4.1]

(*format*=10, ARM_SUBBUFFER_MESSAGE_SENT_EVENT)

Syntax

```
typedef struct arm_subbuffer_message_sent_event
{
    arm_subbuffer_t                  header;
    arm_boolean_t                    end_of_flow;
    arm_int32_t                      event_count;
    const arm_message_sent_event_t *message_event_array;
} arm_subbuffer_message_sent_event_t;

typedef struct arm_message_sent_event
{
    arm_int32_t        sent_message_count;
    const arm_char_t  *description;
} arm_message_sent_event_t;
```

Description

The Message Sent Event sub-buffer is optionally used to indicate that one or more messages have been sent that have an effect on the execution state of a transaction. The cases of interest are the following. For examples, see Section 4.2.

- A message is sent that causes the transaction to block.
- A message is sent that initiates or terminates an asynchronous transaction or a step in an asynchronous transaction.
- A message is sent that is part of a conversational exchange between this transaction and another transaction.
- A message is sent that is not part of a known conversational exchange.

The Message Sent Event sub-buffer can be provided on the following function calls:

- *arm_start_transaction*()
- *arm_update_transaction*()
- *arm_stop_transaction*()
- *arm_block_transaction*()
- *arm_unblock_transaction*()

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=10 (ARM_SUBBUFFER_MESSAGE_SENT_EVENT).
- **End of Flow indicator**: An **arm_boolean_t** that if true indicates that the currently executing transaction represents the last processing step in a compound transaction that started in another process. The *arm_stop_transaction*() of the currently executing transaction can be considered the stop time for the entire compound transaction. It serves no purpose to set this flag more than once for a given transaction instance.
- **Count of message sent event structures**: An **arm_int32_t** count of message sent event structures. The maximum value is 32. The minimum value is 0, which might occur if the End of Flow indicator = true and there are no new message events to report.
- **Pointer to an array of message sent event structures:**

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

  — **Sent Count**: A count of the number of messages sent that are considered equivalent at the discretion of the instrumenter. A value of 0 is treated as the default value of 1. The count field provides an optional way for an application to indicate multiple messages of the same type with a single message event buffer.

  — **Description**: This is an optional null-terminated character string with a maximum length of 128 characters, including the termination character, which describes the message event. A null value indicates that no message event description is provided.

13.10 Formatted Arrival Time UsecJan1970

[Added in ARM 4.1]

(*format*=11, ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME_USECJAN1970)

Syntax

```
typedef struct arm_subbuffer_formatted_arrival_time_usecJan1970
{
    arm_subbuffer_t  header;
    arm_int64_t      usecJan1970;
} arm_subbuffer_formatted_arrival_time_usecJan1970_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. This scenario is described in Chapter 6.

When the application is unable to call ARM when the transaction starts, but it can capture and save a timestamp at that time, the application can format the arrival time, store it in the Formatted Arrival Time UsecJan1970 sub-buffer, and provide the sub-buffer with *arm_start_transaction*().

There are five sub-buffers (listed below) that are used for essentially the same purpose – to provide a more accurate start time than can be achieved with *arm_start_transaction*() alone. If more than one of these sub-buffers is passed, the ARM implementation will select which one to use, such as a heuristic that it will use the first sub-buffer in the list that it recognizes and ignore all others. Under no circumstances should the ARM implementation use data from more than one of the sub-buffersL

- Arrival Time sub-buffer (which contains an opaque formatted time)
- Formatted Arrival Time MsecJan1970 sub-buffer
- Formatted Arrival Time Strings sub-buffer
- Preparation Time sub-buffer
- Preparation Statistics sub-buffer

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=11 (ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME_MSECJAN1970).
- **Microseconds since Jan 1, 1970:** An **arm_int64_t** containing the number of microseconds since midnight January 1, 1970, UTC (coordinated universal time). The time source should be the same as the process in which the *arm_start_transaction*() executes.

13.11 Formatted Arrival Time Strings

[Added in ARM 4.1]

(*format*=12, ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME_STRINGS)

Syntax

```
typedef struct arm_subbuffer_formatted_arrival_time_strings
{
    arm_subbuffer_t header;
    const arm_char_t  *yyyymmdd;
    const arm_char_t  *hhmmssth;
    const arm_char_t  *muuu;   /* null pointer implies muuu = '0000' */
} arm_subbuffer_formatted_arrival_time_strings_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. This scenario is described in Chapter 6.

When the application is unable to call ARM when the transaction starts, but it can capture and save a timestamp at that time, the application can format the arrival time, store it in the Formatted Arrival Time Strings sub-buffer, and provide the sub-buffer with *arm_start_transaction*().

There are five sub-buffers (listed below) that are used for essentially the same purpose – to provide a more accurate start time than can be achieved with *arm_start_transaction*() alone. If more than one of these sub-buffers is passed, the ARM implementation will select which one to use, such as a heuristic that it will use the first sub-buffer in the list that it recognizes and ignore all others. Under no circumstances should the ARM implementation use data from more than one of the sub-buffers.

- Arrival Time sub-buffer (which contains an opaque formatted time)
- Formatted Arrival Time MsecJan1970 sub-buffer
- Formatted Arrival Time Strings sub-buffer
- Preparation Time sub-buffer
- Preparation Statistics sub-buffer

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=12 (ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME).

- **yyyymmdd:** An **arm_char_t*** to a null-terminated string containing the date (UTC) in the format Year, Month, Date. The time source should be the same one as the process in which the *arm_start_transaction*() executes.
- **hhmmssth:** An **arm_char_t*** to a null-terminated string containing the time (UTC) in the format Hours, Minutes, Seconds, Tenths of seconds, Hundredths of seconds. The time source should be the same as the process in which the *arm_start_transaction*() executes.
- **muuu:** An **arm_char_t*** to a null-terminated string containing the time (UTC) in the format milliseconds (one digit), microseconds (three digits). The time source should be the same as the process in which the *arm_start_transaction*() executes.

13.12 Preparation Time

[Added in ARM 4.1]

(*format*=13, ARM_SUBBUFFER_PREP_TIME)

Syntax

```
typedef struct arm_subbuffer_prep_time
{
    arm_subbuffer_t  header;
    arm_int64_t      prep_time_nanosec;
} arm_subbuffer_prep_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. This scenario is described in Chapter 6.

When the application is unable to call ARM when the transaction starts, but it can measure the delay for the transaction instance, the application can provide the measurements in the Preparation Time sub-buffer, and provide the sub-buffer with *arm_start_transaction*().

There are five sub-buffers (listed below) that are used for essentially the same purpose – to provide a more accurate start time than can be achieved with *arm_start_transaction*() alone. If more than one of these sub-buffers is passed, the ARM implementation will select which one to use, such as a heuristic that it will use the first sub-buffer in the list that it recognizes and ignore all others. Under no circumstances should the ARM implementation use data from more than one of the sub-buffers.

- Arrival Time sub-buffer (which contains an opaque formatted time)
- Formatted Arrival Time MsecJan1970 sub-buffer
- Formatted Arrival Time Strings sub-buffer
- Preparation Time sub-buffer
- Preparation Statistics sub-buffer

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=13 (ARM_SUBBUFFER_PREPARATION TIME).
- **Preparation Time in Nanoseconds:** An **arm_int64_t** containing the number of nanoseconds for this transaction instance that are considered the Prep Time.

13.13 Preparation Statistics

[Added in ARM 4.1]

(*format*=14, ARM_SUBBUFFER_PREP_STATS)

Syntax

```
typedef struct arm_subbuffer_prep_stats
{
    arm_subbuffer_t    header;
    arm_int64_t  prep_time_mean_nanosec;
    arm_int64_t  prep_time_std_dev_nanosec;
    arm_int32_t  prep_time_mean_count;
    arm_int32_t  prep_time_mean_interval_millisec;
} arm_subbuffer_prep_stats_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. This scenario is described in Chapter 6.

When the application is unable to call ARM when the transaction starts, or measure the delay for a specific transaction instance (for which it uses the Preparation Time sub-buffer), but it can measure the mean over several transactions, the application can provide the measurements in the Arrival Statistics sub-buffer, and provide the sub-buffer with *arm_start_transaction*().

There are five sub-buffers (listed below) that are used for essentially the same purpose – to provide a more accurate start time than can be achieved with *arm_start_transaction*() alone. If more than one of these sub-buffers is passed, the ARM implementation will select which one to use, such as a heuristic that it will use the first sub-buffer in the list that it recognizes and ignore all others. Under no circumstances should the ARM implementation use data from more than one of the sub-buffers.

- Arrival Time sub-buffer (which contains an opaque formatted time)
- Formatted Arrival Time MsecJan1970 sub-buffer
- Formatted Arrival Time Strings sub-buffer
- Preparation Time sub-buffer
- Preparation Statistics sub-buffer

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, value=14 (ARM_SUBBUFFER_PREP_STATS).

- **Preparation Time Mean in Nanoseconds:** An **arm_int64_t** containing the arithmetic mean of the Prep Time in nanoseconds.
- **Preparation Time Standard Deviation Nanoseconds:** An **arm_int64_t** containing the standard deviation of the Prep Time in nanoseconds. Its purpose is to provide more information for statistical analysis. This value is optional; a value of zero indicates that no value is provided.
- **Preparation Time Mean Count:** An **arm_int32_t** containing a count of the number of transactions that were used when calculating the preparation time mean. Its purpose is to provide more information for statistical analysis. This value is optional; a value of zero indicates that no value is provided.
- **Preparation Time Mean Interval Milliseconds:** An **arm_int32_t** containing the duration of the interval in milliseconds that was used when calculating the preparation time mean. Its purpose is to provide more information for statistical analysis. This value is optional; a value of zero indicates that no value is provided.

