CAE Specification

Systems Management:
Universal Measurement Architecture (UMA)

The Open Group

CAE Specification

Systems Management: Universal Measurement Architecture (UMA)

ISBN: 1-85912-117-9
Document Number: C427

Published in the U.K. by The Open Group, February 1997.

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

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

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE 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.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This is a CAE Specification. It presents a compendium of the Universal Measurement Architecture (UMA) documents, which comprises a UMA Guide plus a set of three specifications. The 4 documents are:

- UMA Guide (see Part 1 of this specification).

  This Guide reviews the issues surrounding performance measurement in Open Systems, describes the general UMA architecture, and discusses the relationships that the UMA has with other technologies. This UMA Guide is also available separately.

- UMA Measurement Layer Interface (MLI) specification (see Part 2 of this specification).

  This defines the functional characteristics of the MLI, and the underlying semantics and function calls that implement them. It also defines a format for headers appended to measurement data captured through the DCI.

- UMA Data Capture Interface (DCI) specification (see Part 3 of this specification).

  This is the interface between the data capture layer and the measurement control layer of the UMA architecture.

- UMA Data Pool Definitions (see Part 4 of this specification).

  The data pool defines a set of performance metrics which may be accessed by the two UMA interfaces.

Audience

The UMA Guide is intended for those who wish to gain an introduction to the issues involved in performance measurement. It is also intended as an introduction to the UMA and the associated MLI, DCI and DPD specifications.

The target audience for the UMA Specifications is both system designers who need to implement the respective interfaces, and performance professionals who need to understand how to deploy these interfaces to optimium advantage.

Structure

- Part 1: UMA Guide
- Part 2: Measurement Layer Iinterface (MLI) CAE Specification
- Part 3: Data Capture Interface (DCI) CAE Specification
- Part 4: Data Pool Definitions (DPD) CAE Specification

Trade Marks

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Limited.

Referenced Documents

The following documents are referenced in this guide:

ASN.1
: ISO 8824: 1990, Information Technology — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).

BER
: ISO / IEC 8825: 1990 (ITU-T Recommendation X.209 (1988)), Information Technology — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

DMI
: ISO / IEC 10165-2: 1992, Information Technology — Open Systems Interconnection — Structure of Management Information — Part 2: Definition of Management Information.

GDMO
: ISO / IEC 10165-4:1992, Information Technology — Open Systems Interconnection — Structure of Management Information — Part 4: Guidelines for the Definition of Managed Objects.

GSS
: CAE Specification, 1995, Generic Security Service API (GSS-API) Base (ISBN:1-85912-131-4).

NMF Strategy for Migration to GDMO
: OSI / NM Forum Strategy for Migration to GDMO; Forum 010, Issue 1.0, January 1991.

SMF
: ISO / IEC 10164: 1992 Information Technology — Open Systems Interconnection — Systems Management (Parts 1 to 13 inclusive).

XMP
: CAE Specification, March 1994, Systems Management: Management Protocol API (ISBN 1-85912-027-X, C306).

XOM, Issue 2
: CAE Specification, February 1994, OSI-Abstract-Data Manipulation API (XOM), Issue 2 (ISBN: 1-85912-008-3, C315).

XSH, Issue 5
: CAE Specification, January 1997, System Interfaces and Headers, Issue 5 (ISBN: 1-85912-181-0, C606).

UMA Specifications

MLI

CAE Specification, January 1997 — Systems Management: UMA Measurement Layer Interface, Part 2 of ‘‘Universal Measurement Architecture (UMA)’’ CAE Specification (ISBN 1-85912-117-9, C427).

Contained in Part 2 of this UMA specification.

DCI

CAE Specification, January 1997 — Systems Management: UMA Data Capture Interface, Part 3 of ‘‘Universal Measurement Architecture (UMA)’’ CAE Specification (ISBN 1-85912-117-9, C427).

Contained in Part 3 of this UMA specification.

DPD

CAE Specification, January 1997 — Systems Management: UMA Data Pool Definitions, Part 4 of ‘‘Universal Measurement Architecture (UMA)’’ CAE Specification (ISBN 1-85912-117-9, C427).

Contained in Part 4 of this UMA specification.

Guide

Part 1:
UMA Guide

The Open Group

Contents

List of Figures

Preface

This Document

This X/Open Guide reviews the problem space of performance measurement in open systems, and introduces the architecture, features and benefits provided by the three X/Open Universal Measurement Architecture (UMA) specifications:

- UMA Measurement Layer Interface (MLI) specification
- UMA Data Capture Interface (DCI) specification
- UMA Data Pool Definitions (DPD) specification

and serves as an introduction to these specifications.

These UMA specifications are published in Parts 2, 3 and 4 respectively of this document.

This UMA Guide is also available separately, as X/Open document number G507, ISBN 1-85912-122-5.

Audience

This Guide is intended for those who wish to gain an introduction to the issues involved in performance measurement.

It is also intended as an introduction to the X/Open UMA and the associated MLI, DCI and DPD specifications as listed above.

Structure

- **Chapter 1. Introduction** — outlines the issues involved in addressing performance measurement in open systems, and explains the scope and purpose of the X/Open UMA specifications.
- **Chapter 2, Overview of UMA Architecture** — explains the UMA architecture.
- **Chapter 3, Features and Benefits of the UMA Interfaces** — brings out the advantages of the UMA approach to performance measurement.
- **Chapter 4, UMA Data Pool** — describes particular features of the Data Pool within the UMA architecture.
- **Chapter 5, Distributed UMA** — addresses extensibility and interoperability issues associated with UMA.
- **Chapter 6, Relationship of UMA to Other Technologies** — discusses other technologies impacting the UMA space.

Acknowledgements

X/Open acknowledges the substantial contribution of the Performance Management Working Group (PMWG), initially under the sponsorship of UNIX Systems Laboratories (USL) and currently of the Computer Measurement Group (CMG), in the development of this Guide and the three associated UMA specifications (MLI, DCI and DPD).

PMWG contributors to this UMA Guide include:

Robert Berry, IBM Corporation, Austin, TX, USA
Ram Chelluri: AT&T/Global Information Solutions, Dayton, OH, USA
Jim Van Sciver: Open Software Foundation, Cambridge, MA, USA
Leon Traister†: Amdahl Corporation, Sunnyvale, CA, USA

0. Editor.

Chapter 1

Introduction

1.1 Performance Measurement in Open Systems

The commercialisation of POSIX-based computing is continuing at a rapid pace, adding capabilities not just expected, but desperately needed by MIS shops and new commercial users. One such feature is performance management. The users familiar with mainframe data processing environments are used to having sophisticated tools available to determine resource utilisation, predict system capacities and growth paths, and even to compare CPU models for making purchase decisions.

Although the open system concept is creating a revolution in applications development and system migration paths, certain capabilities (such as performance management) have not been standardised. Currently there is generally insufficient performance management functionality in Open Systems, and even where it does exist it is often provided in a different way on systems from different vendors.

Key areas of work in the development of the UMA specifications include performance data availability and interfaces for its collection. Until the data and interfaces are standardised, each computer vendor, performance software vendor, or large end user is faced with the task of kernel modification to collect the necessary data, development of a proprietary kernel interface to move the data to user-space, and development of custom performance monitoring and management software. Until such interfaces are standardised, few performance management tools will be built because of the cost of their migration between operating system versions or POSIX-based system implementations.

As open systems become the operating systems of choice for larger, faster, and more complex computer systems, there is an increased need to effectively manage these systems. But there exists little software to support performance management of these complex systems. For example, administrators of standard UNIX systems must rely on the system activity reporter (**sar**) data to manage their systems. However, such information is often insufficent in scope, inadequate in depth and cannot be properly controlled, especially by multiple performance management applications in distributed environments. Performance management of large applications, including databases, often has to rely on accounting data to measure activity, but such data can be inappropriate since it was intended for a different purpose.

There are several reasons for the lack of performance management software. One reason is that many of the desired metrics are not available. Another reason is the fear that release-to-release kernel changes will make it necessary to frequently modify performance-related applications. This discourages developers from using any but the most basic metrics or developing any but the most basic applications, particularly in cases where the application must execute on platforms supplied by different vendors. There are, furthermore, no well-defined interfaces for obtaining even the existing performance data from the kernel, and the current access methods are restrictive and expensive.

1.2 Issues addressed by the UMA

In this section, the reasons for the definition of the UMA are outlined in terms of the issues that have arisen with existing performance facilities in Open Systems.

1.2.1 Kernel Data

Extracting performance data from the kernel of an Open System has traditionally been done by methods which involve user level utilities accessing the kernel data structures. An example of this is the UNIX **/dev/kmem** interface which has historically been the primary interface used by UNIX System performance measurement utilities for extracting data from the kernel. This mechanism generally relies on the user level performance utility using the name of a particular data structure to derive from the symbol table the virtual address of the structure. It can then access the kernel data (using **/dev/kmem** in the case of UNIX) to seek to and read the value of that data structure. The advantage of this approach is its generality: if the address of a data structure can be found, its value can be read. But its generality is also a disadvantage. Since almost any data structure can be used to provide performance data, the tendency is to do so without regard to whether it is supported. This makes it very difficult to maintain a performance application across releases when data structures change. For example, programs such as **ps** and **sadc** have been notoriously difficult to maintain from release to release.

Processing Cost

The retrieval of each virtually contiguous piece of information requires a seek system call and a read system call to extract the information from the kernel. If there are many such pieces, the central processing unit (cpu) costs of gathering the information can be very high. Also, since each piece requires a separate seek and read, it is very hard to guarantee that the data obtained is consistent.

Access Permissions

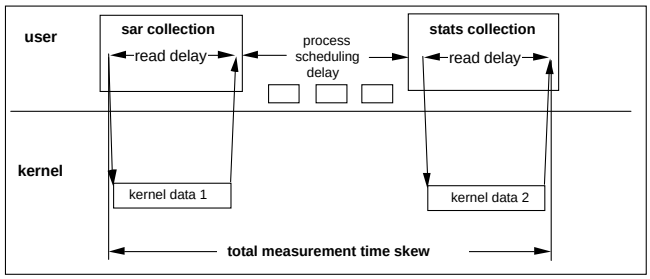
For security reasons kernel data is not set to be readable by ordinary users. Thus performance utilities (such as **ps** and **sadc** in the case of UNIX) must be run as privileged programs. Ordinary programs must invoke the performance utilities and read data either through pipes or files. This adds to the cost of accessing this information.

Binary Compatibility

In order to reduce the number of seeks and reads necessary to obtain the data, many metrics are combined into a single data structure (for example, **sysinfo** in UNIX). The result is that programs must be aware of the layout and contents of the data structure. If the data structure layout or content change significantly between releases, binary compatibility cannot be maintained; the programs must be recompiled with new headers that reflect the new data structure layout and contents.

Data Synchronisation

Using a variety of user space collectors to gather data can result in skewed collection times due to the scheduling delays for each process (see Figure 1-1 for a UNIX example). Hence if two user level utilities (for example **sar** and **stats** in the case of UNIX) obtain performance information that is then analysed as if it refers to the same time period, this skew means that the usefulness of the data is impaired. A common source of user level collection would reduce such time skews.




```
sar   |_____|_____|_____|_____|_____|_____|_____|_____|_____|

stats |_____|_____|_____|_____|_____|_____|_____|___ ___|_____|

clock |_____|_____|_____|_____|_____|_____|_____|_____|_____|
```

Figure 1-1 Collection Time Skew from Separate Collection Components

Data Applicability

The privileged utilities that collect kernel information needed for performance analysis is often oriented towards a particular use for the data. An example of this is the use of accounting information for performance analysis. The effect of this is that performance applications often get information they do not want, get it in the wrong form or cannot get it at all.

1.2.2 Measurement Applications

Existing performance measurement applications suffer from the lack of facilities specific to their requirements to obtain performance information. The issues in the previous section concerning kernel data obviously contribute to the problems faced by these applications but in addition there are general issues that apply.

Multiple Data Collection

There may be several measurement applications running, performing different analyses of performance information. It is commonly the case that there is no common collection mechanism between such applications, resulting in the same data being collected, distributed and stored separately by each application.

Control of Collection

Where there are several measurement applications running, each may try to control the way in which performance data is collected resulting in a conflict. So, for example, where a privileged program is invoked to collect performance information, one application may set the collection interval to one value and another may set it to a different value.

Methods of Collection

Where measurement applications have to use a variety of mechanisms to effect the collection of performance information, the writing of such applications is unnecessarily complex. Different methods have to be written to collect very similar data from different sources and provision must be made for additional methods to appear for different systems and new release.

Real Time Data

Measurement applications that wish to have access to real time data as opposed to historical data have to use different mechanisms. The effect of this is that data may be collected, distributed and stored more than once and that it is difficult to write an application that will work on both real time and historical data.

Events

By the nature of the mechanisms that are used to obtain performance information it is difficult to integrate events, and the information they contain, into the pool of performance information. Measurement applications should be able, if they wish, to access events as well as synchronously requested data.

New Information

Increasingly systems are becoming capable of dynamic reconfiguration (for example hot pull discs) and measurement applications need to be able to find out dynamically the objects that exist and the performance information they can supply. Measurement applications also need to have a mechanism by which they can be notified of changes that have occurred (that is, an event mechanism).

Data Description

Generic measurement applications need to be able to handle classes of objects without necessarily being aware of detailed differences between different classes of the same general type. So, for example, it should be possible for a measurement application to be able to use the performance information from any make and type of disc device. However, specialised applications should be able to make use of detailed information from a particular make of device.