13.14 Diagnostic Properties

[Added in ARM 4.1]

(*format*=15, ARM_SUBBUFFER_DIAG_PROPERTIES)

Syntax

```
typedef struct arm_subbuffer_diag_properties
{
          arm_subbuffer_t  header;
          arm_int32_t      tran_property_count;
    const arm_property_t  *tran_property_array;
} arm_subbuffer_diag_properties_t;

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

Description

Applications may provide additional properties of the form *name*=*value* when a transaction ends [*arm_stop_transaction*() or *arm_report_transaction*()] by passing properties in the Diagnostic Properties sub-buffer. The sub-buffer is ignored on all other calls. There are no implicit requirements on how or if these values are processed.

Either the Diagnostic Properties or the Diagnostic Details sub-buffer may be provided, but not both.

There are constraints on the number of non-null properties and their size at any point in time.

Constraint Description	Constraint Value
Maximum number of non-null properties	20
Maximum number of characters of any property name, including the termination character	128
Maximum number of characters of any property name + property value, including the termination characters	2048
Maximum number of characters of all property names + property values, including the termination characters	4096

Table 15: Diagnostic Property Constraints

Format

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=14 (ARM_SUBBUFFER_DIAG_PROPERTIES).
- **Count of property values:** An **arm_int32_t** count of property values in the following array.
- **Pointer to an array of diagnostic 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.

  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**: An **arm_char_t*** to a null-terminated string representing the name part of the (*name*,*value*) pair. If the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Names should not contain trailing blank characters or consist of only blank characters.

  — **Value**: An **arm_char_t*** to a null-terminated string representing the value part of the (*name*,*value*) pair. If the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Values should not contain trailing blank characters or consist of only blank characters.

13.15 Application Identity

(*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 25:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=102 (ARM_SUBBUFFER_APP_IDENTITY).
- **Count of property values:** An **arm_int32_t** 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:E

  — **Name:** An **arm_char_t*** 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 the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Names should not contain trailing blank characters or consist of only blank characters.

  — **Value:** An **arm_char_t*** 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 the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Values should not contain trailing blank characters or consist of only blank characters.

- **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:** An **arm_char_t*** 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 points to a zero-length string, the (*name*,*value*) pair is ignored. The values are provided in the Application Context Values sub-buffer. Names should not contain trailing blank characters or consist of only blank characters.

13.16 Application Context Values

(*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 25:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=103 (ARM_SUBBUFFER_APP_CONTEXT).
- **Count of property values:** An **arm_int32_t** count of property values in the following array.
- **Pointer to an array of context 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. If the corresponding property name pointer is null or points to a zero-length string, the value is ignored. Each pointer is aligned as required for the C compiler on the platform. Each array element contains:

  — **Value:** An **arm_char_t*** 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 points to a zero-length string, the meaning is that there is no value for this instance. The value should not contain trailing blank characters or consist of only blank characters.

13.17 Transaction Identity

(*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 illustrated in Figure 25:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=104 (ARM_SUBBUFFER_TRAN_IDENTITY).
- **Count of elements in the identity property names and values array:** An **arm_int32_t** 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. Each structure contains:

  — **Name:** An **arm_char_t*** 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 the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Names should not contain trailing blank characters or consist of only blank characters.

  — **Value:** An **arm_char_t*** 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 the pointer is null or points to a zero-length string, the (*name*,*value*) pair is ignored. Values should not contain trailing blank characters or consist of only blank characters.

- **Count of elements in the context property names array:** An **arm_int32_t** 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 points to a zero-length string, the name is ignored. Each array element is an **arm_char_t*** 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 Transaction 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:** An **arm_char_t*** 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 points to a zero-length string, the (*name*,*value*) pair is ignored. The values are provided in the Transaction Context Values sub-buffer. Names should not contain trailing blank characters or consist of only blank characters.

- **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. A zero-length string is treated as a null value.

13.18 Transaction Context

(*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 25:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=105 (ARM_SUBBUFFER_TRAN_CONTEXT).
- **Count of elements in the context property values array:** An **arm_int32_t** count of elements in the following array.
- **Pointer to an array of context 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 Transaction Identity sub-buffer. If the corresponding property name pointer is null or points to a zero-length string, the value is ignored. Each pointer is aligned as required for the C compiler on the platform. Each array element contains:

  — **Value:** An **arm_char_t*** 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 points to a zero-length string, the meaning is that there is no value for this instance. Values should not contain trailing blank characters or consist of only blank characters.

- **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.

13.19 Metric Bindings

(*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 25.

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=106 (ARM_SUBBUFFER_METRIC_BINDINGS).
- **Count of metric bindings:** An **arm_int32_t** 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*().

13.20 Character Set Encoding

(*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 7). 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 encoding must comply with the following constraints:

- The character set must 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.
- No more than three bytes may be used to encode each 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 **arm_char_t*** 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 24:

- **Sub-buffer format ID:** An **arm_int32_t** sub-buffer format ID, *value*=107 (ARM_SUBBUFFER_CHARSET).
- **Character set ID (MIBenum):** An **arm_int32_t** containing a MIBenum value assigned by the IANA. Some common encodings are listed in Table 7.
- **Flags:** An **arm_int32_t** containing bit flags. The field is currently reserved for future use.

13.21 Application Control

[Added in ARM 4.1]

(*format*=108, ARM_SUBBUFFER_APP_CONTROL)

Syntax

```
typedef struct arm_subbuffer_app_control
{
   /*[in]*/       arm_subbuffer_t  header;
   /*[in]*/       arm_int32_t      control_count_app;
   /*[out]*/      arm_int32_t      control_count_arm;
   /*[in/out]*/   arm_boolean_t    app_control_used;
   /*[out]*/      arm_boolean_t    tran_id_control_used;
   /*[out]*/      arm_boolean_t    tran_instance_control_used;
   /*[out]*/      arm_int32_t      collection_depth;
   /*[out]*/      arm_boolean_t    show_private;
   /*[out]*/      arm_boolean_t    show_secure;
   /*[out]*/      arm_boolean_t    use_bind_thread;
   /*[out]*/      arm_boolean_t    use_block;
   /*[out]*/      arm_boolean_t    use_diagnostic;
   /*[out]*/      arm_boolean_t    use_message_event;
   /*[out]*/      arm_boolean_t    use_metric;
   /*[out]*/      arm_boolean_t    use_user;
} arm_subbuffer_app_control_t;
```

Description

The Application Control sub-buffer is used by applications to request the type and scope of instrumentation the ARM implementation prefers. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to all transactions registered by this application. These settings, except for *app_control_used*, *tran_id_control_used*, *tran_instance_control_used*, *show_private*, and *show_secure* may be overridden for specific transaction IDs and/or instances.

The Application Control sub-buffer is passed on *arm_start_applicaiton*() or with a special form of *arm_generate_correlator*(), as described in Chapter 10. This special form sets *current_correlator* = Null. Setting *current_correlator* = Null renders the *arm_generate_correlator*() call meaningless for the purpose of generating a correlator.

Note: Some ARM 4.0 implementations do not expect to find a null current correlator value in *arm_generate_correlator*() and set the return code to –1012. See the **Note** in the *arm_generate_correlator*() description for an explanation of how to proceed.

The *app_control_used* setting is used as a handshake to determine whether both the application and the ARM implementation are using instrumentation control. Both must agree if instrumentation control is to be used. This provides protection for applications and implementations that do not support the capability.

Note: A return value of *app_control_used*=False does not indicate that the ARM library is not collecting data. The application should continue to make ARM calls using its default assumptions about the appropriate amount of transaction detail.

Format

The pattern is illustrated in Figure 24:

- **header** (**sub-buffer format ID**): An **arm_int32_t** sub-buffer format ID, *value*=108 (ARM_SUBBUFFER_APP_CONTROL).
- **control_count_app**: An **arm_int32_t** constant that indicates the sub-buffer version used by the application. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with *app_control_used*. For ARM 4.1 this value is 12 (ARM41_APP_CONTROL_COUNT).
- **control_count_arm**: An **arm_int32_t** constant that indicates the sub-buffer version used by the ARM implementation. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with *app_control_used*. For ARM 4.1 this value is 12 (ARM41_APP_CONTROL_COUNT).
- **app_control_used**: An **arm_boolean_t** that indicates whether instrumentation control is being used. This in/out value provides a handshake between the application and the ARM implementation to determine whether controls are used. This value must be set to False by the application. The ARM implementation leaves the value False if it is not using controls, such as if it does not recognize the sub-buffer or chooses not to use it. The meaning upon return from the API call is as follows:

  — True = The other flags in the Application Control sub-buffer are set to meaningful values.

  — False = All other values in the Application Control sub-buffer must be ignored and the Transaction ID Control and Transaction Instance Control sub-buffers must be ignored if either is passed.
- **tran_id_control_used:** An **arm_boolean_t** that indicates whether the ARM implementation uses the Transaction ID Control sub-buffer. The Transaction ID Control sub-buffer is used to set the default collection parameters for all transaction instances for that registered ID.

  — True =Preferences may be set at the (registered) transaction ID level, and therefore it would be useful to use the Transaction ID Control sub-buffer.

  — False =The Transaction ID Control sub-buffer must be ignored if it is passed.
- **tran_instance_control_used:** An **arm_boolean_t** that indicates whether the ARM implementation uses the Transaction Instance Control sub-buffer. The Transaction

Instance Control sub-buffer is used to set the collection parameters for a single transaction instance.

— True =Preferences may be set at the transaction instance level, and therefore it would be useful to use the Transaction Instance Control sub-buffer.

— False =The Transaction Instance Control sub-buffer must be ignored if it is passed.

- **collection_depth:** An **arm_int32_t** that indicates the granularity of transactions that the ARM implementation prefers. This setting is orthogonal to other control settings, such as *use_bind_thread* and *use_block*. It influences only whether and how often start/update/stop API calls are used.

  If set to a value >0, then the following ARM capabilities should be used if the application supports them:

  — *arm_start_transaction*()

  — *arm_update_transaction*()

  — *arm_stop_transaction*()

  — Parent correlator

  — Current correlator

  — Application Identity sub-buffer

  — Application Context sub-buffer

  — Transaction Identity sub-buffer

  — Transaction Context sub-buffer

  There are four possible values:

  — 0 (ARM_COLLECTION_DEPTH_NONE)
  The implementation prefers that the application does not instrument any transactions.

  — 1 (ARM_COLLECTION_DEPTH_PROCESS)
  The implementation prefers that the application instrument these transactions at a process granularity; i.e., that it uses a single *arm_start_transaction*()/*arm_stop_transaction*() pair per process.

  — 2 (ARM_COLLECTION_DEPTH_CONTAINER)
  The implementation prefers that the application instruments these transactions at a container granularity; i.e., that it use a single start/stop pair per container.

  The determination of what constitutes a container is at the discretion of the application architect who designs the ARM instrumentation. For many applications the only logical container boundary is also the process boundary, in which case there would be no difference in how the settings of 1 and 2 would be interpreted. For other applications, such as complex middleware that may contain a web server, J2EE container, JDBC connectors, there could be multiple containers running within the same process.

— 3 (*ARM_COLLECTION_DEPTH_MAX*)
The implementation prefers that the application instruments transactions at the maximum possible granularity. This may be the same as the process or container granularity, depending on the application.

- **show_private:** An **arm_boolean_t** that indicates whether private data (e.g., account numbers) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The determination of what constitutes private data is made by the application.

  — True = Provide private data

  — False = Do not provide private data

- **show_secure:** An **arm_boolean_t** that indicates whether secure data (e.g., passwords) should be provided in any form, such as in a metric or diagnostic data, or in a correlator. The determination of what constitutes secure data is made by the application.

  — True = Provide secure data

  — False = Do not provide secure data

- **use_bind_thread:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_bind_thread*() and *arm_unbind_thread*() calls.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if thread bindings are reported, then *all* thread bindings should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

  — True = Call *arm_bind_thread*()/*arm_unbind_thread*() whenever a transaction is newly bound/unbound to/from a thread.

  — False = The implementation will ignore any *arm_bind_thread*() and *arm_unbind_thread*() calls.

- **use_block:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_block_transaction*() and *arm_unblock_transaction*() calls, optionally passing the Block Cause sub-buffer on each call.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if blocking conditions are reported then *all* blocking conditions should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

  — True = Call *arm_block_transaction*()/*arm_unblock_transaction*() whenever a control flow is blocked waiting on an external event.

  — False = The implementation will ignore any *arm_block_transaction*() and *arm_unblock_transaction*() calls.

- **use_diagnostic:** An **arm_boolean_t** that indicates that the implementation prefers that the application use the Diagnostic Detail and/or Diagnostic Properties sub-buffers.

  — True = Use the Diagnostic Detail and/or Diagnostic Properties sub-buffers whenever appropriate.

  — False = The Diagnostic Detail and Diagnostic Properties sub-buffers will be ignored if they are passed.

- **use_message_event:** An **arm_boolean_t** that indicates that the implementation prefers that the application inform it about message exchanges and other asynchronous flows.

  — True = Use the Message Received and Message Sent sub-buffers, and use the SET_CORRELATOR_FLAG macro to set the Asynchronous Flow and Independent Transaction flags.

  — False = The Message Received and Message Sent sub-buffers will be ignored if they are passed.

- **use_metric:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide metric data.

  — True = Use the Metric Values sub-buffer whenever appropriate.

  — False = The Metric Values sub-buffer will be ignored if it is passed.

- **use_user:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide the identity of the user, if known.

  — True = Use the User sub-buffer whenever appropriate.

  — False = The User sub-buffer will be ignored if it is passed.

13.22 Transaction ID Control

[Added in ARM 4.1]

(*format*=109, ARM_SUBBUFFER_TRAN_ID_CONTROL)

Syntax

```
typedef struct arm_subbuffer_tran_id_control
{
   /*[in]*/           arm_subbuffer_t   header;
   /*[in]*/           arm_int32_t       control_count_app;
   /*[out]*/          arm_int32_t       control_count_arm;
   /*[in/out]*/       arm_boolean_t     control_used;
   /*[in]*/     const arm_id_t         *tran_id,
   /*[out]*/          arm_int32_t       collection_depth;
   /*[out]*/          arm_boolean_t     use_bind_thread;
   /*[out]*/          arm_boolean_t     use_block;
   /*[out]*/          arm_boolean_t     use_diagnostic;
   /*[out]*/          arm_boolean_t     use_message_event;
   /*[out]*/          arm_boolean_t     use_metric;
   /*[out]*/          arm_boolean_t     use_user;
} arm_subbuffer_tran_id_control_t;
```

Description

The Transaction ID Control sub-buffer is used by applications to request the type and scope of instrumentation the ARM implementation prefers for all instances using a registered transaction ID. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to all instances of the specified transaction ID. These settings may be overridden by the Transaction Instance Control sub-buffer.

The Transaction ID Control sub-buffer is passed using a special form of *arm_generate_correlator*(), as described in Chapter 10. This special form sets *current_correlator* = Null. Setting *current_correlator* = Null renders the *arm_generate_correlator*() call meaningless for the purpose of generating a correlator.

Note: Some ARM 4.0 implementations do not expect to find a null current correlator value in *arm_generate_correlator*() and set the return code to –1012. See the **Note** in the *arm_generate_correlator*() description for an explanation of how to proceed.

The *control_used* setting is used as a handshake to determine whether both the application and the ARM implementation are using instrumentation control. Both must agree if instrumentation control is to be used. This provides protection for applications and implementations that do not support the capability.

Note: A return value of *control_used*=False does not indicate that the ARM library is not collecting data. The application should continue to make ARM calls using its default assumptions about the appropriate amount of transaction detail, or the settings established using the application control buffer.

Format

The pattern is illustrated in Figure 25:

- **header** (**sub-buffer format ID**): An **arm_int32_t** sub-buffer format ID, *value*=109 (ARM_SUBBUFFER_TRAN_ID_CONTROL).
- **control_count_app**: An **arm_int32_t** constant that indicates the sub-buffer version used by the application. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with control_used. For ARM 4.1 this value is 9 (ARM41_TRAN_ID_CONTROL_COUNT).
- **control_count_arm**: An **arm_int32_t** constant that indicates the sub-buffer version used by the ARM implementation. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with control_used. For ARM 4.1 this value is 9 (ARM41_TRAN_ID_CONTROL_COUNT).
- **control_used**: An **arm_boolean_t** that indicates whether instrumentation control per transaction instance is being used. This in/out value provides a handshake between the application and the ARM implementation to determine whether controls are used. This value must be set to False by the application. The ARM implementation leaves the value False if it is not using instance-level controls, such as if it does not recognize the sub-buffer or chooses not to use it. The meaning upon return from the API call is as follows

  — True = The other flags in the Transaction ID Control sub-buffer are set to meaningful values.

  — False = All other values in the Transaction ID Control sub-buffer must be ignored.
- **tran_id:** An **arm_id_t** constant that points to a transaction ID values returned in an *out* parameter from an *arm_register_transaction*() call in the same process.
- **collection_depth:** An **arm_int32_t** that indicates the granularity of transactions that the ARM implementation prefers. This setting is orthogonal to other control settings, such as *use_bind_thread* and *use_block*. It influences only whether and how often start/update/stop API calls are used.

  If set to a value >0, then the following ARM capabilities should be used if the application supports them:

  — *arm_start_transaction*()

  — *arm_update_transaction*()

  — *arm_stop_transaction*()

— Parent correlator

— Current correlator

— Application Identity sub-buffer

— Application Context sub-buffer

— Transaction Identity sub-buffer

— Transaction Context sub-buffer

There are four possible values:

— 0 (ARM_COLLECTION_DEPTH_NONE)
The implementation prefers that the application does not instrument any transactions.

— 1 (ARM_COLLECTION_DEPTH_PROCESS)
The implementation prefers that the application instrument these transactions at a process granularity; i.e., that it use a single *arm_start_transaction*()/*arm_stop_transaction*() pair per process.

— 2 (ARM_COLLECTION_DEPTH_CONTAINER)
The implementation prefers that the application instruments these transactions at a container granularity; i.e., that it use a single start/stop pair per container.

The determination of what constitutes a container is at the discretion of the application architect who designs the ARM instrumentation. For many applications the only logical container boundary is also the process boundary, in which case there would be no difference in how the settings of 1 and 2 would be interpreted. For other applications, such as complex middleware that may contain a web server, J2EE container, JDBC connectors, there could be multiple containers running within the same process.

— 3 (ARM_COLLECTION_DEPTH_MAX)
The implementation prefers that the application instruments transactions at the maximum possible granularity. This may be the same as the process or container granularity, depending on the application.

- **use_bind_thread:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_bind_thread*() and *arm_unbind_thread*() calls.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if thread bindings are reported, then *all* thread bindings should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

  — True = Call *arm_bind_thread*()/*arm_unbind_thread*() whenever a transaction is newly bound/unbound to/from a thread.

  — False = The implementation will ignore any *arm_bind_thread*() and *arm_unbind_thread*() calls.

- **use_block:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_block_transaction*() and *arm_unblock_transaction*() calls and the Block Cause sub-buffer, if appropriate.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if blocking conditions are reported, then *all* blocking conditions should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

  — True = Call *arm_block_transaction*()/*arm_unblock_transaction*() whenever a control flow is blocked waiting on an external event.

  — False = The implementation will ignore any *arm_bind_thread*() and *arm_unbind_thread*() calls.

- **use_diagnostic:** An **arm_boolean_t** that indicates that the implementation prefers that the application use the Diagnostic Detail and/or Diagnostic Properties sub-buffers.

  — True = Use the Diagnostic Detail and/or Diagnostic Properties sub-buffers whenever appropriate.

  — False = The Diagnostic Detail and Diagnostic Properties sub-buffers will be ignored if they are passed.

- **use_message_event:** An **arm_boolean_t** that indicates that the implementation prefers that the application inform it about message exchanges and other asynchronous flows.

  — True = Use the Message Received and Message Sent sub-buffers, and use the SET_CORRELATOR_FLAG macro to set the Asynchronous Flow and Independent Transaction flags.

  — False = The Message Received and Message Sent sub-buffers will be ignored if they are passed.

- **use_metric:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide metric data.

  — True = Use the Metric Values sub-buffer whenever appropriate.

  — False = The Metric Values sub-buffer will be ignored if it is passed.

- **use_user:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide the identity of the user, if known.

  — True = Use the User sub-buffer whenever appropriate.

  — False = The User sub-buffer will be ignored if it is passed.

13.23 Transaction Instance Control

[Added in ARM 4.1]

(*format*=110, ARM_SUBBUFFER_TRAN_INSTANCE_CONTROL)

Syntax