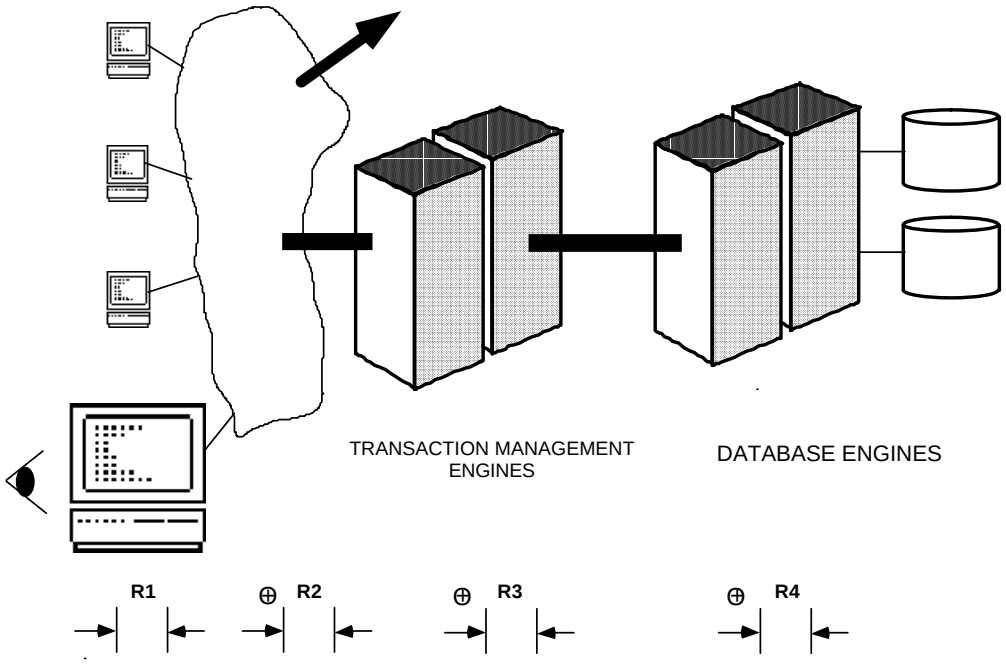


Figure 1-2 Components of a Distributed Transaction

1.2.3 Distributed systems

Finally, we must consider the distributed environment. In the past, performance analysis activities of a single platform at a time were meaningful because most, if not all, of the processing of a user interaction took place on a single platform. In the emerging open systems environment, however, this is no longer the case. Figure 1-2 illustrates the situation where a user interaction is serviced by processing on a number of platforms and in addition, these platforms may be supplied by a variety of vendors. In this case, the response time experienced by the user is dependent on the response times of the individual service platforms and on the response times of various network components. To be able to perform an analysis of response time requires that data be captured and tagged with identification at least at a transaction level and that there be a mechanism that can gather this data from distributed systems where it is captured[1].

1. The tagging of workload components is predominately the concern of provider instrumentation and the analysis of performance data is an issue for measurement applications; both are formally outside of the scope of UMA itself, which is focused on the control of data acquisition and on the delivery and management of performance data. UMA does provide a mechanism (UMAWorkInfo instances) for containing and transmitting a flexible number of workload identifiers which may include a transaction ID. It will be necessary to track emerging instrumentation methodologies and standards efforts from DCE, ISO, and OMG working groups to ensure that UMA remains capable of appropriate functionality in this area.

1.3 Scope and Purpose of UMA

To help address the above data collection issues and limitations, the following three specifications for Universal Measurement Architecture (UMA) have been developed:

- UMA Performance Measurement Data Pool (DPD)
- UMA Data Capture Interface (DCI)
- UMA Measurement Layer Interface (MLI).

This Guide describes the benefits and features of the Universal Measurement Architecture, and serves as an introduction to these UMA specification documents for those new to this architecture.

The Universal Measurement Architecture (UMA) provides support for the collection, management and reporting of performance data and events.

Its goals include:

- standardisation and portability of interfaces and data
- collection from both kernel and application sources
- distributed access - multiple system images
- control of collection overhead through common collection, configurable metrics and threshold filtering of data
- improved data capture synchronisation
- scalable and extensible services
- seamless access between historical and current data
- simple specification of interval and event data reporting.

UMA, therefore, may be considered as a powerful agent for collecting and managing performance data.

The following Chapters describe the interfaces and services in more detail.

Chapter 2

Overview of UMA Architecture

The UMA reference model defines four layers and two interfaces as shown in Figure 2-1.

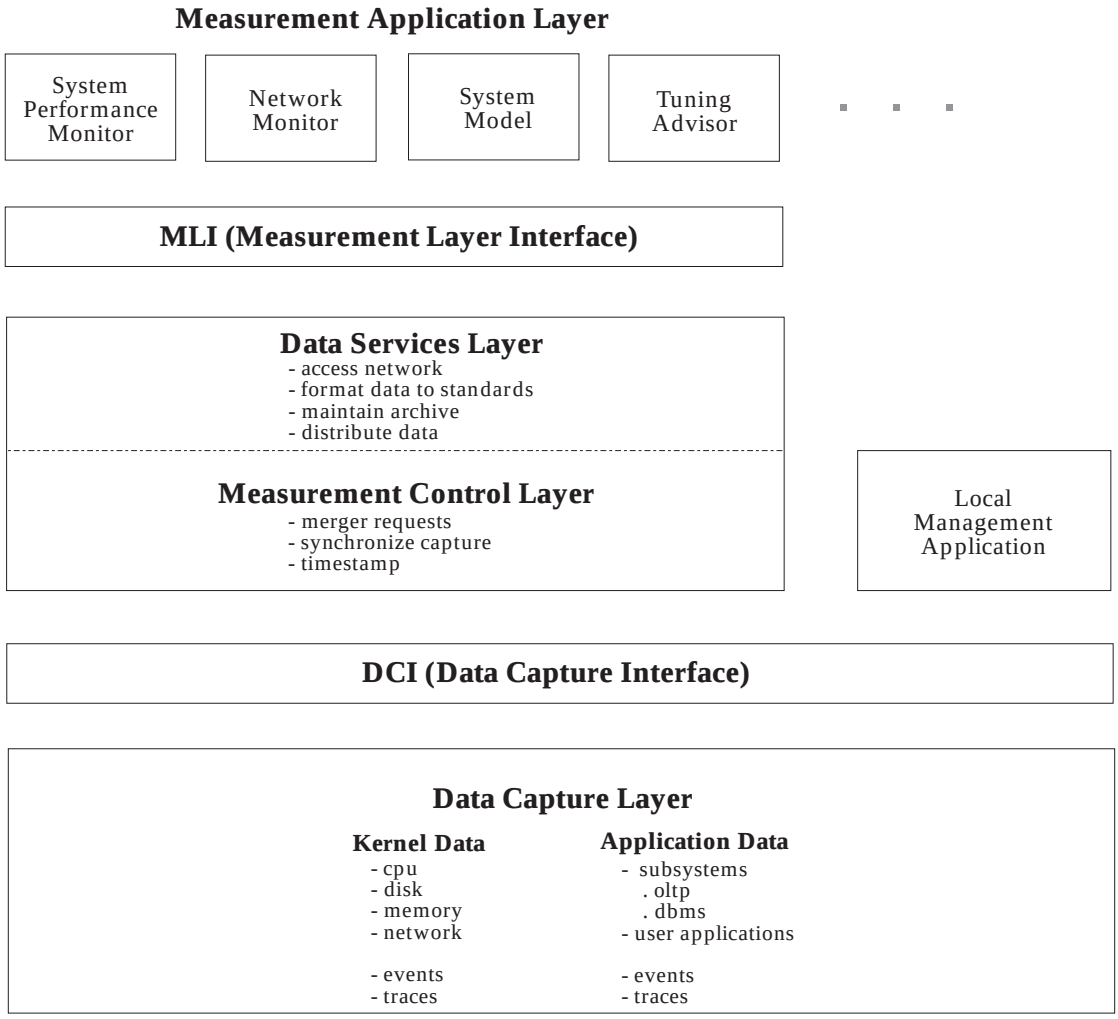


Figure 2-1 UMA Reference Model

In this Chapter, these layers and interfaces are briefly described from the bottom up, that is, starting from the Data Capture Layer.

2.1 Data Capture Layer

The Data Capture Layer is responsible for collecting raw data. Its architecture together with the Data Capture Interface (DCI) allow data from multiple sources to be obtained by a single consumer above the DCI, and this in turn improves the synchronisation of the data collection.

2.2 Data Capture Interface

The Data Capture Interface is the interface between the Measurement Control Layer and the Data Capture Layer. It provides the means for dynamically extending data collection to new providers such as databases without affecting existing programs.

2.3 Measurement Control Layer

The Measurement Control Layer schedules and synchronises data collection through the Data Capture Interface.

2.4 Data Services Layer

The Data Services Layer accepts measurement requests from Measurement Application Programs (MAPs) through the Measurement Layer Interface, and distributes data to the destination requested by the MAP. A destination may include, the MAP itself, a private file or the UMA Data Storage (UMADS), which will be described later.

A feature of UMA is that the interface between the Data Services Layer and the Measurement Control Layer is not formally specified. These two layers, though functionally distinct, and which constitute a logical service layer for the MLI, may be combined in some implementations.

2.5 Local Measurement Application

Where the facilities provided by the Measurement Control Layer and the Data Services Layer are not required, Local Measurement Applications can be provided which use the DCI directly. Such an application could also function as an agent for distributing performance data outside the scope of the UMA.

2.6 Measurement Layer Interface

The Measurement Layer Interface (MLI) is the interface between the Measurement Application Layer and the Data Services Layer. It provides the medium for all interactions between a MAP and UMA, thus isolating the application for the implementation details of the rest of UMA.

The Measurement Layer Interface allows transparent communication across networks, therefore a MAP running on one system can request and examine data from another system. Together with the Data Services Layer, it provides an infrastructure for the distribution of data over large numbers of heterogeneous sites and multiple platforms.

2.7 Measurement Application Layer

The Measurement Application Layer consists of the various Measurement Application Programs (MAPs) that provide services for technical support of management goals. These MAPs may consist of performance monitors, capacity planning tools, tuning advisors, and so on.

Chapter 3

Features and Benefits of the UMA Interfaces

This Chapter provides further description of the interfaces defined in the two UMA interface specifications:

- the UMA Data Capture Layer Interface (DCI) specification (see reference **DCI**)
- the UMA Measurement Layer Interface (MLI) specification (see reference **MLI**).

It also describes how they relate to one another.

3.1 Features and Benefits of the DCI

The Data Capture Interface (DCI) is the lowest architectural layer in the Universal Measurement Architecture (UMA). This section will describe the DCI and the services provided by the DCI, give an understanding of the problems solved by this layer, and the problems that the DCI was not meant to solve.

The DCI is a collection of programming interfaces. The DCI specification defines the set of DCI interfaces and the arguments and return values for those interfaces. The DCI specification also defines the service provided by these programming interfaces.

3.1.1 Performance Management and the DCI

The DCI addresses several important problems in the performance management arena:

- it provides a consistent interface between system functions that are providing metrics and those functions that consume these metrics
- it allows any system entity, applications, daemons, or the operating system to provide metrics
- it separates the metric source from the method for acquiring the metrics. This allows metric consumers to use a uniform acquisition method regardless of source.

One of the problems the DCI was not meant to solve is the transmission of data across the network. The DCI interfaces explicitly limit their scope to metric transmission between providers and consumers on the same system. The reason for this scope limitation is that the intersystem metric transmission problem is already addressed by both the higher UMA architectural level (MLI and Data Services Layer) and by existing solutions, such as SNMP.

In summary, the DCI is a relatively simple collection of interfaces to provide a uniform mechanism for transmitting and collecting performance information from any system entity, from the operating system to applications. Its primary benefits are the standardisation of the collection interface, the elimination of prior knowledge of the metrics being collected, and use of a uniform access mechanism regardless of metric source.

3.1.2 DCI Service

The service provided by this API (Application Programming Interface) is twofold. First, the DCI acts as a connection broker between those system components which produce metrics *(metric providers)* and those system components which consume metrics *(metric consumers)*. Second, the DCI provides a repository, called the DCI name space, for metric providers to store information about the set of available metrics. Metrics consumers can traverse and interrogate the DCI name space to find out information about the available metric set. It is not the metrics that are stored in the name space, instead it is information about the metrics; the metrics themselves are managed and supplied by the individual metrics providers. The DCI structure and the client/server relationships are illustrated in Figure 3-1.

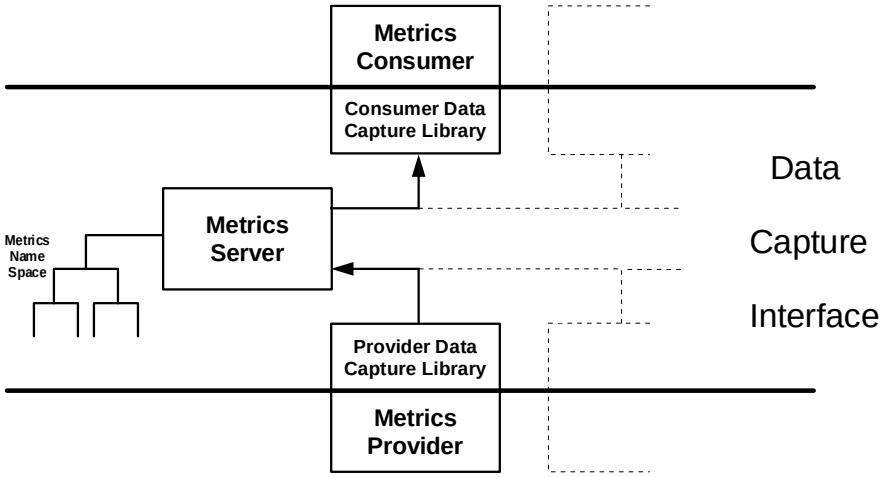


Figure 3-1 DCI Structure and Client/Server Relationships

To clarify the relationship of this figure with the one previous showing the full UMA architecture, it should be observed that when an implementation includes the MLI, the UMA Data Services and Measurement Control Layers (which are the MLI service layers) play the role of a DCI metrics consumer.

3.1.3 Name Space

Through use of the DCI, performance applications and MLI service consumers can traverse the name space and find out what type of metrics are available, the units and data types of individual metrics, the number and type of available measured objects, and human readable descriptive labels for both the metrics and measured objects.

Through the use of wildcards in the description of a metric, a metric consumer can request multiple metrics in one call. This enables the cost of delivering the metrics to be reduced and the skew between metrics to be minimised.

In the terminology used by the DCI specification, the metrics name space contains names for metric classes and instances of those classes. A metric class is a grouping of metrics and the information used to describe that metric set. These *metric attributes* are such things as units and data types. A metric instance is a representation of a measured object, such as a disk. Thus there can be a metric class which describes disk I/O metrics and that class can have five instances, one

for each of the system's disk drives. A metrics consumer can find out about the metrics by reading the metric class attributes. The consumer can then read the disk performance data for one or all class instantiations. It is at this point the DCI's connection broker service comes into play. The metrics consumer does not require prior knowledge of which provider supports the desired metric set nor does it need to know the mechanics of how those metrics are delivered. This is all handled by the DCI service and the relationship it maintains with the set of system providers.

3.1.4 Polled Metric and Event Support

There are two types of data supported by the DCI: polled metrics and events. The distinction between the two is whether the consumer or provider is primarily responsible for metric delivery. In the case of polled metrics, the consumer gets metrics at whatever rate is convenient. In the case of event metrics, consumers must wait for providers to deliver events as these events occur. Traces in UMA are implemented as high frequency events and are normally directed to a file by the DCI consumer.

3.2 Features and Benefits of the MLI

The Measurement Layer Interface (MLI) is an application programming interface (API) and a set of services (the UMA Data Services Layer) that simplify the implementation of measurement application programs (MAPs) in a distributed environment.

Note: In the following discussion we refer to both polled data and events as *data*.