```
typedef struct arm_subbuffer_tran_instance_control
{
   /*[in]*/           arm_subbuffer_t   header;
   /*[in]*/           arm_int32_t       control_count_app;
   /*[out]*/          arm_int32_t       control_count_arm;
   /*[in/out]*/       arm_boolean_t     control_used;
   /*[in]*/     const arm_id_t         *tran_id,
   /*[opt in]*/ const arm_correlator_t *parent_correlator,
   /*[out]*/          arm_int32_t       collection_depth;
   /*[out]*/          arm_boolean_t     use_bind_thread;
   /*[out]*/          arm_boolean_t     use_block;
   /*[out]*/          arm_boolean_t     use_diagnostic;
   /*[out]*/          arm_boolean_t     use_message_event;
   /*[out]*/          arm_boolean_t     use_metric;
   /*[out]*/          arm_boolean_t     use_user;
} arm_subbuffer_tran_instance_control_t;
```

Description

The Transaction Instance Control sub-buffer is used by applications to request the type and scope of instrumentation the ARM implementation prefers for a transaction instance. Its use is optional for both applications and ARM implementations. Further, the control settings represent preferences; they are not binding on the application.

The scope of the settings applies to a single transaction instance. They do not carry over to any other transaction instance. These settings override any settings set using the Application Control or Transaction ID Control sub-buffers.

The Transaction Instance Control sub-buffer is passed using a special form of *arm_generate_correlator*(), as described in Chapter 10. This special form sets *current_correlator* = Null. Setting *current_correlator* = Null renders the *arm_generate_correlator*() call meaningless for the purpose of generating a correlator.

Note: Some ARM 4.0 implementations do not expect to find a null current correlator value in *arm_generate_correlator*() and set the return code to –1012. See the **Note** in the *arm_generate_correlator*() description for an explanation of how to proceed.

The *control_used* setting is used as a handshake to determine whether both the application and the ARM implementation are using instrumentation control. Both must agree if instrumentation control is to be used. This provides protection for applications and implementations that do not support the capability.

Note: A return value of *control_used*=False does not indicate that the ARM library is not collecting data. The application should continue to make ARM calls using its default assumptions about the appropriate amount of transaction detail, or the settings estabslished using the application control buffer.

Format

The pattern is illustrated in Figure 25:

- **header (sub-buffer format ID):** An **arm_int32_t** sub-buffer format ID, *value*=110 (ARM_SUBBUFFER_TRAN_INSTANCE_CONTROL).
- **control_count_app:** An **arm_int32_t** constant that indicates the sub-buffer version used by the application. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with *control_used*. For ARM 4.1 this value is 10 (ARM41_TRAN_INSTANCE_CONTROL_COUNT).
- **control_count_arm:** An **arm_int32_t** constant that indicates the sub-buffer version used by the ARM implementation. This provides a way to add new controls in later versions of ARM without breaking backwards compatibility. The version number equals the number of controls starting with *control_used*. For ARM 4.1 this value is 10 (ARM41_TRAN_INSTANCE_CONTROL_COUNT).
- **control_used:** An **arm_boolean_t** that indicates whether instrumentation control per transaction instance is being used. This in/out value provides a handshake between the application and the ARM implementation to determine whether controls are used. This value must be set to False by the application. The ARM implementation leaves the value False if it is not using instance level controls, such as if it does not recognize the sub-buffer or chooses not to use it. The meaning upon return from the API call is as follows:

  — True = The other flags in the Transaction Instance Control sub-buffer are set to meaningful values.

  — False = All other values in the Transaction Instance Control sub-buffer must be ignored.
- **tran_id:** An **arm_id_t** constant that points to a transaction ID values returned in an *out* parameter from an *arm_register_transaction*() call in the same process.
- **parent_correlator:** An **arm_correlator_t** constant that points to the parent correlator, if any, for the transaction instance. The pointer may be null (ARM_CORR_NONE).
- **collection_depth:** An **arm_int32_t** that indicates the granularity of transactions that the ARM implementation prefers. This setting is orthogonal to other control settings, such as *use_bind_thread* and *use_block*. It influences only whether and how often start/update/stop API calls are used.

If set to a value >0, then the following ARM capabilities should be used if the application supports them:

— Transaction measurements; e.g., using *arm_start_transaction*(), *arm_stop_transaction*(), and *arm_report_transaction*(). It is assumed that the application will provide arrival time information if that is needed to achieve accurate response time measurements.

— Parent and child correlators

— Application and transaction identity and context

There are four possible values:

— 0 (ARM_COLLECTION_DEPTH_NONE)
The implementation prefers that the application does not instrument any transactions.

— 1 (ARM_COLLECTION_DEPTH_PROCESS)
The implementation prefers that the application instrument these transactions at a process granularity; i.e., that it use a single *arm_start_transaction*()/*arm_stop_transaction*() pair per process.

— 2 (ARM_COLLECTION_DEPTH_CONTAINER)
The implementation prefers that the application instruments these transactions at a container granularity; i.e., that it use a single start/stop pair per container.

The determination of what constitutes a container is at the discretion of the application architect who designs the ARM instrumentation. For many applications the only logical container boundary is also the process boundary, in which case there would be no difference in how the settings of 1 and 2 would be interpreted. For other applications, such as complex middleware that may contain a web server, J2EE container, JDBC connectors, there could be multiple containers running within the same process.

— 3 (ARM_COLLECTION_DEPTH_MAX)
The implementation prefers that the application instruments transactions at the maximum possible granularity. This may be the same as the process or container granularity, depending on the application.

- **use_bind_thread:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_bind_thread*() and *arm_unbind_thread*() calls.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if thread bindings are reported, then *all* thread bindings should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

— True = Call *arm_bind_thread*()/*arm_unbind_thread*() whenever a transaction is newly bound/unbound to/from a thread.

— False = The implementation will ignore any *arm_bind_thread*() and *arm_unbind_thread*() calls.

- **use_block:** An **arm_boolean_t** that indicates that the implementation prefers that the application makes *arm_block_transaction*() and *arm_unblock_transaction*() calls and the Block Cause sub-buffer, if appropriate.

  Note that this control is orthogonal to the *collection_depth* control for all values except ARM_COLLECTION_DEPTH_NONE: if blocking conditions are reported, then *all* blocking conditions should be reported regardless of the number of *arm_start_transaction*() and *arm_stop_transaction*() calls.

  — True = Call *arm_block_transaction*()/*arm_unblock_transaction*() whenever a control flow is blocked waiting on an external event.

  — False = The implementation will ignore any *arm_bind_thread*() and *arm_unbind_thread*() calls.

- **use_diagnostic:** An **arm_boolean_t** that indicates that the implementation prefers that the application use the Diagnostic Detail and/or Diagnostic Properties sub-buffers.

  — True = Use the Diagnostic Detail and/or Diagnostic Properties sub-buffers whenever appropriate.

  — False = The Diagnostic Detail and Diagnostic Properties sub-buffers will be ignored if they are passed.

- **use_message_event:** An **arm_boolean_t** that indicates that the implementation prefers that the application informs it about message exchanges and other asynchronous flows.

  — True = Use the Message Received and Message Sent sub-buffers, and use the SET_CORRELATOR_FLAG macro to set the Asynchronous Flow and Independent Transaction flags.

  — False = The Message Received and Message Sent sub-buffers will be ignored if they are passed.

- **use_metric:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide metric data.

  — True = Use the Metric Values sub-buffer whenever appropriate.

  — False = The Metric Values sub-buffer will be ignored if it is passed.

- **use_user:** An **arm_boolean_t** that indicates that the implementation prefers that the application provide the identity of the user, if known.

  — True = Use the User sub-buffer whenever appropriate.

  — False = The User sub-buffer will be ignored if it is passed.

14 <arm4.h> Header File for Compiling

```
/* ------------------------------------------------------------------------ */
/*                                                                          */
/* Copyright (c) 2003 The Open Group                                        */
/*                                                                          */
/* Permission is hereby granted, free of charge, to any person obtaining a   */
/* copy of this software (the "Software"), to deal in the Software without   */
/* restriction, including without limitation the rights to use, copy,        */
/* modify, merge, publish, distribute, sublicense, and/or sell copies of     */
/* the Software, and to permit persons to whom the Software is furnished     */
/* to do so, subject to the following conditions:                           */
/*                                                                          */
/* The above copyright notice and this permission notice shall be included   */
/* in all copies or substantial portions of the Software.                   */
/*                                                                          */
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS   */
/* OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF                */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.    */
/* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY      */
/* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT */
/* OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR  */
/* THE USE OR OTHER DEALINGS IN THE SOFTWARE.                               */
/*                                                                          */
/* ------------------------------------------------------------------------ */
/*                                                                          */
/* File revision information                                                */
/*                                                                          */
/* $Source: /hand_cvs/arm4/sdk4/c/include/arm4.h,v $    */
/* $Revision: 1.2.2.1 $  */
/* $Date: 2006/02/19 18:33:50 $      */
/*                                                                          */
/* ------------------------------------------------------------- */
/* 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 */
```

15 <arm41.h> Header File for Compiling

```
/*     **** NOTE: preliminary version, not for public release !! ****        */
/* ------------------------------------------------------------------------- */
/*                                                                           */
/* Copyright (c) 2006 The Open Group                                         */
/*                                                                           */
/* Permission is hereby granted, free of charge, to any person obtaining a   */
/* copy of this software (the "Software"), to deal in the Software without   */
/* restriction, including without limitation the rights to use, copy,        */
/* modify, merge, publish, distribute, sublicense, and/or sell copies of     */
/* the Software, and to permit persons to whom the Software is furnished     */
/* to do so, subject to the following conditions:                            */
/*                                                                           */
/* The above copyright notice and this permission notice shall be included   */
/* in all copies or substantial portions of the Software.                    */
/*                                                                           */
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS   */
/* OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF                */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.    */
/* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY      */
/* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT */
/* OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR  */
/* THE USE OR OTHER DEALINGS IN THE SOFTWARE.                                */
/*                                                                           */
/* ------------------------------------------------------------------------- */
/*                                                                           */
/* File revision information                                                 */
/*                                                                           */
/* $Source: /tang_cvs/arm4/sdk4/c/include/Attic/arm41.h,v $    */
/* $Revision: 1.1.2.5 $  */
/* $Date: 2006/06/12 11:29:47 $      */
/*                                                                           */
/* ------------------------------------------------------------- */
/* arm41.h - ARM4.1 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 ARM4.1 C language binding extends the ARM 4.0 C     */
/* binding.                                                      */
/* ------------------------------------------------------------- */

#if !defined(ARM41_H_INCLUDED)
#define ARM41_H_INCLUDED

#ifndef ARM4_H_INCLUDED
#include "arm4.h"
#endif /* ARM4_H_INCLUDED */

/* ------------------------ correlator flags ----------------------------- */
/* Correlator interface flag numbers. See                          */
/* arm_get_correlator_flags().                                     */
/* MMT: anticipated spec change: rename _FLAG_ to _FLAGNUM_ */
#define ARM_CORR_FLAGNUM_ASYNCH                 3
#define ARM_CORR_FLAGNUM_INDEPENDENT            4

/* Macro for modifying correlator flags. The flag parameter */
/* must be one of the *_FLAGNUM_* constants.                */
#define ARM_SET_CORRELATOR_FLAG(corr, flag_num, boolean_value) \
do \
{ \
  if (boolean_value) \
    (corr)->opaque[3] |= (arm_uint8_t)0x80 >> (arm_uint8_t)((flag_num)-1) ; \
  else \
    (corr)->opaque[3] &= ~((arm_uint8_t)0x80 >> (arm_uint8_t)((flag_num)-1)) ;
\
} while(0)

/* --------------------- message event constants ------------------------ */

#define ARM_MESSAGE_SENT_EVENT_MAX_COUNT       32
#define ARM_MESSAGE_RCVD_EVENT_MAX_COUNT       32

/* ----------------- diagnostic properties constants -------------------- */

/* max. length of name+value of a diagnostic property */
#define ARM_DIAG_PROPERTY_MAX_CHARS          2046
#define ARM_DIAG_PROPERTY_MAX_LENGTH \
   (ARM_DIAG_PROPERTY_MAX_CHARS*3+2)

/* max. total length of diagnostic properties */
/* MMT Cannot provide an accurate formula here. This rule was meant */
/* to keep the property block below 4KB length? tbd. */
#define ARM_DIAG_PROPERTY_MAX_BYTES          4096

/* ---------------------- miscellaneous constants ------------------------ */

/* max. length of the description fields of block cause and message events */
#define ARM_EVENT_DESCRIPTION_MAX_CHARS       127
#define ARM_EVENT_DESCRIPTION_MAX_LENGTH \
   (ARM_EVENT_DESCRIPTION_MAX_CHARS*3+1)

/* ------------------------ block cause constants ------------------------ */

#define ARM_BLOCK_CAUSE_SYNCHRONOUS_EVENT       1
#define ARM_BLOCK_CAUSE_ASYNCHRONOUS_EVENT      2

/* -------------- application/transaction control constants -------------- */

#define ARM41_APP_CONTROL_COUNT                12
#define ARM41_TRAN_ID_CONTROL_COUNT             9
#define ARM41_TRAN_INSTANCE_CONTROL_COUNT      10

#define ARM_COLLECTION_DEPTH_NONE               0
#define ARM_COLLECTION_DEPTH_PROCESS            1
#define ARM_COLLECTION_DEPTH_CONTAINER          2
#define ARM_COLLECTION_DEPTH_MAX                3

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

```
#define ARM_SUBBUFFER_BLOCK_CAUSE                8
#define ARM_SUBBUFFER_MESSAGE_RCVD_EVENT         9
#define ARM_SUBBUFFER_MESSAGE_SENT_EVENT        10
#define ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME_USECJAN1970 11
#define ARM_SUBBUFFER_FORMATTED_ARRIVAL_TIME_STRINGS     12
#define ARM_SUBBUFFER_PREP_TIME                 13
#define ARM_SUBBUFFER_PREP_STATS                14
#define ARM_SUBBUFFER_DIAG_PROPERTIES           15

#define ARM_SUBBUFFER_APP_CONTROL              108
#define ARM_SUBBUFFER_TRAN_ID_CONTROL          109
#define ARM_SUBBUFFER_TRAN_INSTANCE_CONTROL    110

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

typedef arm_int32_t arm_block_cause_t;

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

/* -------------------- block cause event sub-buffer -------------------- */

typedef struct arm_subbuffer_block_cause
{
    arm_subbuffer_t              header;
    arm_block_cause_t            cause;
    arm_int32_t                  extended_cause;
    const arm_char_t             *description;
} arm_subbuffer_block_cause_t;

/* ------------------ message received event sub-buffer ------------------- */

typedef struct arm_message_rcvd_event
{
   const arm_correlator_t *received_correlator;
   const arm_char_t       *description;
} arm_message_rcvd_event_t;

typedef struct arm_subbuffer_message_rcvd_event
{
   arm_subbuffer_t                 header;
   arm_boolean_t                   end_of_flow;
   arm_int32_t                     event_count;
   const arm_message_rcvd_event_t  *message_event_array;
} arm_subbuffer_message_rcvd_event_t;

/* ------------------- message sent event sub-buffer --------------------- */

typedef struct arm_message_sent_event
{
   arm_int32_t               sent_message_count;
   const arm_char_t                *description;
} arm_message_sent_event_t;

typedef struct arm_subbuffer_message_sent_event
{
   arm_subbuffer_t                 header;
   arm_boolean_t                   end_of_flow;
   arm_int32_t                     event_count;
   const arm_message_sent_event_t  *message_event_array;
} arm_subbuffer_message_sent_event_t;

/* ------------ formatted arrival time usecJan1970 sub-buffer ------------ */

typedef struct arm_subbuffer_formatted_arrival_time_usecJan1970
{
   arm_subbuffer_t                 header;
   arm_int64_t                     usecJan1970;
} arm_subbuffer_formatted_arrival_time_usecJan1970_t;

/* ------------- formatted arrival time strings sub-buffer --------------- */

typedef struct arm_subbuffer_formatted_arrival_time_strings
{
   arm_subbuffer_t header;
   const arm_char_t                *yyyymmdd;
   const arm_char_t                *hhmmssth;
   const arm_char_t                *muuu;
                                   /* null pointer implies muuu = '0000' */
} arm_subbuffer_formatted_arrival_time_strings_t;

/* ------------------- preparation time sub-buffer ---------------------- */

typedef struct arm_subbuffer_prep_time
{
   arm_subbuffer_t                 header;
   arm_int64_t                     prep_time_nanosec;
} arm_subbuffer_prep_time_t;

/* ----------------- preparation statistics sub-buffer ------------------- */

typedef struct arm_subbuffer_prep_stats
{
   arm_subbuffer_t                 header;
   arm_int64_t                     prep_time_mean_nanosec;
   arm_int64_t                     prep_time_std_dev_nanosec;
   arm_int32_t                     prep_time_mean_count;
   arm_int32_t                     prep_time_mean_interval_millisec;
} arm_subbuffer_prep_stats_t;

/* ----------------- diagnostic properties sub-buffer -------------------- */

typedef struct arm_subbuffer_diag_properties
{
   arm_subbuffer_t                 header;
   arm_int32_t                     tran_property_count;
   const arm_property_t            *tran_property_array;
} arm_subbuffer_diag_properties_t;

/* ------------------ application control sub-buffer -------------------- */

typedef struct arm_subbuffer_app_control
{
   const arm_subbuffer_t           header;
   arm_int32_t               control_count_app;
   arm_int32_t                     control_count_arm;
   arm_boolean_t                   app_control_used;
   arm_boolean_t                    tran_id_control_used;
   arm_boolean_t                    tran_instance_control_used;
   arm_int32_t                      collection_depth;
   arm_boolean_t                    show_private;
   arm_boolean_t                    show_secure;
   arm_boolean_t                    use_bind_thread;
   arm_boolean_t                    use_block;
   arm_boolean_t                    use_diagnostic;
   arm_boolean_t                    use_message_event;
   arm_boolean_t                    use_metric;
   arm_boolean_t                    use_user;
} arm_subbuffer_app_control_t;

/* ---------------- transaction ID control sub-buffer ------------------- */

typedef struct arm_subbuffer_tran_id_control
{
   const arm_subbuffer_t            header;
   arm_int32_t                control_count_app;
   arm_int32_t                      control_count_arm;
   arm_boolean_t                    control_used;
   const arm_id_t                   *tran_id;
   arm_int32_t                      collection_depth;
   arm_boolean_t                    use_bind_thread;
   arm_boolean_t                    use_block;
   arm_boolean_t                    use_diagnostic;
   arm_boolean_t                    use_message_event;
   arm_boolean_t                    use_metric;
   arm_boolean_t                    use_user;
} arm_subbuffer_tran_id_control_t;

/* -------------- transaction instance control sub-buffer --------------- */

typedef struct arm_subbuffer_tran_instance_control
{
   const arm_subbuffer_t            header;
   arm_int32_t                control_count_app;
   arm_int32_t                      control_count_arm;
   arm_boolean_t                    control_used;
   const arm_id_t                   *tran_id;
   const arm_correlator_t           *parent_correlator;
   arm_int32_t                      collection_depth;
   arm_boolean_t                    use_bind_thread;
   arm_boolean_t                    use_block;
   arm_boolean_t                    use_diagnostic;
   arm_boolean_t                    use_message_event;
   arm_boolean_t                    use_metric;
   arm_boolean_t                    use_user;
} arm_subbuffer_tran_instance_control_t;

#endif /* ARM41_H_INCLUDED */
```