The MLI implements the following aspects of the UMA architecture:

- allows simple specification of polled data and event collection parameters,
- establishes a consistent message architecture for UMA data, and data is available in simply parsed structures
- manages the distributed collection, reporting and recording of current and historical data
- provides synchronised capture of data
- implements filtering of data based on selection criteria and thresholds to minimise network traffic
- implements seamless switching between current and historical data.

3.2.1 Data Collection, Reporting and Recording

Through the MLI, a MAP can specify the types and characteristics of data to be reported to a MAP. UMA distinguishes between the *reporting* of data to a MAP and data collection. A MAP requests data from a specified source to be *reported* to a specified destination. The UMA services act on behalf of a MAP to perform the actual data *collection* through the Data Capture Interface (DCI). Performance overhead is minimised by making use of existing collections in progress for other MAPs that have requested the same performance measurement data or events.

3.2.2 UMA Messages

UMA messages provide the basis for transmitting existing notifications and data from UMA to a MAP. In addition, they are the default basis for transmitting requests between Data Services Layers on distributed nodes. The data in UMA messages is identified by *classes* and *subclasses*; this is defined in detail in Chapter 4.

Control Messages

UMA control messages include MAP requests to the UMA facility, UMA *condition* notifications from UMA to a MAP, and in distributed environments, request and acknowledgment messages between remote and local Data Services Layers.

Data Messages

Data Messages contain either *interval* data, *event* data or *configuration* data:

- *interval data* is requested by a MAP for capture at the end of a specified time interval. The data reported through the MLI is the difference in value of the requested metrics over the interval, or absolute counter values
- *event data* consists of notification messages to the MAP indicating that some predefined set of system events has occurred. The system events include UMA configuration (for example, the availability of metrics), system configuration (for example, hardware information — such as number / type of processors), and process-end summaries.

- *configuration data* contains data informing an MLI-based application of the data classes and subclasses available for each registered provider to the DCI.

Certain message subclasses have both interval and event forms. This permits the MAP to select whether data is to be reported at each interval end, at an event (for example, the termination of a process), or both.

Depending on the specified destination, a data message may be directed to the MAP itself, to UMADS (a common UMA data storage facility), or to a private file for later processing.

3.2.3 Screening and Filtering of Data

UMA provides two means by which the message traffic to a MAP (and possibly connected network traffic) can be reduced:

- establish threshold settings, thereby preventing the transmission of data messages unless the threshold conditions are satisfied (for example when the runqueue length reaches a particular value)
- adjust the granularity of the collected data (for example, by restricting reporting to a particular process or user id).

3.2.4 Constructed Workloads and Summarisation

The UMA MLI supports requesting of workload construction by permitting the labelling of workloads. These constructed workloads typically represent the result of a request for filtering and/or summarisation of workload metric subclasses. For example, one could request the selection of all commands starting with the letters ‘‘abc’’ and one could additionally request that a specific per-work unit metric subclass report its process metrics over the sum of all processes whose command names start with these same letters.

A constructed workload is assigned an identifier by the caller which can then be used to tag this workload for later reference.

A special constructed workload that is the *complement* of a specified workload is also available. The complement workload metrics are derived by subtracting the selected per-work-unit workload data values from the available global equivalents. For example, for reporting at the process level, if the selection criterion is ‘‘User Name: Albert’’, then the cpu utilization metric for the complement workload would consist of the global cpu utilization minus the usage for all processes running under the user name ‘‘Albert’’.

3.2.5 UMA Data Storage

UMA provides for the reading and writing of messages to and from conventional (private) files. In addition, UMA provides UMADS, a common facility for access and maintenance of historical data.

UMADS maintains individually accessible collections of data by host, but there is no requirement that data for a specific host be kept on that host. Instead, a systems administrator can arrange to have UMADS collections for any number of hosts stored on *performance data servers*.

Seamless Access

UMA provides *seamless access* between historical and recent (live) data. This means that a MAP may be receiving UMADS historical data until the time reaches the present, at which time UMA automatically switches its source to provide live data (see Figure 3-2).

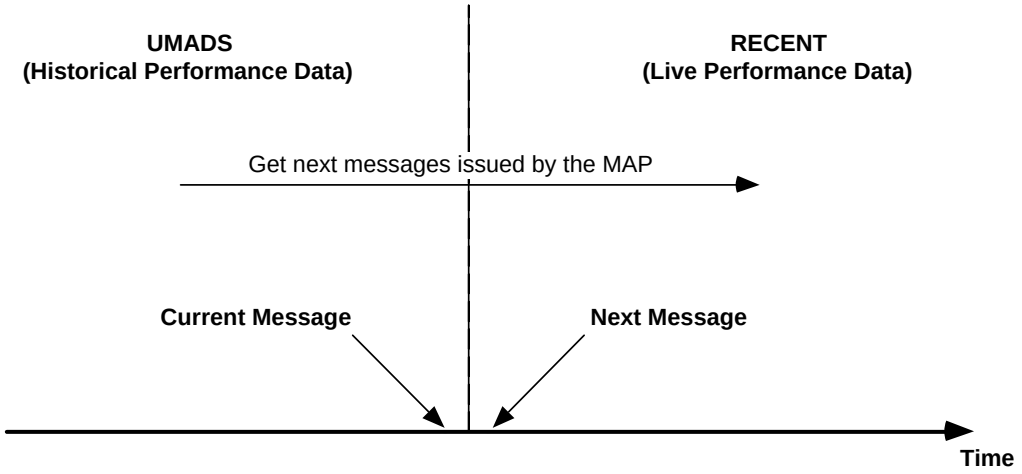


Figure 3-2 Seamless Switch - Historical to Recent Data

UMA also provides a *seek* mechanism, so that a MAP can navigate through time and can seamlessly access UMADS data from the present time (or the reverse). A seek from RECENT (current data) to UMADS (historical data) is illustrated in Figure 3-3.

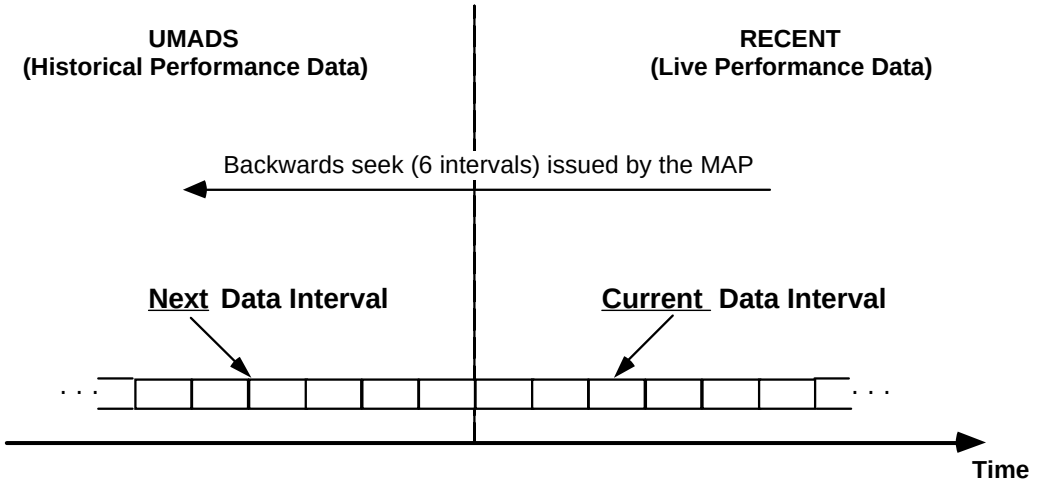


Figure 3-3 Backwards Seek - Recent to Historical Data

3.2.6 Data Capture Synchronisation

The UMA Data Services and Measurement Control Layers enable better synchronised data capture in two ways:

- first, the UMA Data Services can utilise global time synchronisation facilities, if they are available, to ensure that polled data collections on different platforms occur at the same time
- second, on each individual platform, the UMA Measurement Control Layer merges all measurement requests for polled data so that they may be requested at one time by a single process. This reduces the time skew of the data to the length of the collection time itself. UMA provides an optional additional check on the time skew at the UMA subclass collection level. If the collection time duration for the subclass is inordinately long, the capture can be re-attempted immediately.

Chapter 4

UMA Data Pool

The UMA Data Pool is the conceptual grouping of all performance data, and is represented at both the DCI and MLI. The representation at these two interfaces is different due to the different purposes of the interfaces but there is a direct mapping between the representations.

The UMA Data Pool groups data into *classes* and *subclasses*. Each data class can have several subclasses. The *class* identifies the major grouping (for example, memory, processor) and the *subclass* provides a specific grouping with class (virtual memory usage, block I/O counters). Within a *subclass* there is data (or metrics) which can be used to represent various kinds of data, including absolute counts, different counter values over an interval, and event data.

4.1 UMA Name Space

Figure 4-1 illustrates characteristics of the UMA name space.

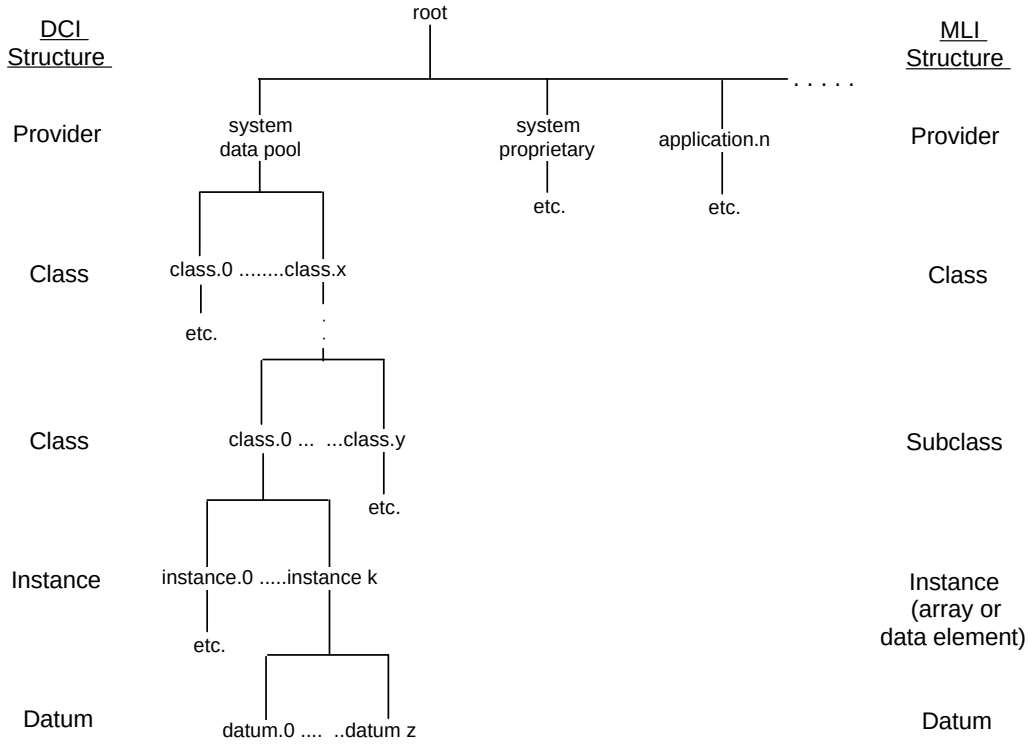


Figure 4-1 UMA Name Space Structure

First, there are the *levels of granularity* as seen by an MLI or DCI consumer. These levels descend through class, subclass, instances and individual data items. Instances are used to identify specific occurrences of data, for example, a process id or disk drive address. In addition, there are metric attributes that describe the data itself. These include data type, units, labels, and event attributes. An MLI application accesses these by requesting specific UMA Configuration

subclasses. A DCI consumer can request these through a set of API calls.

Another characteristic is that of a *provider dimension* (shown as the first level under the name space root). The UNIX kernel is one such provider, albeit one for which the class definitions have been already addressed in detail. Providers for other operating systems are not only possible but will be encouraged, as will providers for user application components such as DBMS or transaction management components. End-users may also wish to instrument their mission-critical applications and include this performance data in the UMA name space as well.

It should be noted that there are some differences between the MLI and DCI views of the name space. The DCI maintains a name space structure which allows multiple class and instance levels (instances and data may only occur at the lowest class level in order to avoid object naming ambiguities). The MLI consumer (that is, a MAP) sees data assembled in simple class/subclass structures presented in data messages. Instances of a metric are either uniquely designated by an identifier in the subclass[2] or by an array index (for example, system call number). The DCI name space cross reference number may be used to uniquely identify each metric in an MLI-based application (for example, for specifying a threshold variable). This number is a string of period-separated numeric characters representing class, subclass and position number in the provider's Data Pool[3].

2. For example, as described in the referenced DPD Specification **per work unit** subclasses, a process identifier is a UMAWorkInfo instance in the DCI and MLI namespaces.

3. See, for example the **xref** data names in the referenced DPD Specification.

4.2 Data Pool Data Segments

The Data Pool specification describes three conceptual data segments or groupings within a data subclass. These are:

- **basic**
  This is a segment of universally supplied data for the subclass as defined by the Data Pool. Every implementation of UMA is expected to supply this segment and its data.
- **enhanced**
  This is a segment of data whose structure and content are defined by the Data Pool, but this segment or some of its data may or may not be present in a particular implementation.
- **extension**
  This segment may be present. It is data specific to a vendor's hardware or software implementation.

The presence or absence of particular segments or data items are indicated by attributes in UMA configuration subclasses.

Chapter 5

Distributed UMA

The performance management of distributed environments implies that in general data can be collected on separate network-connected hosts, archived on any host or hosts an administrator chooses to, and can be accessed by sufficiently authorised MAPs executing anywhere in the network.

5.1 Extensible UMA Services and Configurations

The UMA Data Services Layer is, for each host, the hub of UMA communications between hosts. In a simple form, this situation is depicted in Figure 5-1.