16 <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 macro 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 Samples

16.1 Sample: Basic End-to-End Measurements

```
/* ---------------------------------------------------------------------- */
/* ----------- ARM 4.0 C API Example: using basic capabilities  ----------- */
/* ---------------------------------------------------------------------- */

#include <stdio.h>

/* from ARM 4.1 on, use arm41.h INSTEAD */
#include "arm4.h"

arm_id_t application_id;
arm_id_t parent_transaction_id, child_transaction_id;
arm_app_start_handle_t application_handle;

/* In-process nested transaction. The return value signals
 * successful completion if 0, failed completion otherwise.
 */
int child_transaction(arm_correlator_t *parent_correlator);

int main(int argc, char *argv[])
{
   arm_correlator_t new_correlator;
   arm_tran_start_handle_t parent_transaction_handle;
   arm_tran_status_t child_status = ARM_STATUS_GOOD;

   /* Capability used: ARM registration and initialization */

   arm_register_application("ARM Spec Basic Example",
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &application_id);

   arm_register_transaction(&application_id,
       "ExampleParentTx", /* transaction name */
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &parent_transaction_id);

   arm_register_transaction(&application_id,
       "ExampleChildTx",
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &child_transaction_id);

   arm_start_application(&application_id,
      "Examples",      /* ARM group name */
      NULL, /* ARM instance name */
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &application_handle);

   /* Capability used: Measure response time and status */
   arm_start_transaction(application_handle,
      &parent_transaction_id, ARM_CORR_NONE,
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &parent_transaction_handle,
      &new_correlator);

   printf("In parent transaction, before child transaction\n");

   if (child_transaction(&new_correlator))
      child_status = ARM_STATUS_FAILED;

   printf("In parent transaction, after child transaction\n");

   arm_stop_transaction(parent_transaction_handle,
      child_status,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   /* ARM shutdown */
   arm_stop_application(application_handle,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   arm_destroy_application(&application_id,
      ARM_FLAG_NONE, ARM_BUF4_NONE);
}

/* In-process nested transaction. The return value signals
 * successful completion if 0, failed completion otherwise.
 */
int child_transaction(arm_correlator_t *parent_correlator)
{
   /* will be generated using arm_start_transaction() */
   arm_tran_start_handle_t child_transaction_handle;
   /* will be generated using arm_start_transaction() */
   arm_correlator_t new_correlator;
   /* this variable emulates the child processing result */
   arm_tran_status_t arm_status = ARM_STATUS_GOOD;

   /* Capability used: Measure response time and status */
   arm_start_transaction(application_handle,
      &child_transaction_id, parent_correlator,
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &child_transaction_handle,
      &new_correlator);

   /* Child transaction work is performed here.
    * When there are more nested transcation below this level,
    * the contents of new_correlator could be passed to
    * a child of this transaction.
    *
    * In the pseudo code below it is assumed that a child transaction
    * level exists that is invoked remotely using synchronous send/receive
    * calls.
    *
    * Also note how the effective length of the correlator content
    * is determined to limit the amount of data transferred.
    */
#ifdef GRANDCHILD_REMOTE_TRANSACTION_EXISTS
   {
      /* assume mymsg_t contains a byte buffer and length field
       * for carrying a correlator.
       */
      mymsg_t msg;
      arm_correlator_length_t new_corr_len = 0;

      arm_get_correlator_length(&new_correlator, &new_corr_len);

      memcpy(msg.corr_buff, new_correlator.opaque, new_corr_len);
      msg.corr_len = new_corr_len;

      /* assuming socket_descr is initialized */
      send(socket_descr, &msg, sizeof(msg), 0);
      /* recv etc. */
   }
#endif /* GRANDCHILD_REMOTE_TRANSACTION_EXISTS */

   arm_stop_transaction(child_transaction_handle,
      arm_status,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   return (ARM_STATUS_GOOD == arm_status) ? 0 : -1;
}
```

16.2 Sample: Detailed Timing and Threading Measurements

```
/* ----------------------------------------------------------------------- */
/* ----- ARM 4.0 C API Example: using timing/threading capabilities  ------ */
/* ----------------------------------------------------------------------- */

#include <stdio.h>

/* from ARM 4.1 on, use arm41.h INSTEAD */
#include "arm4.h"

arm_id_t application_id;
arm_id_t parent_transaction_id, child_transaction_id;
arm_app_start_handle_t application_handle;

/* A pseudo dispatcher function. Conceptually, this function
 * creates a new thread, lets the childTransaction() function
 * execute in this thread, blocks waiting for completion
 * and finally returns the execution result to the caller.
 */
int dispatch_child_transaction_to_thread();

/* A pseudo remote procedure call */
void perform_rpc();

/* In-process nested transaction. The return value signals
 * successful completion if 0, failed completion otherwise.
 * See comments in dispatch_child_transaction_to_thread()
 * for an explanation of the arrival_time parameter.
 */
int child_transaction(arm_correlator_t *parent_correlator,
                      arm_arrival_time_t arrival_time);

int main(int argc, char *argv[])
{
   arm_correlator_t new_correlator;
   arm_tran_start_handle_t parent_transaction_handle;
   arm_tran_status_t child_status = ARM_STATUS_GOOD;
   int child_execution_count = 5;
   /* Capability used: ARM registration and initialization */

   arm_register_application("ARM Spec Threading Example",
                            ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
                            &application_id);

   arm_register_transaction(&application_id,
                            "ThreadExampleParentTx", /* transaction name */
                            ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
                            &parent_transaction_id);

   arm_register_transaction(&application_id,
                            "ThreadExampleChildTx",
                            ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
                            &child_transaction_id);

   arm_start_application(&application_id,
                         "Examples",      /* ARM group name */
                         NULL, /* ARM instance name */
                         ARM_FLAG_NONE, ARM_BUF4_NONE,
                         &application_handle);

   /* Capability used: Measure response time and status */
   arm_start_transaction(application_handle,
                         &parent_transaction_id, ARM_CORR_NONE,
                         ARM_FLAG_NONE, ARM_BUF4_NONE,
                         &parent_transaction_handle,
                         &new_correlator);

   printf("In parent transaction, before child transaction\n");

   /* see comments in dispatch_child_transaction_to_thread() */
   if (dispatch_child_transaction_to_thread(&new_correlator))
      child_status = ARM_STATUS_FAILED;

   printf("In parent transaction, after child transaction\n");

   arm_stop_transaction(parent_transaction_handle,
                        child_status,
                        ARM_FLAG_NONE, ARM_BUF4_NONE);

   /* ARM shutdown */
   arm_stop_application(application_handle,
                        ARM_FLAG_NONE, ARM_BUF4_NONE);

   arm_destroy_application(&application_id,
                           ARM_FLAG_NONE, ARM_BUF4_NONE);
}

/* A pseudo dispatcher function. Conceptually, this function
 * creates a new thread or selects one from a thread pool,
 * lets the childTransaction() function execute in this thread,
 * blocks waiting for completion
 * and finally returns the execution result to the caller.
 */
int dispatch_child_transaction_to_thread(arm_correlator_t *parent_correlator)
{
   /* filled by arm_get_arrival_time() */
   arm_arrival_time_t opaque_arrival_time;
   /* This code location - before the thread select/dispatch - is considered
    * the effictive start time of the transaction. Therefore, the arrival
    * time is taken here.
    */
   arm_get_arrival_time(&opaque_arrival_time);

#ifdef SYNCHRONOUSLY_EXECUTE_FUNCTION_IN_THREAD_SUPPORTED
   /* Vreate a new thread or select one from a thread pool
    * This is pseudo code; the functions used below are imaginary.
    */
   thread_t child_thread = get_some_thread();
   /* Now let child_transaction() execute with the arguments
    * &parent_correlatr and opaque_arrival_time in the givien thread
    */

   return synchronously_execute_function_in_thread(child_thread,
                                                   child_transaction,
                                                   parent_correlator,
                                                   opaque_arrival_time);
#else
   /* for demonstration purposes, the function is called directly instead */
   return child_transaction(parent_correlator, opaque_arrival_time);
#endif /* SYNCHRONOUSLY_EXECUTE_FUNCTION_IN_THREAD_SUPPORTED */
}

/* A pseudo remote procedure call */
void perform_rpc()
{
   /* During the RPC, the caller is 'blocked' */
}

/* In-process nested transaction. The return value signals
 * successful completion if 0, failed completion otherwise.
 */
int child_transaction(arm_correlator_t *parent_correlator,
                      arm_arrival_time_t arrival_time)
{
   /* will be generated using arm_start_transaction() */
   arm_tran_start_handle_t child_transaction_handle;
   /* will be generated using arm_start_transaction() */
   arm_correlator_t new_correlator;
   /* this variable emulates the child processing result */
   arm_tran_status_t arm_status = ARM_STATUS_GOOD;
   /* used for pairing arm_block()/arm_unblock() calls */
   arm_tran_block_handle_t block_handle;

   /* Capability used: Measure response time and status */
   arm_start_transaction(application_handle,
      &child_transaction_id, parent_correlator,
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &child_transaction_handle,
      &new_correlator);

   /* Capability used: Associate transactions to threads */
   /* In this example, several child_transaction() calls are assumed to
    * be executed by different threads.
    */
   arm_bind_thread(child_transaction_handle,
      ARM_FLAG_NONE, ARM_BUF4_NONE);
   /* Indicate a blocked status during a RPC */
   arm_block_transaction(child_transaction_handle,
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &block_handle);

   perform_rpc();

   arm_unblock_transaction(child_transaction_handle,
      block_handle,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   /* Not strictly necessary in this example, because an implicit
    * unbind_thread() will be performed by arm_stop_transaction() */
   arm_unbind_thread(child_transaction_handle,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   arm_stop_transaction(child_transaction_handle,
      arm_status,
      ARM_FLAG_NONE, ARM_BUF4_NONE);

   return (ARM_STATUS_GOOD == arm_status) ? 0 : -1;
}
```