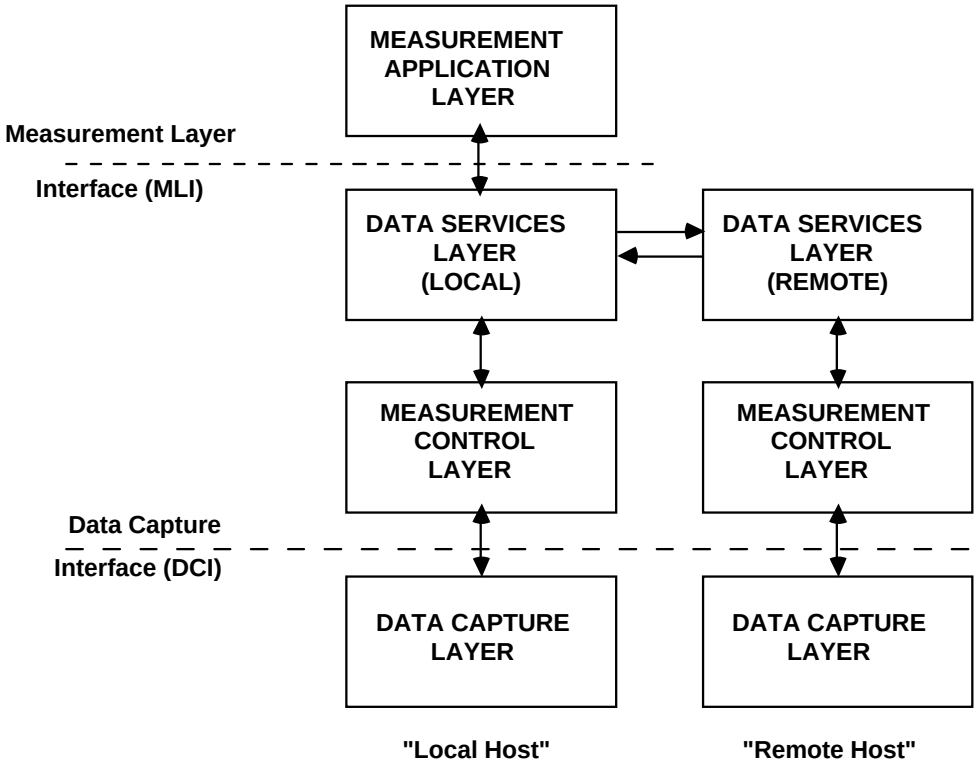


Figure 5-1 Fundamental Distributed UMA Configurations & Communications

Figure 5-1 shows a full UMA configuration on the *local host* (the host executing a MAP) communicating with a *remote host* that is, in this case, executing only the bottom three UMA Layers.

It is quite possible on the other hand to have the local host exist only to be a platform for control and display of performance data collected elsewhere. In this case, it will not be executing a Measurement Control Layer nor a Data Capture Layer, but instead, just a Data Services Layer and the MAP (together with its linked MLI). Furthermore, its Data Services Layer is much simplified, since it has no local data collections to coordinate or process, and it may not even be supporting a local UMADS. This situation is depicted below in Figure 5-2.

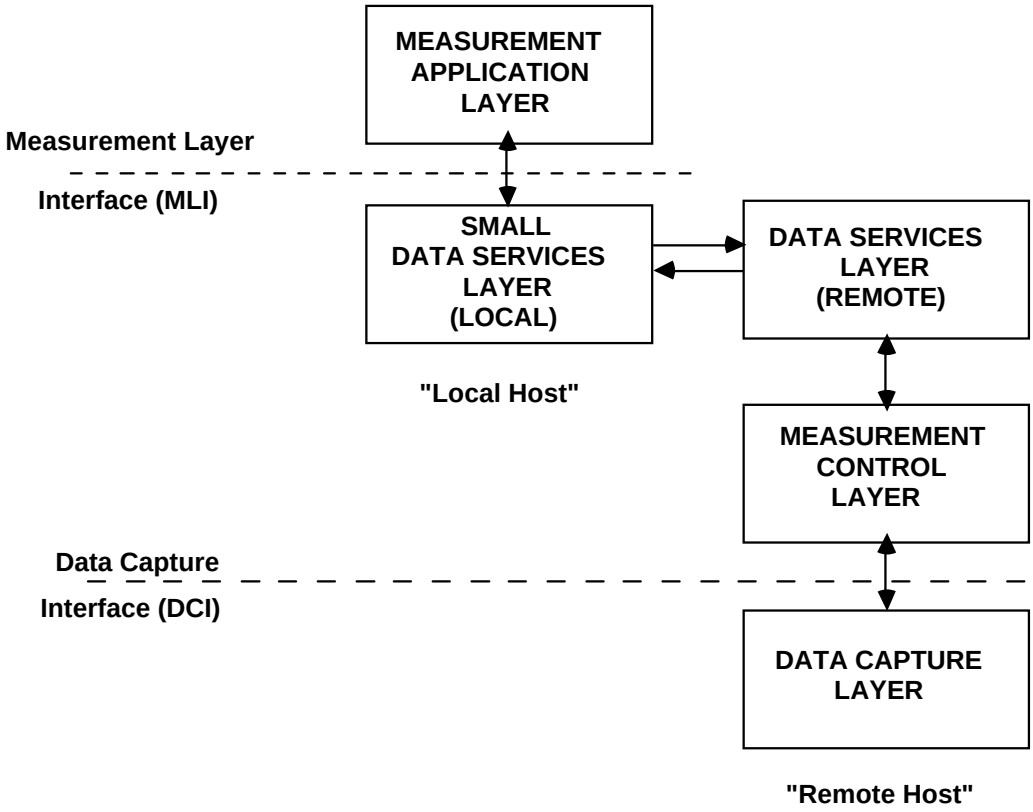


Figure 5-2 ‘‘Small’’ Data Services Layer on a Local Host

Here the local Data Services Layer (referred to as ‘‘small’’) is much simplified, merely acting as a forwarding service for local MLI requests to the remote host and for responses and data from the remote host back to the MAP through the MLI.

In practice, one would like to deploy instances of UMA components that provide only those services that may be required in a given case. For example, there is no need to (nor is it desirable to) deploy an instance of the Data Services Layer that implements UMADS reading and writing on a host that is exclusively an object for data collection.

A related step to deploying only those UMA services required in a given instance is to deploy only those *configuration* components required. For example, there is no need to deploy either local Measurement Control or Data Capture Layers on a host that is used only for the analysis and display of performance data from *other* hosts.

Figure 5-3 depicts a sampling of possible UMA service and configuration instances.

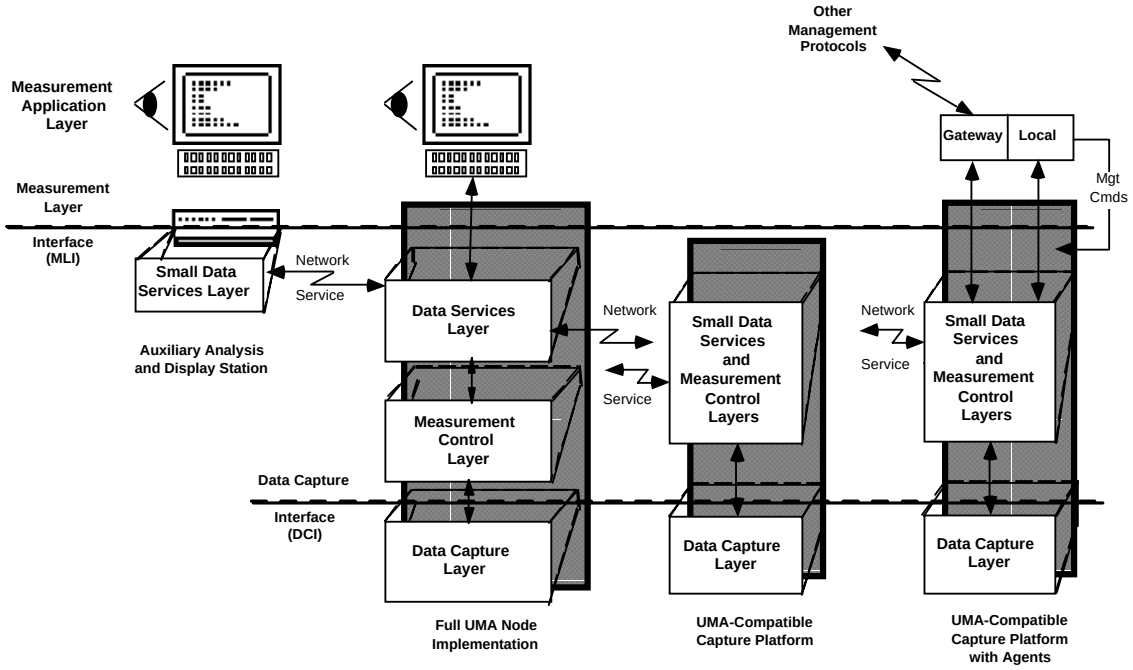


Figure 5-3 Flexible UMA Services and Configurations

Moving from left to right, we first have a host that is used exclusively for analysis and display of data - it implements only the MLI and a ‘‘small’’ Data Services Layer. Next, is a ‘‘full’’ configuration that has all of the layers of the UMA reference model and all of its services - this host is used both as an object of data collection, a repository of archival performance data (UMADS), and as a platform for analysis and display applications. The next two platforms are used only for the collection of live data (they have no read or write access to UMADS) and so implement only ‘‘small’’ Data Services Layers. The rightmost platform additionally supports two local measurement applications, one that monitors its performance and issues local management commands (not part of UMA) that make adjustments to promote better performance of certain critical applications, the other to export some of the performance data via other protocols (to an SNMP-based monitor, for example).

5.2 Interoperability

The Measurement Layer Interface supports the semantics of distributed data sources and destinations. The effect of this is that a Measurement Application Program (MAP) can request data from a number of sources throughout the distributed systems and these will be presented, at the MLI, in a co-ordinated manner. In addition, a MAP can specify both the source and destination of the data, thus enabling it to control the storage of measurement data on a *Performance Data Server*. The Data Services Layer is responsible for co-ordinating the distribution of data, which it does by interactions between such layers on each system that is involved in providing a UMA service.

The default protocol and interface by which Data Services Layers communicate is TCP/IP with sockets (using the registered ports).

UMA messages contain information on byte ordering, enabling them to be used between systems which have different conventions.

Chapter 6

Relationship of UMA to Other Technologies

This Chapter briefly consider the relationships that the UMA architecture and functionality have with other system management or measurement technologies in open system environments.

6.1 Relationship to Frameworks

In the context of distributed systems management frameworks, the UMA measurement model fits naturally into CORBA-based[4] environments. The Measurement Application Programs are clients using the services provided by the Data Services Layer and this gives a number of administrative and functional benefits provided by these frameworks. These include:

- the definition of services that provide discrete functionality enables individual systems to be configured to provide only those services that are required.

  Thus, for example, on one node that functions as a measurement data server, we might wish to configure a measurement server instance that includes the method for writing to a historical archive of performance data, but not do so at another node that functions only as a business application entity, that is, an object that provides performance metrics about itself.

- the ability to identify and authenticate a measurement application program and its invoker, and to authorise access to appropriate measurement data services (security).

  For example, some measurement applications and users might have authorisation to write to some specific database within a measurement data archive but not to others, or some users might have the authorisation to see performance data concerning some business application(s) but not others, etc.

- the ability to transparently operate measurement applications, server objects, and Data Capture Layer collector objects across locations in a network.

  This implies, for instance, that a measurement application at one network location (a manager system) may request and receive data from a measurement server or managed system at another network location without having to directly establish contact with the remote provider. The measurement model formulates such access as a peer-to-peer communications between objects in the Data Services Layer.

- providing a repository for well-defined interfaces.

  For the measurement model these would be the Measurement Layer Interface (MLI) and the Data Capture Interface (DCI).

4. Common Object Request Broker Architecture, an approach for supporting distributed object-oriented applications, formulated by the OMG.

Figure 6-1 illustrates a possible mapping of the UMA onto CORBA.

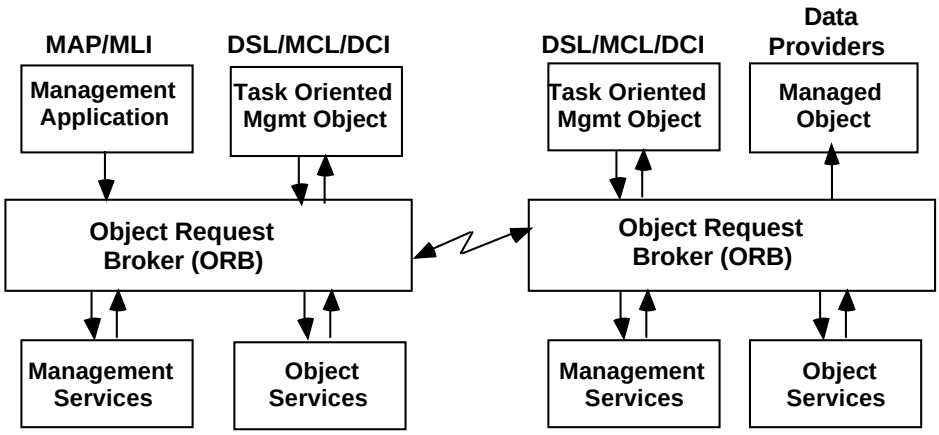


Figure 6-1 UMA-CORBA Relationship

6.2 SNMP and UMA

The Simple Network Management Protocol (SNMP) is based on an architectural model that consists of a number of network management stations and network elements. The management stations execute management applications which monitor and control the network elements. The network elements are devices such as hosts, gateways, terminal servers, which have management agents responsible for performing the network management functions requested by the network management stations. The SNMP protocol is used to communicate management information between the network management station and the agents in the network elements.

The Management Information Base (or MIB) is a virtual data store through which managed objects are accessed. Objects in the MIB are defined using Abstract Syntax Notation One (ASN.1) (which is how data objects are defined in the UMA DCI name space). Object types include integers, IP addresses and counters (non-negative integers).

Although the MIB has included some host level information, most of it is not of the fine granularity required for system performance management. The MIB focus has been primarily on network information, whereas the UMA Data Pool has focused extensively on the host with some network data included. In this sense, the MIB and the UMA Data Pool are mostly complementary.

SNMP as a protocol implies that the management station and the agent in a network element communicate via the network, thus data collection implies network load. In UMA, this is not necessarily true; one can have a local DCI or MLI consumer on a host that collects, analyzes and acts on data collected for that host, with no resulting network load. Furthermore, where data is to be exported from a host, UMA intrinsically provides mechanisms for data selection and threshold filtering, thus reducing network traffic to only the most essential.

Architecturally, UMA and SNMP can complement each other in the following two ways:

- using SNMP protocol, a provider can acquire and supply extended network information to UMA from the MIB to UMA consumers.

  This is done in the most straightforward manner by registering the name space contents for MIB- provided data under an SNMP provider position in the DCI name space.

- UMA may act as a data- or event-supplying agent to a management application using SNMP protocols.

  Depending on the complexity of the task, either a simple DCI consumer application or a MAP can play this agent role. Using a MAP built on the MLI, it is possible to produce either composite numerical information or SNMP traps based on data simultaneously acquired from a set of distributed platforms.

It is expected that a number of organisations will investigate and prototype implementations based on either or both of these approaches to the SNMP/UMA relationship.

6.3 DMI and UMA

The Desktop Management Interface (DMI) specification from the Desktop Management Task Force (DMTF) describes a general API for obtaining management information from system components. As a single system interface, it cannot really be compared with UMA which through its MLI and Data Services Layer gathers data from distributed nodes and archives it in a location-transparent fashion. Furthermore, DMI has no provision for maintenance of or access to historical data.