16.3 Sample: Messaging

```
/* ----------------------------------------------------------------------- */
/* --------- ARM 4.1 C API Example: using messaging capabilities  --------- */
/* ----------------------------------------------------------------------- */

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

/* ARM 4.1 features are used in this example */
#include "arm41.h"

/* a pseudo message containing correlator data only. */
typedef struct pseudo_msg
{
   arm_correlator_length_t actual_correlator_length;
   /* in a real-world application, the size of correlator_bytes[] would
    * be actual_correlator_length.
    */
   arm_uint8_t correlator_bytes[ARM_CORR_MAX_LENGTH];
} pseudo_msg_t;

arm_id_t application_id;
arm_id_t parent_transaction_id, child1_transaction_id, child2_transaction_id;
arm_app_start_handle_t application_handle;

/* for this example, the two message instances below serve as
 * message forwarding substitute (1-element queues)
 */
pseudo_msg_t message_sent_from_parent;
pseudo_msg_t message_sent_from_child1;

/* In-process child transactions (levels 1 and 2). Although for demonstration
 * purposes, this function is called synchronously from main(), its
 * execution should be regarded as asynchronous message send/receive
 * processing.
 */
void child_transaction1(void);
void child_transaction2(void);

int main(int argc, char *argv[])
{
   arm_correlator_t new_correlator;
   arm_tran_start_handle_t parent_transaction_handle;
   arm_correlator_length_t new_corr_len = 0;

   arm_message_sent_event_t msg_tx_ev_array[1] = {{
      1,                 /* sent message count */
      "parent message"   /* description */
   }};

   arm_subbuffer_message_sent_event_t sb_msg_tx_ev =
   {
      {ARM_SUBBUFFER_MESSAGE_SENT_EVENT},
      ARM_FALSE, /* end_of_flow */
      sizeof(msg_tx_ev_array)/sizeof(arm_message_sent_event_t),
      msg_tx_ev_array
   };

   arm_subbuffer_t *subbuffers[] = {ARM_SB(sb_msg_tx_ev)};

   arm_buffer4_t buffer4 = {
         sizeof(subbuffers)/sizeof(arm_subbuffer_t*),
         subbuffers
   };

   /* Capability used: ARM registration and initialization */

   arm_register_application("ARM Spec Basic Example",
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &application_id);

   arm_register_transaction(&application_id,
       "ExampleParentTx", /* transaction name */
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &parent_transaction_id);

   arm_register_transaction(&application_id,
       "ExampleChildTx1",
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &child1_transaction_id);

   arm_register_transaction(&application_id,
       "ExampleChildTx2",
       ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
       &child2_transaction_id);

   arm_start_application(&application_id,
      "Examples",      /* ARM group name */
      NULL, /* ARM instance name */
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &application_handle);

   /* Capability used: Measure response time and status */
   arm_start_transaction(application_handle,
        &parent_transaction_id, ARM_CORR_NONE,
        ARM_FLAG_NONE, ARM_BUF4_NONE,
        &parent_transaction_handle,
        &new_correlator);


    printf("In parent transaction, before child transaction\n");

    /* contruct a message to child1, send() is implied here
     * The copying, shown here to clarify the steps) could be avoided if
     * message_sent_from_parent.correlator_bytes is used directly instead
     * of new_correlator in the arm_start_transaction() call.
     */
    arm_get_correlator_length(&new_correlator,
        &message_sent_from_parent.actual_correlator_length);
    memcpy(message_sent_from_parent.correlator_bytes,
            new_correlator.opaque,
            message_sent_from_parent.actual_correlator_length);

    /* Capability used: ARM messaging */
    /* signal a messge sent event to the ARM library: buffer4 contains
     * the prepared arm_subbuffer_message_sent_event_t.
     */
    arm_update_transaction(parent_transaction_handle,
        ARM_FLAG_NONE, &buffer4);

    /* Techically, the child transaction functions are invoked synchronously.
     * This ensures that all processing in child_transaction1() is finished
     * when child_transaction2() begins, and both are finished before
     * the application ends.
     * Semantically, the child transaction functions "collect and process
     * messages" asynchronously.
     */
    child_transaction1();
    child_transaction2();

    /* In this example, the stop time of the parent transaction is not
     * relevant for calculating the parent/child transaction flow. Still,
     * calling arm_stop_transaction() for every arm_start_transaction()
     * is mandatory.
     */
    arm_stop_transaction(parent_transaction_handle,
        ARM_STATUS_GOOD,
        ARM_FLAG_NONE, ARM_BUF4_NONE);

    /* ARM shutdown */
    arm_stop_application(application_handle,
        ARM_FLAG_NONE, ARM_BUF4_NONE);

    arm_destroy_application(&application_id,
        ARM_FLAG_NONE, ARM_BUF4_NONE);
}

/* In-process child transaction (level 1). Although for demonstration
 * purposes, this function is called synchronously from main(), its
 * execution should be regarded as asynchronous message send/receive
 * processing.
 */
void child_transaction1(void)
{
```

```
/* will be generated using arm_start_transaction() */
arm_tran_start_handle_t child_transaction_handle;
/* will be generated using arm_start_transaction() */
arm_correlator_t new_correlator;
arm_correlator_length_t new_corr_len = 0;

/* Capability used: ARM messaging */
arm_message_rcvd_event_t msg_rx_ev_array[1] = {{
   ARM_CORR_NONE,     /* parent correlator, will be filled in below */
   "parent message"   /* description */
}};

arm_subbuffer_message_rcvd_event_t sb_msg_rx_ev =
{
   {ARM_SUBBUFFER_MESSAGE_RCVD_EVENT},
   ARM_FALSE, /* end_of_flow */
   sizeof(msg_rx_ev_array)/sizeof(arm_message_rcvd_event_t),
   msg_rx_ev_array
};

arm_message_sent_event_t msg_tx_ev_array[1] = {{
   1,                 /* sent message count */
   "child1 message"   /* description */
}};

arm_subbuffer_message_sent_event_t sb_msg_tx_ev =
{
   {ARM_SUBBUFFER_MESSAGE_SENT_EVENT},
   ARM_FALSE, /* end_of_flow */
   sizeof(msg_tx_ev_array)/sizeof(arm_message_sent_event_t),
   msg_tx_ev_array
};

arm_subbuffer_t *subbuffers[] = {ARM_SB(sb_msg_rx_ev)};

arm_buffer4_t buffer4 = {
      sizeof(subbuffers)/sizeof(arm_subbuffer_t*),
      subbuffers
};


/* assume this transaction is triggered by the receipt of a message */

/* prepare message received event buffer. The correlator data could also
 * be copied from the message, depending on data memory layout.
 */
msg_rx_ev_array[0].received_correlator =
   (arm_correlator_t *)message_sent_from_parent.correlator_bytes;

/* Capability used: ARM messaging */
arm_start_transaction(application_handle,
   &child1_transaction_id,
   /* The correlator is passed via the message received event sub-buffer */
   ARM_CORR_NONE,
   ARM_FLAG_NONE, &buffer4,
   &child_transaction_handle,
   &new_correlator);

/* prepare a "message" to be sent to child2 */
/* The correlator bytes have already been created in the message struct */
```

```
   arm_get_correlator_length(&new_correlator,
      &message_sent_from_child1.actual_correlator_length);

   /* In this example, the stop time of the child1 transaction coincides
    * with the (assumed) sending of a message to child2, therefore the message
    * received sub-buffer is provided with arm_stop_transaction().
    */

   /* Capability used: ARM messaging */
   subbuffers[0] = ARM_SB(sb_msg_tx_ev); /* re-using subbufers[] */
   arm_stop_transaction(child_transaction_handle,
      ARM_STATUS_GOOD,
      ARM_FLAG_NONE, &buffer4);
}

/* In-process child transaction (level 2). Although for demonstration
 * purposes, this function is called synchronously from main(), its
 * execution should be regarded as asynchronous message send/receive
 * processing.
 */
void child_transaction2(void)
{
   /* will be generated using arm_start_transaction() */
   arm_tran_start_handle_t child_transaction_handle;

   /* Capability used: ARM messaging */
   arm_message_rcvd_event_t msg_rx_ev_array[1] = {{
      ARM_CORR_NONE,     /* parent correlator, will be filled in below */
      "child1 message"   /* description */
   }};

   arm_subbuffer_message_rcvd_event_t sb_msg_rx_ev =
   {
      {ARM_SUBBUFFER_MESSAGE_RCVD_EVENT},
      ARM_FALSE, /* end_of_flow */
      sizeof(msg_rx_ev_array)/sizeof(arm_message_rcvd_event_t),
      msg_rx_ev_array
   };

   arm_subbuffer_t *subbuffers[] = {ARM_SB(sb_msg_rx_ev)};

   arm_buffer4_t buffer4 = {
         sizeof(subbuffers)/sizeof(arm_subbuffer_t*),
         subbuffers
   };

   /* assume this transaction is triggered by the receipt of a message */

   /* prepare message received event buffer. The correlator data could also
    * be copied from the message, depending on data memory layout.
    */
   msg_rx_ev_array[0].received_correlator =
      (arm_correlator_t *)message_sent_from_child1.correlator_bytes;

   /* this transaction signals the end of a flow chain */
   sb_msg_rx_ev.end_of_flow = ARM_TRUE;

   /* Capability used: ARM messaging */
   arm_start_transaction(application_handle,
      &child1_transaction_id,
      /* The correlator is passed via the message received event sub-buffer */
      ARM_CORR_NONE,
      ARM_FLAG_NONE, &buffer4,
      &child_transaction_handle,
      ARM_CORR_NONE);

   arm_stop_transaction(child_transaction_handle,
      ARM_STATUS_GOOD,
      ARM_FLAG_NONE, ARM_BUF4_NONE);
}
```