On cursory examination, the DMI may appear to have similar function to the DCI component of UMA. Both have the goal of providing component instrumentation to management applications. Both are single system interfaces. Both provide for polled data acquisition as well as component events.

However, there are substantial differences. These arise from the DCI having been designed specifically for performance management. This has resulted in a richer name space and navigation mechanism (consider, for example, the unrestricted name space depth and the wildcarding capability supported by the DCI); the ability to obtain large amounts of data spanning different system components with relatively small queries and to do so at a high rate — the DMI instead mandates the formulation of requests into small units making the requesting of large amounts of data require the issuing of many requests. Furthermore, the UMA interfaces and name space can support the notion of trace by use of high-speed events, for which substantial volumes of data can be directed to a file at a very high rate.

As for events (sometimes called *indications* in the DMI specification), the DMI requires that each management application that may wish to be notified of even just a single event receive all events, which requires management applications to filter for those events they wish to examine. In server environments, there will be many events occurring (say, for example, process terminations) and it would be most inefficient to notify all management applications of each and every one. The UMA interfaces allow consumers to specify event sensitivity so that the DCI and the Data Services Layer will selectively notify their consumers of only those events that were selected.

Considering next the DMI MIF file that describes a component's *manageable characteristics*, this file must be loaded (or reloaded) in its entirety into the MIF database each time there is a change to the availability of a *characteristic*. This approach to registration is far more suited to relatively static data such as represented by system configuration and availability information than it is to rapidly changing metric availability. Performance management requires (and the DCI provides) more dynamic registration right down to the metric and instance levels as might, for instance, be represented by the appearance and termination of a process thread.

It is therefore expected that systems will support both DMI and UMA. UMA (through the DCI) will provide access to the typically very dynamic and high volume performance data, while DMI will make configuration and vital product data available (for example). Where appropriate, DCI providers could make use of the DMI to obtain the configuration information required to support the UMA Data Pool. This would ensure a single source for this type of data.

Glossary

CORBA
Common Object Request Broker Architecture, from the OMG. See the referenced **CORBA** specification.

DBMS
DataBase Management System

DCI
Data Capture Interface

DMI
Desktop Management Interface, from the DMTF.

DMTF
Desktop Management Task Force: a special-interest group developing a Desktop Management Interface for systems management of distributed desktop computer resources.

DPD
Data Pool Definition

/dev/kmem
Historically, this is the primary interface used by UNIX system performance measurement utilities for extracting data from the kernel.

interoperability
The ability of distributed systems - software and hardware on multiple machines and from multiple vendors - to communicate effectively.

kernel
The name commonly used to refer to the central processing software which all interfaces (to memory, device drivers and input/output subsystems) use in a computer system.

MAP
Measurement Application Program

MIB
Management Information Base

MIS
Management Information System

MLI
Measurement Layer Interface

OMG
Object Management Group

ORB
Object Request Broker: provides the means by which clients make and receive requests and responses.

PMWG
Performance Management Working Group (PMWG). This group was initially sponsored by UNIX Systems Laboratories (USL) and is now sponsored by Computer Measurement Group (CMG).

POSIX
A group of operating system interface and environment standards, developed under the aegis of the Institute of Electrical and Electronic Engineers (IEEE), and based on the UNIX operating system documentation, which are designed to support application portability at the source level.

ps
A UNIX environment command to display the status of current processes.

sadc
data collection process associated with sar.

sar
system activity reporter - a tool in the UNIX environment.

sysinfo
A UNIX environment structure that contains system information.

SNMP
Simple Network Management Protocol: a protocol for managing networks which use the Internet Protocol Suite (IPS).

UMA
Universal Measurement Architecture

UMADS
UMA Data Storage

Index

CAE Specification

Part 2:

UMA Measurement Layer Interface (MLI)

The Open Group

Contents

List of Figures

List of Tables

Preface

This Document

This document is a CAE Specification. It provides a set of specifications for the functional characteristics of the Universal Measurement Architecture's (UMA) Measurement Layer Interface (MLI), and describes the underlying semantics and the function calls that implement these characteristics. The MLI also defines a format for headers appended to data captured by the low-level Data Capture Interface (DCI). (These headers are part of the control and status messages sent between UMA and the applications it serves.)

There are two associated UMA specifications which, along with the MLI specification, define the UMA system:

- UMA Data Capture Interface (DCI) specification (see Part 3 of this specification).

  This is the interface between the data capture layer and the measurement control layer of the UMA architecture.

- UMA Data Pool Definitions (see Part 4 of this specification).

  The data pool defines a set of performance metrics which may be accessed by the two UMA interfaces.

The UMA Guide (see Part 1 of this specification). reviews the issues surrounding performance measurement in Open Systems, describes the general UMA architecture, and discusses user considerations in adopting the UMA.

Audience

The target audience for this document is both system designers, who need to implement this interface, and performance professionals, who need to understand how this interface can be used.

Structure

- **Chapter 1, Introduction** — defines the objectives of this specification and defines terms and acronyms.
- **Chapter 2, UMA Reference Model** — defines the layers and interfaces of the UMA Reference Model.
- **Chapter 3, UMA Sessions** — describes the characteristics of UMA sessions, introduces MLI calls, and gives an overview of the messages used for communication between Measurement Application Programs (MAPs) and UMA.
- **Chapter 4, Data Collection, Reporting, and Recording** — introduces the concepts used by UMA in data collection and reporting to a MAP, and discusses the UMA Data Storage (UMADS) data recording facility.
- **Chapter 5, MLI Application Programming Interface** — defines the MLI calls used to manage UMA sessions, and to specify and access reported data.
- **Chapter 6, UMA Message and Header Formats** — defines control and data segments formats.

- **Chapter 7, Distributed UMA** — defines specifications that allow multiple UMA implementations to interoperate in a distributed environment.
- **Chapter 8, The UMA Configuration Class** — defines the messages that describe the parameters pertainin to a specific UMA implementation and data providers.
- **Appendix A, C Language Header Files** — presents the **<mli.h>** and **<uma.h>** header files.
- **Appendix B, Future Directions** — describes a generalized command interface for the MLI that is currently undergoing development.

Acknowledgements

This specification was developed by the Performance Management Working Group. The PMWG was originally part of UNIX International, and is now part of the Computer Measurement Group.

X/Open gratefully acknowledges the work of the PMWG in the development of this specification and in the review process for this publication.

Major contributors to the Measurement Layer Interface specification include:

Peter Benoit	Digital Equipment Corp.	Ansgar Erlenkoetter	Tandem Computers, Inc.
Paul Farr	Aim Technology	Lewis T. Flynn	Amdahl Corporation
Tony Gaseor	AT&T Bell Laboratories	Javad Habibi	Amdahl Corporation
Jim Richard	Amdahl Corporation	Leon Traister†	Amdahl Corporation
Neal Wyse	Sequent Computer Systems, Inc.	Seung Yoo	Amdahl Corporation

Participants who have made contributions to the process of developing these specifications are listed below along with their corporate affiliation at the time of their contribution. Our sincere apologies to anyone whom we may have missed.

Sara Abraham	Amdahl Corporation		
Subhash Agrawal	BGS Systems	Barrie Archer	ICL
Robert Berry	IBM Corporation	Tom Beretvas	IBM Corporation
Wolfgang Blau	Tandem Computers, Inc.	Jim Busse	NCR Corporation
David Butchart	Digital Equipment Corp.	David Chadwick	Performance Awareness Corp.
Ram Chelluri	AT&T Global Information Solutions	Danny Chen	AT&T Bell Laboratories
Niels Christiansen	IBM Corporation	Paul Curtis	Hitachi computer Products (America), Inc.
Paul Douglas	Digital Equipment Corp.	Janice Dumont	AT&T Bell Laboratories
Jerome Feder	UNIX System Laboratories	Mark Feldman	Sequent Computer Systems, Inc.
Thierry Fevrier	Hewlett-Packard	Ken Gartner	Hitachi Computer Products (America), Inc.
Joseph Glenski	Cray Research, Inc.	Dave Glover	Hewlett-Packard
Jay Goldberg	UNIX System Laboratories	William Hidden	Open Software Foundation
Liz Hookway	NCR Corporation	John Howell	Amdahl Corporation
Ken Huffman	Hewlett-Packard	Mario Jauvin	Bell Northern Research
Chester John	IBM Corporation	Sue John	IBM Corporation
Rebecca Koskela	Cray Research, Inc.	Bill Laurune	Digital Equipment Corp.
Ted Lehr	IBM Corporation	Greg Mansfield	Instrumental
Shane McCarron	UNIX International	Michael Meissner	Open Software Foundation
Marge Momberger	IBM Corporation	Bernice Moy	Open Software Foundation
Henry Newman	Instrumental	Jee-Fung Pang	Digital Equipment Corp.
James Pitcairn-Hill	Open Software Foundation	David Potter	Open Systems Performance, Inc.
Melur K. Raghuraman	Digital Equipment Corp.	O. T. Satyanarayanan	Amdahl Corporation
Yefim Somin	BGS Systems	Steve Sonnenberg	Landmark Systems
Douglas R. Souders	UNIX System Laboratories	Jim Van Sciver	Open Software Foundation
Jaap Vermeulen	Sequent Computer Systems, Inc.	Michael Wallulis	Digital Equipment Corp.
Ping Wang	Open Software Foundation	Steve Whitney	Boeing Computer Services
Elizabeth Williams	Super Computer Research	Willie Williams	Open Software Foundation

† Editor

Chapter 1

Introduction

1.1 Purpose

This document is one of a family of documents that comprise the Universal Measurement Architecture (UMA), which define interfaces and data formats for Performance Measurement. UMA was originally defined by the Performance Management Working Group (PMWG) and subsequently adopted by The Open Group.

This document is a specification for the functional characteristics of the Measurement Layer Interface (MLI), as defined in the UMA Reference Model in Chapter 2. The document describes the underlying semantics and the function calls that implement them. It also defines a format for headers appended to data as captured by the low-level Data Capture Interface (DCI). (These headers are part of the control and status messages sent between UMA and the applications it serves.)

The UMA is defined in the following documents:

- Universal Measurement Architecture Guide (see reference **UMA**). This document provides an overview of the UMA.
- UMA Measurement Layer Interface Specification (this document).
- UMA Data Capture Interface Specification (see reference **DCI**). This document defines a standard programming interface for capturing system and application provided data.
- UMA Data Pool Definitions (see reference **DPD**). This document defines a performance data pool for the analysis and management of computer systems, and an organisation to facilitate the collection and use of such data.

1.2 Scope of the Universal Measurement Architecture

The scope of the Universal Measurement Architecture is:

- to provide a set of common measurement control and data delivery services for (client) performance applications
- to provide seamless access to current and historical measurement data
- to insulate the operating system kernel from performance display and analysis applications by means of a common application programming interface (API)
- to maintain portability of user tools to any systems that implement the architecture (again through the common API)
- to provide specific mechanisms for data capture that implement metric registration functions from distinct data providers
- to provide a mechanism for control of the instrumentation that coordinates the capture of kernel and non-kernel data sources
- to support access to distributed performance functions and data from remote nodes.

The UMA architecture specifies two interfaces:

- the MLI — a high-level application programming interface for specification and reporting of formatted measurement data, (this document)

  and

- the DCI — a low-level application programming interface for acquisition of raw kernel, trace, event, and subsystem data, (see reference **DCI**).

In addition, the architecture provides services for distributing data to multiple applications, for maintaining historical data, and for synchronising the capture of metrics from different sources.

Performance and capacity management of operating systems have been considered *internal* to the operating system and as such differ from one operating system to another operating system, and from one implementation to another implementation. Most operating systems have, as a matter of necessity, performance analysis modules, narrowly targeted at the type of hardware, software and networking facilities implemented within the system.

Most operating systems provide ad-hoc developed or tailored performance metrics. Some of these tools are developed as internal support tools for benchmarking or on demand of performance analysts and capacity planners. These tools are generally also confined to one machine only and can not be interrogated remotely.

The new era of networking and interoperability views performance management and capacity planning from the user's perspective. Multiple machines and operating systems can be involved in the interaction with the user. This approach requires capture and presentation of performance metrics to be clearly defined and portable between platforms and operating systems.

1.3 Definitions, Acronyms and Abbreviations

Terms, acronyms and abbreviations used in this specification are defined in the Glossary.

1.4 Conformance

A conformant implementation must support all of normative requirements in this MLI specification, that is, as specified in chapters 1 - 8) except in the following respects:

1. Support of unsolicited events:

   MLI support of unsolicited events is optional. When a measurement application requests delivery of unsolicited events from an MLI implementation which does not support this feature, status and reason codes will be returned from the call indicating the condition.

2. Support of metric, instance tag and work unit description by metadata:

   MLI support of metric, instance tag and work unit description by metadata is optional. This information, when available, is presented in the ''Subclass Attributes'' subclass of the class ''UMA Configuration'' solicited with the *umaRequestConfig*( ) MLI call. When not supported, an attempt to retrieve this subclass will return status and reason codes indicating that the subclass is not implemented.

3. Support of dynamic data:

   MLI support of dynamic data is optional. The availability of dynamic classes and subclasses is indicated in the subclass ''Implementation'' of class ''UMA Configuration'' by the setting of the bit flag UMA_DYNAMIC for each class and subclass. Support of dynamic data implies the MLI support of metric metadata.

4. Support of protocol section contents in control messages:

   The only required usage of the control UDU header protocol section is that the *umaCreate*( ) and *umaReconnect*( ) logical message protocols and their acknowledgements include the protocol section field describing the sending platform's wordsize (*mh_wdsize*). The other fields may be used to support private protocols between MLI service layers. The presence or absence of the additional protocol fields has no impact on the use of the MLI API.

5. MLI use of the DCI:

   Use of the DCI by the MLI for data acquisition is optional. However, if the DCI is used as a source of data, the user should refer to the DCI conformance requirements.

6. Enhanced security services:

   These are specified external to the MLI.

7. WorkInfo granularities:

   While MLI implementation of the Workload definition mechanism (using UMAWorkInfo) is mandatory, the availability of specific WorkInfo granularities is defined by the data provider.

Chapter 2

UMA Reference Model

The UMA reference model defines four layers and two interfaces, as shown in Figure 2-1.

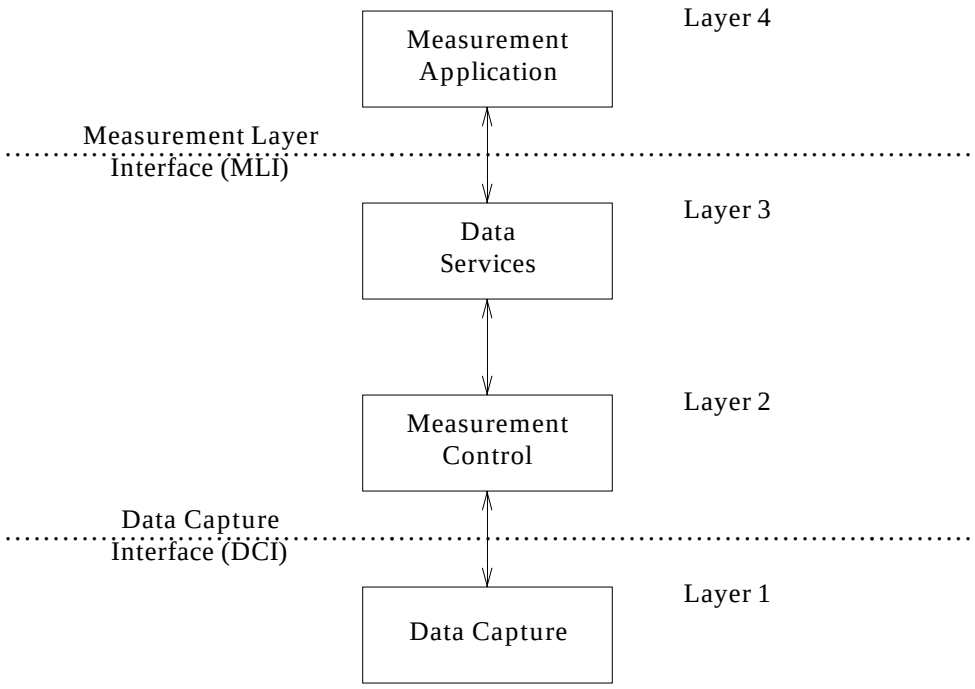


Figure 2-1 UMA Layers and Interfaces

2.1 Functional Layers

In order, from data capture to application program, the functional layers are:

- the Data Capture Layer (DCL), which collects the raw data
- the Measurement Control Layer (MCL), which manages the data collection
- the Data Services Layer (DSL), which distributes data to archive, to networked (that is, distributed) components, to files, and directly to MAPs, and which handles measurement requests, data transformation, and filtering
- the Measurement Application Layer (MAL), which consists of the various MAPs requesting data collection and providing service capabilities for technical support of management goals.

2.2 Interfaces

UMA establishes two programmatic interfaces which concern data provider developers and measurement applications:

- a Data Capture Interface (DCI) to request data through *collection orders* to UMA-compatible data providers. The DCI lies between the DCL and the MCL, and is used to register, provide, and acquire data. (See reference **DCI**).
- a Measurement Layer Interface (MLI), an application programming interface for measurement applications to communicate with the UMA facility. The MLI lies between the DSL and the MAL, and is used by MAPs to specify collection and reporting attributes.

Note: The interface between the DSL and the MCL is not formally specified. These two layers, though functionally distinct, may be combined in some implementations.

Figure 2-2 illustrates the two programmatic interfaces (DCI and MLI) in relation to the functionality of UMA components.

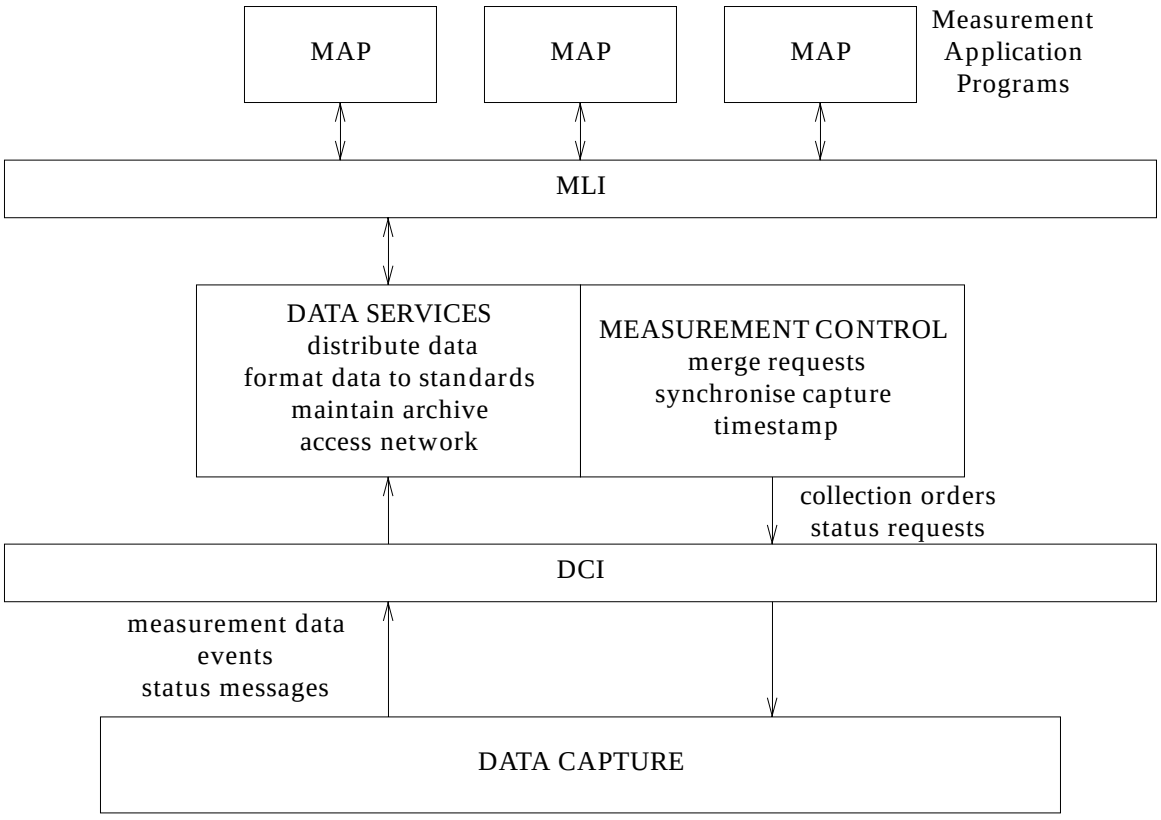


Figure 2-2 Components and Interfaces of UMA

2.3 UMA Characteristics

The three layers, DCL, MCL, and DSL, and the two interfaces, DCI and MLI, comprise the basic UMA facility. When this document describes UMA or refers to UMA characteristics, it is describing the combined capabilities of these layers and interfaces. The MLI isolates a MAP from the implementation details of the rest of UMA. Interactions between a MAP and UMA are carried out through the MLI.

The UMA reference model provides several advantages over the older data collection and display programs such as *sar*. The structure of the DCL, and the DCI, permits the extension of data collection to new devices or services without affecting existing programs.

Additionally, the architecture of the DCL allows the data from multiple providers to be collected by a single layer and this in turn improves the synchronisation of the collected data. For example, since the data reported by *sar* and various other statistical commands use different data collection routines, there is an unpredictable time delay between datapoints collected by one program and datapoints collected by the other. With UMA, the collection is more closely synchronised.

By having UMA manage the collection and distribution of data, multiple application programs can use the same collected data. This results in savings of memory, disk space, and system overhead. The MLI and the DSL allow transparent communication across networks, therefore a MAP running on one system can manage and display data from another system across the network.

Chapter 3

UMA Sessions

This Chapter describes:

- Session characteristics
- UMA messages.

3.1 Session Characteristics

The UMA facility is that part of UMA that provides data and session services to the MAP (through the MLI).

A MAP accesses the services of the UMA facility by first establishing a session and then issuing the appropriate MLI calls. A session is a channel of communication over which the MLI sends messages to the UMA facility to set up and control the reporting of data and to receive status and data messages.

Each session has an associated data source, a data destination, and *property* flags that specify certain fixed characteristics of the session; these constitute the session *context*. The creation and specification of the session and its context are described in the description of the *umaCreate*( ) call.

A session also has certain changeable *attributes*. These include a session start time, a session end time, a reporting priority, a reporting interval size, and search control attributes. The start time session attribute determines when reporting is to begin. There is an internally maintained session current time that indicates the time of the data interval currently being reported. Nominally, the session current time is initially set to the session start time (subject to constraints imposed by the settings of the certain session search attributes, as will be later described).

3.1.1 MLI Calls

The MLI calls are the means by which a MAP communicates with the UMA facility to establish and access UMA sessions. All MLI calls return a status code indicating the general outcome of the call. Further detail regarding failed calls can be obtained by invoking the *umaGetReason*( ) call.

A number of UMA-defined data types are used for specifying UMA objects such as classes, subclasses, message flags, etc. The type definitions, their values and valid operator definitions are incorporated into a MAP by including header file **<uma.h>** (see Appendix A on page 107).

3.1.2 MLI Security

When enhanced security provisions are required beyond what is the operating system default, the MLI and UMA Data Services Layer (DSL) encapsulate the security-related exchanges with other entities. This means that the MLI library does not give the MAP explicit access to security-related keys or *tickets*. Instead, the MLI library itself acquires an authentication key and sends it to the Data Services Layer (DSL), where it is authenticated. Subsequently issued tickets and encryption/decryption keys are retained by the MLI library for use until expiration. Thus, the MAP proper and its user do not necessarily know that encryption, or what levels of security are in effect. The only visible result of a security interaction between local and remote UMA Data Services Layers (DSLs) is a possible UMR_PERMISSION reason code returned from session establishment, and data solicitation MLI calls if access is denied.

To ensure interoperability between various implementations that support such enchanced security, the Generic Security Service API (GSS-API) is to be used as the default security API by the UMA DSL for session establishment, and, where desired by administrators, for ensuring per-message integrity and confidentiality. These enhanced security services are optional for both implementers and administrators. This means that implementations are possible that do not support enhanced security and that administrators may select their use or non-use when available by means of installation or configuration options. GSS-API potentially supports a variety of underlying security mechanisms. (See reference **GSS**).

3.2 Basics of UMA Messages

Once a session is established, a MAP communicates with the UMA facility by means of MLI calls. These result in the MLI formatting messages that are delivered to the UMA facility. The MLI provides a set of functions and utilities that provide for the creation, sending, receiving and handling of these messages. Information from UMA to the MAP is also communicated (through the MLI) via messages. The MAP accesses these messages with the MLI function *umaGetMsg*().

Messages divide into two groups: *data* and *control*. Both data and control messages contain data items identified by the class and subclass.

All messages consist of a header and one or more segments. In addition, *data* messages have an extension header between the header and segments. The header provides the basic information necessary to start extracting information from the message, including the time stamp of the message, the duration (if an interval message), and offsets into the message body for the segments. Additionally, the class and subclass of a message are indicated.

The message, its header and segments, include ASN.1/BER encoded tags and length descriptors (used for parsing the message). (ASN.1/BER encoding is used in data communication between open systems.) The contents and detailed structure of the segments (body) depends on the class and subclass.

3.2.1 Data Messages

Data messages contain either *interval* or *event* data:

- Interval data is that which is specifically scheduled for capture at each expiration of a specified time interval. The data reported for that interval is the increment of the item values over the interval or, if requested, the value of the metric accumulated from a startup through that interval.
- Event data consists of notification messages to the MAP indicating that some predefined set of events has occurred. The currently defined events include UMA configuration data, system configuration data, and process-end summaries. Other classes and subclasses for event data may be defined. Delivery of event data to the measurement application may be requested as in-band (that is, in timestamp sequence) or out-of-band (ahead of timestamp sequence).

The type of data is indicated by a flag in the header.

Data segments for both interval and event data can contain the following:

- *Basic* data, which each UMA implementation must supply (segment must be present)
- *Optional* data, which is generally available but not mandatory (segment is optional)
- *Extension* data, which is vendor defined (segment is optional).

Each of the data segments begins with an ASN.1/BER tag-length prefix. The location of each segment is specified in the extension header as an offset from a message global start position.

The *UMA configuration* data describes which classes, subclasses, and data fields are implemented. The *system configuration* data describes the hardware and software configuration of the system. It includes system parameters statically defined at boot time and hardware status changes such as disk mounts and unmounts.

The class identifies the major grouping (memory, processor, and so on) and subclass provides a specific grouping within class (virtual memory usage, block I/O counters, and so on). The definitions and grouping of data items by these classes and subclasses are documented in the document, Data Pool Definition (see reference **DPD**).

A MAP initiates ongoing reporting of data for a class and subclass via a *umaStart*() call. Depending on the specified destination, data may be directed to the MAP itself, to UMADS, or to a file. A MAP may use the *umaRequestConfig*() call to request one time event data, that is, UMA or system configuration data.

Because of the flexibility and extensibility of UMA messages, a MAP may need to trace through several fields and data structures in the header and extension header to extract the requested segment.

3.2.2 Control Messages

Control messages include MAP requests to the UMA facility, UMA condition notifications from UMA to a MAP, and, in distributed environments, request and acknowledgement messages between remote and local Data Services Layers.

Chapter 4

Data Collection, Reporting and Recording

4.1 UMA Collection and Reporting

UMA distinguishes between the *reporting* of data to a MAP and the *collection* of data. A MAP requests reporting from a specified source to a destination (which may be the MAP itself) in the form of UMA messages. The UMA facility acts on behalf of a MAP to perform the actual data collection through the Data Capture Interface (DCI). This may mean initiating the collection or making use of an existing collection in progress for other MAPs that have requested the same measurement classes and subclasses. Data from the DCI will have an appropriate header appended to it and may undergo certain transformations and filtering before it is reported to one or more MAPs.

The UMA Data Capture Layer (DCL) is responsible for data collection to UMA-controlled structures from hardware registers, system counters and tables, driver counters and tables, etc. The collection of data has occurred when the data is in a structure controlled by UMA, that is, a structure that cannot be modified until a UMA component permits it.

Most subclasses defined in the Data Pool Definition (see reference **DPD**) specify *interval* data reporting, that is reporting of a *difference* in data values over the duration of the interval or a data value at the end of the interval. Reporting of absolute (undifferenced) data values at each interval can be chosen by specifying a flag UMA_WORKLOAD_ABSOLUTE in the *umaStart*( ) MLI call. In addition, there are subclasses associated with events and subclasses that may have both interval and event forms.

4.1.1 Capture Synchronisation and Data Coherency

Specifying capture synchronisation means that UMA Measurement Control will attempt to schedule data capture from different DCL data providers or instances of DCLs so that they occur at very near the same time. Capture coherency means that, as defined by implementation criteria, the collection of a UMA subclass will be as near an atomic operation as possible. This means that a collection may be rejected and retried if the atomicity criteria are not met.

4.1.2 Data Reporting - Intervals

In UMA, data is reported at intervals that a MAP specifies. Data collections take place at *regular* interval times, that is, at predictable wall clock times as measured from 12 midnight. See the *umaSetAttr*( ) specification for a definition of regular intervals.