16.4 Sample: Instrumentation Control

```
/* ------------------------------------------------------------------------ */
/* --------- ARM 4.0 C API Example: using instrumentation control --------- */
/* ------------------------------------------------------------------------ */

#include <stdio.h>

/* ARM 4.1 features are used in this example */
#include "arm41.h"

#define TOTAL_CHILD_CALLS 10000     /* call child_transaction N times */
#define CHECK_CONTROL_INTERVAL 1000 /* check every N transactions */

arm_id_t application_id;
arm_id_t parent_transaction_id, child_transaction_id;
arm_app_start_handle_t application_handle;

/* gobal flags reflecting the instrumentation control state */
arm_boolean_t enable_global;
arm_boolean_t enable_child_tran;

/* Capability used: ARM instrumentation control */
/* For simplicity, the buffer results are made globally accessible */
arm_subbuffer_app_control_t sb_app_control =
{
   {ARM_SUBBUFFER_APP_CONTROL},
   ARM41_APP_CONTROL_COUNT,
   0,
};

/* In-process nested transaction. The return value signals
 * successful completion if 0, failed completion otherwise.
 */
int child_transaction(arm_correlator_t *parent_correlator);

int main(int argc, char *argv[])
{
   arm_correlator_t new_correlator;
   arm_tran_start_handle_t parent_transaction_handle;
   arm_tran_status_t child_status = ARM_STATUS_GOOD;

   arm_subbuffer_t *subbuffers[] = {ARM_SB(sb_app_control)};

   arm_buffer4_t buffer4 = {
         sizeof(subbuffers)/sizeof(arm_subbuffer_t*),
         subbuffers
   };
```

```
/* Capability used: ARM registration and initialization */

arm_register_application("ARM Spec Basic Example",
    ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
    &application_id);

arm_register_transaction(&application_id,
    "ExampleParentTx", /* transaction name */
    ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
    &parent_transaction_id);

arm_register_transaction(&application_id,
    "ExampleChildTx",
    ARM_ID_NONE, ARM_FLAG_NONE, ARM_BUF4_NONE,
    &child_transaction_id);

/* Capability used: ARM instrumentation control */

sb_app_control.app_control_used = ARM_FALSE; /* required */
arm_start_application(&application_id,
   "Examples",      /* ARM group name */
   NULL, /* ARM instance name */
   ARM_FLAG_NONE,
   /* contains the application control sub-buffer */
   &buffer4,
   &application_handle);

/* Note: Real-world applications should check control_count_arm */

/* Default: if no instrumentation control is provided, enable everything */
enable_global = ARM_TRUE;
enable_child_tran = ARM_TRUE;
if (ARM_TRUE == sb_app_control.app_control_used)
{
   /* For simplicity, this example interprets only on/off granularity */
   if (sb_app_control.collection_depth <= ARM_COLLECTION_DEPTH_NONE)
   {
      enable_global = ARM_FALSE;
      enable_child_tran = ARM_FALSE;
   }
}
else
{
   /* signal absence of control for nested checks */
   sb_app_control.tran_id_control_used = ARM_FALSE;
}

if (ARM_TRUE == enable_global)
   arm_start_transaction(application_handle,
      &parent_transaction_id, ARM_CORR_NONE,
      ARM_FLAG_NONE, ARM_BUF4_NONE,
      &parent_transaction_handle,
      &new_correlator);

printf("In parent transaction, before child transaction\n");

if (child_transaction(&new_correlator))
   child_status = ARM_STATUS_FAILED;
```

```
      printf("In parent transaction, after child transaction\n");

      if (ARM_TRUE == enable_global)
         arm_stop_transaction(parent_transaction_handle,
            child_status,
            ARM_FLAG_NONE, ARM_BUF4_NONE);

      /* ARM shutdown */
      arm_stop_application(application_handle,
         ARM_FLAG_NONE, ARM_BUF4_NONE);

      arm_destroy_application(&application_id,
         ARM_FLAG_NONE, ARM_BUF4_NONE);
   }

   /* In-process nested transaction. The return value signals
    * successful completion if 0, failed completion otherwise.
    */
   int child_transaction(arm_correlator_t *parent_correlator)
   {
      /* will be generated using arm_start_transaction() */
      arm_tran_start_handle_t child_transaction_handle;
      /* will be generated using arm_start_transaction() */
      arm_correlator_t new_correlator;
      /* this variable emulates the child processing result */
      arm_tran_status_t arm_status = ARM_STATUS_GOOD;

      /* Capability used: ARM instrumentation control */
      arm_subbuffer_tran_id_control_t sb_tran_id_control =
      {
         {ARM_SUBBUFFER_TRAN_ID_CONTROL},
         ARM41_TRAN_ID_CONTROL_COUNT,
         ARM_FALSE,
         0,
         &child_transaction_id, /* query control state for this ID */
      };
      arm_subbuffer_t *subbuffers[] = {ARM_SB(sb_tran_id_control)};

      arm_buffer4_t buffer4 = {
            sizeof(subbuffers)/sizeof(arm_subbuffer_t*),
            subbuffers
      };

      int i;

      for(i=0; i<TOTAL_CHILD_CALLS; ++i)
      {
         /* Check for transaction ID level control, if supported and
          * interval count is reached .
          */
         if ((ARM_TRUE == sb_app_control.tran_id_control_used)&&
            (0 == i%CHECK_CONTROL_INTERVAL))
         {
            sb_tran_id_control.control_used = ARM_FALSE; /* required */
            arm_generate_correlator(
               application_handle,
               &child_transaction_id, /* ignored */
               ARM_CORR_NONE,         /* ignored */
               ARM_FLAG_NONE,
               &buffer4,
               ARM_CORR_NONE);        /* required to signal special purpose */

            /* Note: Real-world applications should check control_count_arm */
            if (ARM_TRUE == sb_tran_id_control.control_used)
            {
               /* override application control settings */
               if (sb_tran_id_control.collection_depth <=
   ARM_COLLECTION_DEPTH_NONE)
                  enable_child_tran = ARM_FALSE;
               else
                  enable_child_tran = ARM_TRUE;
            }
         }

         if (ARM_TRUE == enable_child_tran)
            arm_start_transaction(application_handle,
               &child_transaction_id, parent_correlator,
               ARM_FLAG_NONE, ARM_BUF4_NONE,
               &child_transaction_handle,
               &new_correlator);

         /* Child transaction work is performed here.*/

         if (ARM_TRUE == enable_child_tran)
            arm_stop_transaction(child_transaction_handle,
               arm_status,
               ARM_FLAG_NONE, ARM_BUF4_NONE);
      }

      return (ARM_STATUS_GOOD == arm_status) ? 0 : -1;
   }
```

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.

B.1 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

B.2 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

B.3 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.

B.4 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.

B.5 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.

B.6 ARM Correlator Format Constraints

These constraints apply to all formats.

Table 16: 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, 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 100 – Reserved for MyARM's implementation 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

Position	Length	Contents
Byte 3	1 byte	Flags All eight-bit flags are reserved by the ARM specification. Four flags are defined in positions 0:3 (the highest order bits), as in *abcd*0000, where *a*, *b*, *c*, and *d* 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. c = 1 if the control flow from the parent was accomplished via an asynchronous mechanism such as a messaging protocol *or* that the transaction flow is asynchronous in nature. See Section 4.2.1 for a more complete description. *d* = 1 indicates that ARM-reported program logic execution in the child recipient of this correlator reflects transaction work that, although initiated by this parent, is performed independently of the parent's ARM-reported transaction work in scope and purpose. See Section 4.2.2 for a more complete description. It logically follows that the child transaction is executing asynchronously to the parent transaction and therefore *d*=1 only if *c*=1. If *c*=0, *d* is ignored.

Index