If the session start time or the reporting request (*umaStart*( )) occurs at a time *between* regular collection times, the first interval reported for the session may be for a shorter duration than that specified. All subsequent collections will be of the correct duration and at regular times.

A MAP may request exemption from regular interval collections for a session by specifying the session property UMA_NOTREGULAR. Session properties will be discussed in detail in a subsequent section.

If historical data has been requested (from UMADS, or a file), data may not be available at the specified interval. In this case, the UMA facility supplies data at the archive interval up to the point where either recent data or current data are available at the requested interval. The UMA message header specifies the applicable interval for any supplied data. The MAP can prevent

data being retrieved from UMADS by setting the RECENT_ONLY session attribute to TRUE.

4.1.3 Data Reporting - Events

Event data in UMA consists of notification messages to the data consumer that some predefined set of system events specified by the class and subclass have occurred. The requester defines a UMA start/end time window for which these notifications are to be received. The data sources and destinations are specified as for interval UMA data.

By specifying certain event-related flags in the *umaStart*( ) MLI call, delivery of event data to the measurement application may be requested as either in-band (that is, in timestamp sequence) or out-of-band (ahead of timestamp sequence). This will be further discussed in the description for the *umaStart*( ) call.

Certain UMA subclasses defined in the Data Pool Definitions (see reference **DPD**) have both event and interval forms. For example, all *Per Work Unit* data pool subclasses admit both forms. This permits the MAP to select whether data is to be reported at each interval end, at an event, say, the termination of the process or at a process change in group ID, or both.

The UMA Configuration subclass is reported either as solicited event using the *umaRequestConfig( )* MLI call or as an unsolicited event specified in a *umaStart( )* MLI call.

4.1.4 Trace Data

Trace data in UMA is treated as high-volume event data. The requester specifies a UMA start/end time window during which he wishes to have the trace data collected. The class and subclass define the specific trace(s) activated. Because of the potentially high data volumes, trace data should normally be directed to a file.

4.1.5 Screening and Filtering Data

UMA provides two data interpretation capabilities that permit the reduction of message traffic to the MAP:

- Screening of subclasses
- Work Unit Data Filtering.

4.1.5.1 Subclass Screening

Assuming that the provider can measure and deliver data at the required level of granularity, it is possible to restrict reporting of subclasses that have met threshold criteria. Interval data transmissions to a MAP are screened, based on a set of variable values that are compared to session-specific threshold settings.

A MAP session will be able to establish and change threshold settings which will inhibit transmission of data to the session for those intervals where the associated variables are within the threshold ranges. The MAP session uses the *umaSetThreshold*( ) function to establish or change thresholds.

4.1.5.2 Work Unit Data Filtering

It is possible to select subclass data by various *Per Work Unit* criteria, for example:

- Process ID
- Command Name

- Command Name in conjunction with User ID
- Process ID in conjunction with Transaction ID.

This is discussed further in the description of the *umaStart*( ) call.

4.1.6 Constructed Workloads and Summarisation

The UMA MLI supports requesting of workload construction by permitting the labelling of workloads specified in the *umaStart*( ) call. These constructed workloads typically represent the result of a request for filtering and/or summarisation of workload metric subclasses. The summarisation is over the elementary workloads specified by selection criteria in the *umaStart*( ) call. Thus, for example, one could request the selection of all commands starting with the letters ''abc'' and one could additionally request that a specific per-work unit metric subclass report its process metrics over the sum of all processes whose command names start with these same letters. A constructed workload is assigned an identifier by the caller which can then be used to tag this workload for later reference. The workload tag is implemented in UMA data messages as a special case of an instance tag.

A special constructed workload that is the *complement* of a specified workload is also available. The complement workload metrics are derived by subtracting the selected per-work-unit workload data values from the available global equivalents. For example, for reporting at the process level, if the selection criterion is <User Name: Albert>, then the cpu utilization metric for the complement workload would consist of the global cpu utilization minus the usage for all processes running under the user name ''Albert''.

4.1.7 Data Buffering and Delivery

Normally, the Data Services Layer (DSL) will buffer data messages associated with intervals that are destined for a MAP. That is, they are not sent until an event happens that requires the buffer to be sent and then they are sent in a block. This triggering event might be the end of a reporting interval, the arrival of the last message requested, a buffer getting filled, etc. The deblocking of such messages is generally handled by the MLI.

Data messages associated with events having the *out-of-band* attribute are sent to the MLI as soon as they arrive.

4.1.8 Recent Data Facility

UMA provides a *Recent Data* facility to hold a limited number of the most recent data messages for MAP-requested interval sizes. The number of intervals and the granularities potentially maintained by this facility would be specified by the systems administrator in a configuration file for each UMA instance.

4.2 UMA Data Recording

4.2.1 UMADS (UMA Data Storage)

The UMADS facility consists of an interface and a set of storage mechanisms for access by the DSL. Data in UMADS may be structured or recorded in implementation-specific ways. The only requirements are that the data be capable of being read by the DSL, that it support positioning in time via the *umaSeek( )* MLI call, and that the DSL can format its contents to the message standards. UMADS, therefore, functions as a time-indexed, non-volatile cache.

For specification of UMADS access, either the *source* and *destination* parameters of the *umaCreate*( ) function may be set to a string of the form "UMADS[(*dbid*)]"[1] , which denotes a specific UMADS. In particular, the *dbid* may be used to designate any of a number of UMADS *areas*, for example *hourly*, *daily*, etc. Access to these areas may be controlled to permit either *public* or *private* read/write through the use of an administrative UMADS authorisation file.

Requests to UMADS can be originated by a MAP making historical reporting references. These historical references may occur in one of two ways. First, a MAP may explicitly constrain access to UMADS by specifying it as the *source*. Second, a historical reference may result from a positioning to a time that is before any contained in the Recent Data facility.

Data messages are sent to the MAP from UMADS or the Recent Data in exactly the same form, thus providing a seamless link of historical and current data. Interval sizes from the historical data sources may, however, be either larger or not integer divisors of the requested interval. In this case, the data provided from these sources may be at a different interval size (the UMA data message header will indicate the size). If this is not satisfactory for some applications, the user may consider use of specially collected data saved in a *private* UMADS area.

The UMA MLI supports location transparency for UMADS data. This means that:

1. UMADS data for a specific host (*sysid*) does not have to be located on that host

   and

2. The measurement application does not need to know where host-specific UMADS data is located.

UMA MLI calls, therefore, can refer to the object system through its symbolic name and let UMA locate the historical data for it - this is the *presumed location*. Alternatively, a caller may specify a location as a source or destination for UMADS data for a given session, if security policy permits it.

No special MLI calls are required for a MAP to access UMADS; however an administrator may need to perform certain support and maintenance that are specific to UMA. Examples include *dbid* initialisation, copy-in or copy-out by class and subclass, setting retention periods by class/subclass, and so on.

1. In subsequent text, "UMADS[(*dbid*)]" will frequently be abbreviated to "UMADS".

4.2.2 Compatible Granularity

For any host, several UMADS areas can be established, each having a different interval size. This allows users to select UMADS data with an interval size that is compatible with various *live* interval sizes (for example, the UMADS interval size could be a multiple of the live interval.)

Interval sizes can be mixed within a single UMADS area. This should be done, however, so that it is consistent and easily handled by MAPs. For example:

- a single short interval size for peak hours, and a longer interval size for off-peak hours
- short intervals for a few minutes at the beginning of each hour.

4.2.3 Private Files

UMA provides for the reading and writing of session messages to and from conventional *flat* files.

Specification of private files, for reading or for writing is via the *source* and *destination* parameters of the *umaCreate*( ) call. The details are discussed in the description of *umaCreate*( ). It is important to note that UMA does not support seamless switching between private files and Recent Data. (Seamless Switching between UMADS and Recent Data is supported.) When a private file is the *source* in the *umaCreate*( ) call, it is the only source of data for the session.

Chapter 5

The MLI Application Programming Interface

This section discusses MLI calls and their associated parameters.

5.1 Overview of MLI Calls

Since different MAPs have different purposes, there is no single sequence of system calls. The following table illustrates the relationship between a typical sequence of MLI calls.

Note that the calls *umaCreate( ), umaReconnect( )* and *umaGetMsg( )* are the only MLI calls that exhibit blocking behavior until they either complete or time out.

umaCreate( )	Open a session, specify source and destination and return a session id used in later calls to identify the session.
umaRequestConfig( )	Obtain system and UMA configuration information (for example, what classes and subclasses are available, and what processors and devices are connected or enabled).
umaSetAttr( )	Specify or change session attributes.
umaSetThreshold( )	Specify threshold values for filtering reported subclasses.
umaStart( )	Specify which data to start reporting. UMA starts any necessary data collection.
umaRelease( )	Release held *starts*. By default when a session is created, the data reporting is held until a *umaRelease*( ) call.
umaGetMsg( )	Return a data or control message for the session.
umaGetReason( )	Except for *umaCreate*( ) and *umaReconnect*( ), which return reasons through their own parameter lists, MLI retains status and reason codes for the most recent MLI call. The *umaGetReason*( ) call returns these codes.
umaSeek( )	The MLI maintains a notion of the time of the start of the next data interval. This call changes that time. The requested time may be in the past, present or future.
umaStop( )	Tell UMA to stop reporting specified data. Data collection will continue if other sessions collect the same data.
umaRelease)	Release the held *stops*.
umaClose( )	Shuts down a session. Data collection may continue. Refer to the UMA_NOTERM property in *umaCreate*( ).
umaReconnect( )	Reestablishes the control connection to a previous non-terminating session that had been closed.

Table 5-1 A Common MLI Call Sequence

5.2 MLI Call Parameters

The parameters for MLI calls are as follows. The UMA or C data type is indicated in brackets, for example [int].

attrpairs In *umaSetAttr*( ) and *umaGetAttr*( ), a set of name-value pairs terminated by NULL, specifying (changeable) session attributes. The names are specified by quoted strings. Currently, these names include:

ETIME Specifies the ending time of data reporting, that is, times greater than ETIME are not reported. [UMATimeSec]

INTERVAL Specifies the session data reporting interval in seconds. [UMATimeSec]

HISTORY_ONLY
Indicates that only *historical* data is requested either from UMADS or from a specified input file. [UMABoolean] Setting this attribute TRUE may cause the session's current time pointer to be repositioned to an earlier time and may make times later than those in UMADS or the specified file unreachable (unless reset). Setting this attribute to TRUE will automatically set the attribute RECENT_ONLY to FALSE.

The default value for *HISTORY_ONLY* depends on the session *source* as described in the manual page of the *umaSetAttr()* MLI call.

PARTIAL When *regular* interval collection times are in effect, setting *PARTIAL* to TRUE indicates that a requested change of interval size can take place at the next regular collection time for the shorter of the new and old intervals [UMABoolean]. Otherwise, the change would take place at the time that the old interval and new interval start times occur together. Therefore, this specification may result in either a single truncated (partial) old or new interval at the time the change is made, depending on whether the new interval is shorter or longer than the old. The default value of *PARTIAL* is FALSE.

Note that this specification has no meaning when *regular* interval collection times are not in effect.

RECENT_ONLY
Indicates that only data from the *Recent Data Facility* is requested from UMA, that is, private file or UMADS data are excluded. [UMABoolean] It is an error to attempt to set this attribute to TRUE when either a private file or UMADS have been designated as the *source*. Setting this attribute to TRUE may cause the session's current time pointer to be repositioned and it may make times prior to those in the *Recent Data Facility* unreachable (unless reset). Setting this attribute to TRUE will automatically set the attribute HISTORY_ONLY to FALSE. The default value for *RECENT_ONLY* depends on the session *source* as described in the manual page of the *umaSetAttr( )* MLI call.

PRIO A non-negative integer specifying the session attribute: relative priority of data message delivery to the MAP. [UMAPrio] Increasing values designate decreasing priority. The default value of *PRIO* is 3. The highest priority is for *PRIO* equal to zero.

STIME — Specifies the starting time of data collection. Times equal to or greater than *STIME* are reported subject to their being less than or equal to *ETIME*. [UMATimeSec]

channel_flag — In *umaGetMsg*( ), specifies whether *umaGetMsg*( ) should return only in-band messages (UMA_IN_BAND_ONLY), only out-of-band messages (UMA_OUT_OF_BAND_ONLY) or either in-band or out-of-band messages (UMA_ANY_BAND). When only out-of-band messages are to be delivered, in-band messages are held until a call is made to *umaGetMsg*( ) with the *channel_flag* set to either UMA_ANY_BAND or UMA_IN_BAND_ONLY. Similarly, when only in-band messages are requested, out-of-band messages are held until a call is made with the *channel_flag* set to either UMA_ANY_BAND or UMA_OUT_OF_BAND_ONLY.

dclass — specifies a message class. [UMAClass]

dsubcls — specifies a message subclass. [UMASubClass]

dbid — In *umaCreate*( ), specifies the data base id in UMADS. [<string>].

destination — In *umaCreate()* specifies the MAP itself, an output file or UMADS as a destination for reporting. See the *umaCreate*( ) call manual page for a complete description of values. [<string>]

flushflags — In *umaStop*( ), contains the flags that indicate which previously issued *umaStart*( ) requests are cancelled. [UMAFlushFlags] The possible values of *flushflags* are:

UMA_HELD — Cancel only those requests that have been started but not released.

UMA_RELEASED — Cancel only those start requests that have been released.

UMA_ALLSTARTED — Cancel both held and released requests.

location — In *umaCreate*( ), qualifies *source* and *destination* with a specification similar to that defined for *sysid*, that is, a qualified node name (e.g. srv.accting.acme.com), or as an IP address in string form with periods (e.g. 129.210.1.1). [<string>]

position — In *umaSeek*( ), a relative position measured in intervals from a specified time. [UMAInt4]

provider — In *umaRequestConfig*( ), *umaSetThreshold*( ), *umaStart*( ) and *umaStop*( ), designates the registered provider for which collection classes and subclasses are specified. [UMAProvider] The defined constant <*OS*>_DATAPOOL is used to designate the classes and subclasses defined by an operating system-specific datapool[2].

reason — A reason code for a UMA error indication in *umaCreate*( ), *umaReconnect*( ), and *umaGetReason*( ). [UMAReasonCode]

segflags — In *umaStart*( ), these are flags that specify the segment types to be reported. [UMASegFlags] The segment types defined in the Data Pool Definitions (see reference **DPD**) currently include:

1. UMA_ASEG, indicating that all available segment types are to be reported.

2. For example, UNIX_DATAPOOL, is the operating system provider for Unix-based systems.

2. UMA_BSEG, indicating universally supplied data.
3. UMA_OSEG, indicating optional data defined in the standard.
4. UMA_ESEG, indicating vendor-specific data.

sessid Specifies a session identifier. [UMASessId]

sess_req In *umaGetMsg*( ) specifies a requested session. [UMASessId]

sess_ret A returned session identifier in *umaGetMsg*( ). [UMASessId]

source In *umaCreate*( ), specifies the source(s) of data for reporting to be current collections from UMA data providers, an input file, or UMADS. [<string>]

sprops In *umaCreate*( ), contains the session property flags. [UMAProp] Session property flags are set, reset, cleared, and tested with the macro operators UMA_SET, UMA_RESET, UMA_CLEAR, and UMA_ISSET, respectively. All property flags are initially in a reset state. Currently, these properties include:

1. A property UMA_COHERENT for an MLI requester session to specify that it requires that subclass coherency criteria be applied to collections.
2. A session property flag UMA_NOTERM to allow all reporting associated with the session to continue if the session is closed. This allows a collection and reporting activity to continue in a background mode.
3. A session property flag UMA_NOTREGULAR for a session to request exemption from common, regularised interval data reporting.
4. A property flag UMA_SYNCH for a requester session to specify that it requires time synchronisation between different UMA data providers.

subclass In *umaStart*( ), specifies the UMA subclass to be reported. [UMASubClass]

sysid In *umaCreate*( ) and *umaReconnect*( ), specifies the node from which data is to be collected. May be specified as a qualified node name (for example srv.payroll.acme.com) or as an IP address in string form with periods (for example 129.210.1.1). [<string>]

timeout In *umaGetMsg*( ), specifies a time in seconds and microseconds to wait for a message to be available before returning. In *umaCreate*( ) and *umaReconnect*( ), specifies a time in seconds and microseconds to wait for communication and establishing or re-establishing a session. The time used is the sum of *timeout.tv_sec* seconds plus *timeout.tv_usec* microseconds. [UMATimeVal]

tstamp In *umaSeek*( ), defines a time stamp in seconds, for UMA data interval searches. [UMATimeStamp]

whence In umaSeek, specifies whether the seek is relative to UMA_CTIME, UMA_LTIME, UMA_STIME or to a timestamp.

UMAWorkInfo

In *umaRequestConfig*( ), the composite of a set of *UMAWorkDescr* structures returned in the metadata class ‘‘Configuration’’, subclass ‘‘WorkInfo Attributes’’. *UMAWorkInfo* defines the work unit tags that a provider may use to link a specific instance of metric subclass collection with a logical workload. *UMAWorkInfo* tags are global for a provider. See the *umaRequestConfig*( ) MLI call for details.

UMAWorkDefn

In *umaStart*( ) and *umaStop*( ), a structure used to specify reporting filters by use of

UMAWorkInfo items, instances, and granularity of reporting. See the *umaStart*( ) MLI call for details.

5.3 MLI Macro Operators

The definitions and implementations of MLI flags (for example session property, message header indicators, etc.) are generally hidden from the MAP developer. Any necessary manipulations and operations on these types by MAPs (and by UMA components themselves) are to be handled by a set of provided macro operators:

```
UMA_SET(FIELD, FLAG)          /* set FLAG in FIELD        */
UMA_RESET(FIELD, FLAG)        /* reset FLAG in FIELD      */
UMA_ISSET(FIELD, FLAG)        /* test IN FIELD            */
UMA_CLEAR(FIELD)              /* clear all flags in FIELD */
UMA_SESSIONEXISTS(sessid)     /* test session existence   */
```

Note that a flag field must first be cleared (using UMA_CLEAR) before it is first used.

The macro operator definitions are established by including the **<uma.h>** header file.

5.4 UMA MLI Call Descriptions

This section contains syntactical and functional descriptions of UMA MLI session calls. A MAP developer includes the header file **<uma.h>** to establish the parameter type declarations.

Each call is described by a model definition that includes the returned type, the function name, and its formal parameters. This is followed by a specification of the formal parameter types as expected by the MLI and a description of the interface behaviour. A list of UMA status and reason codes applicable to the call concludes the call description (definitions of these codes are included with the UMA type definitions for *UMAStatusCode* and *UMAReasonCode*, respectively).

NAME

umaClose - Shut down one or all active sessions of a MAP.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaClose(
    UMASessId       sessid          /* in  */
    UMAReasonCode   *reason         /* out */
);
```

DESCRIPTION

The function *umaClose*( ) shuts down the session denoted by *sessid*. Data will continue to be collected and reported to the session's destination (until the session's end time is reached) if the UMA_NOTERM property flag was set when the session was created (see *umaCreate*( ) on page 27.

DIAGNOSTICS

The returned value of *umaClose*( ) indicates the general outcome (status code). Status and reason codes that apply to *umaClose*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	—
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA facility
UMS_SESSID	—	Invalid sessid specified
UMS_SESSION	UMR_RESOURCE	Resource not available

NAME

umaCreate - Establish a new session for a MAP.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaCreate(
    char            *destination,      /* in  */
    char            *sysid,            /* in  */
    char            *source,           /* in  */
    UMAProp         sprops,            /* in  */
    UMASessId       *sessid,           /* out */
    UMATimeVal      *timeout           /* in  */
    UMAReasonCode   *reason            /* out */
);
```

[*Editor's note for the CAE Draft*:

The principal change to this call is to allow the explicit specification of import and export private files in canonical message format.

DESCRIPTION

The source designator may be any of:

```
"RECENT"                                /* recent data only            */
"[<location>:]UMADS[(dbid)]"            /* UMADS data only             */
"[<location>:]UMADS[(dbid)]  + RECENT"  /* UMADS and recent data       */
"UMA_SFILE"                             /* environment-specified file  */
"[<location>:]<path>[(import)]"         /* private file                */
```

The destination designator may be any of:

```
"[<location>:]UMADS[(dbid)]"            /* UMADS data only               */
"UMA_DFILE"                             /* environment-specified file    */
"[<location>:]<path>[(export)]"         /* private file                  */
UMA_MAP                                 /* defined macro MAP destination */
```

The function *umaCreate*( ) requests the UMA facility to open a session with the specified *source* (source of the performance data) and *destination* (where the data is to be reported). The session properties are indicated by *sprops*. The returned parameter *sessid* is subsequently used by the calling MAP to communicate with UMA over this session.

If a connection to *sysid* and a session are not established within *timeout.tv_sec* seconds plus *timeout.tv_usec* microseconds, *umaCreate*( ) will return with a status code of UMS_COMM and reason code UMR_TIMEOUT.

<location> identifies a specific UMADS or file physical location. It overrides the presumed location established by UMA administration. If unspecified, the default *source* location is the presumed location for *sysid*. The default *destination* location is that in effect for *source*. <location> is specified either as a qualified node name (e.g. server.accting.acme.com), or as an IP address in string form with periods (e.g. 129.210.1.1).

The data reported is for the node designated by *sysid*. The string "UMA_LOCAL" designates the node that is executing the Data Services Layer (DSL) accessed by this MAP.

Note that *sysid* is the system for which the data is reported; it is not necessarily the system on which data resides.

To specify reporting of current collections from UMA data providers and UMADS in a seamless fashion, a string of the form ‘‘UMADS + RECENT’’ is used for *source*. (In this context ‘‘UMADS’’ is an abbreviation for ‘‘[<location>:]UMADS[(*dbid*)]’’.) To restrict the source to the current collections (i.e. to the contents of the Recent Data Facility), the string ‘‘RECENT’’ is used. To restrict the source to UMADS only, a string of the form ‘‘UMADS’’ is used.

In addition, data may be read from a private file as follows:

1. If *source* is the string "UMA_SFILE", the MAP’s environment is searched for a string "UMA_SFILE=[<location>:]<path>[(import)]";

   otherwise

2. *source* contains a direct specification of the form [<location>:]<path>[(import)].

If *destination* is "UMA_MAP" then the data collected will be queued for reading by the issuing session. If the destination is not "UMA_MAP" or "UMADS", the data will be written to the file specified by *destination* as follows:

1. if *destination* is the string "UMA_DFILE", the MAP’s environment is searched for a string of the form "UMA_DFILE=[<location>:]<path>[(export)]";

   otherwise

2. *destination* contains a direct specification of the form [<location>:]<path>[(export)].

Note that when the destination is "UMADS" and the source is not "UMADS", the only source permitted is the Recent Data Facility; that is, *source* must be "RECENT" or it may be "UMADS + RECENT" with the RECENT_ONLY attribute set to TRUE.

If the source and destination are two different UMADS areas (i.e. for different *sysid*s or different *dbid*s), the following additional characteristics apply to interval sizes:

1. If an interval in the source UMADS area is equal to or greater than the specified session interval, the destination interval actually written will be a copy of the source UMADS interval;

   otherwise

2. intervals from the source UMADS area are aggregated until the time that is a multiple of the requested destination interval is reached or exceeded.

In addition, if the destination UMADS area already exists, the first interval actually written will be one that strictly follows the last one existing in the destination. That is, data will only be appended and not overwritten or merged.

Data from a file source may only be written to a not-previously-existing UMADS area. Data written to a private file destination can only be written to a not-previously-existing file, that is, file destinations cannot be appended.

Private files are normally written so that they may be read by the UMA implementation that created the file. If, however, it is desired to write such a file in UMA message format so that it may be read by any UMA implementation, the ‘‘(export)’’ suffix is added to the file path following the file name as in /home/uma/daily(export). As a *source*, the ‘‘(import)’’ suffix is added instead. Export private files are written in Canonical A message format and include all necessary metadata.

Also note that sources or destinations that are specified by a simple file name will be searched for in or written to the MAP’s current working directory.

If permissions allow the user and the MAP to create a session, and if the session can be successfully established, the argument sessid is set to the session identifier and the call returns a status of UMS_SUCCESS.

SESSION PROPERTIES

A session may be opened by a MAP purely for the setup of data reporting directed to a named destination (that is, a destination other than the MAP itself). In this case, once the setup is complete, the invoking MAP may no longer be needed and the session may then be *shut down*. Reporting to the specified destination can be made to continue by setting the session property UMA_NOTERM. It is not valid to specify UMA_NOTERM when the destination is a MAP. If, after shutting down a session with the UMA_NOTERM property, it is necessary to resume control of the reporting, a *umaReconnect*( ) call may be issued for the same destination named in the session that originated the reporting.

UMA will not normally attempt to schedule data collections from different UMA Data Capture Layer instances (that is, Data Capture Layers on different hosts) so that they occur at the same point in time, that is they will not necessarily be synchronised). Setting the session property UMA_SYNCH indicates that for this session, global collector synchronisation is requested.

UMA permits the application of collection atomicity criteria to ensure that the data collected within UMA subclasses are coherent. (If the criteria are not met, UMA will attempt to repeat the collection). Setting the session property UMA_COHERENT indicates that for this session, data coherency criteria are to be applied if they exist. Coherency criteria are specified as a time window value for allowable skew and a retry count in the configuration file for the UMA instance.

The session property UMA_NOTREGULAR can be set to exempt a session from the enforced use of *regular* intervals. The maximum interval is equivalent to 24 hours in any case. See the discussion for the MLI function call *umaSetAttr*( ) for more details about intervals.

DIAGNOSTICS

The returned values of *umaCreate*( ) indicate the outcome of the call (status code) and the reason for a failed status (reason code). If the call is not successful, the returned value of *sessid* will not be defined. Status and reason codes that apply to *umaCreate*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_COMM	UMR_CONNECT	Attempt to connect to UMA failed
	UMR_NETWORK	Network error
	UMR_RECEIVE	Communications error when receiving messages from UMA
	UMR_RESOURCE	Insufficient system resources
	UMR_SEND	Communications error when sending messages to UMA
	UMR_SYSERR	System error while communicating with UMA
	UMR_TIMEOUT	Timeout while attempting to connect to UMA
UMS_DEST	UMR_ACTIVE	This *destination* is currently the target of another session
	UMR_CONFLICT	Conflict between source and destination
	UMR_UNKNOWN	Specified destination unknown
UMS_PROPERTY	UMR_CONFLICT	Conflict between property and destination
	UMR_INVALID	Property invalid
UMS_PROTOCOL	UMR_HEADER	Control message header not recognised by UMA Data Services
	UMR_MESSAGE	Control message content not recognised by UMA Data Services
UMS_SESSION	UMR_MAX	Maximum resources exceeded for this MAP
	UMR_PERMISSION	Access denied
	UMR_RESOURCE	Resource not available
	UMR_SYSERR	System error while establishing session
	UMR_UMADS	UMADS error while establishing session
UMS_SOURCE	UMR_UNKNOWN	Specified source unidentified

NAME

umaGetAttr - Obtain a session's attributes.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaGetAttr(
    UMASessId    sessid,        /* in     */
                 ...            /* in/out */
);
```

DESCRIPTION

umaGetAttr( ) returns the session attributes for the session denoted by *sessid*.

The *sessid* parameter is followed by a null-terminated variable length list of name variable pairs (*attrpairs*), each pair consisting of a quoted attribute name followed by a comma and the attribute variable. Attributes for the names specified in this list are returned to their corresponding variables. See the description of the *umaSetAttr* call and Section 5.2 for details on syntax and available session attributes. The variable length argument list **must** be terminated by a NULL argument.

EXAMPLE

This example assumes that there is an active session with sessid *sessid1*.

```
#include <uma.h>

UMAStatusCode status;
UMASessId sessid1;
UMATimeSec stime;
UMATimeSec intval;

status=umaGetAttr(sessid1, "STIME", &stime, "INTERVAL",
       &intval, (char*)NULL);
```

DIAGNOSTICS

The returned value of *umaGetAttr*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaGetAttr*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_ATTR	UMR_INVALID	An invalid attribute name specified
UMS_EOS	—	End of session encountered
UMS_SESSID	—	Invalid session specified

NAME

umaGetMsg - Return a pointer to the next available data or control message.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaGetMsg(
    UMASessId        sess_req,       /* in  */
    UMAChannelFlags  channel_flag,   /* in  */
    UMASessId        *sess_ret,      /* out */
    char             **msg_ptr,      /* out */
    UMATimeVal       *timeout        /* in  */
);
```

DESCRIPTION

The *umaGetMsg*( ) call is used to access data and control messages for a session. It sets the pointer *msg_ptr* to the next available message for the session indicated by *sess_ret*. The message will either be a data message or a control status message. The setting of *channel_flag* determines whether messages are to be selected from the in-band queue only (UMA_IN_BAND_ONLY), the out-of-band queue only (UMA_OUT_OF_BAND_ONLY), or from any band (UMA_ANY_BAND).

If there are no queued messages for this session, *umaGetMsg*( ) will wait at most *timeout.tv_sec* seconds plus *timeout.tv_usec* microseconds for a message to appear. If the numeric value of both fields is zero, the return is immediate whether there is a message queued or not (if a message is queued, it will be returned). If the pointer argument *timeout* is the null pointer, the call blocks until either a message has been queued, or until an interrupt signal (SIGINT) is received by the caller, in which case no message is returned and the status code is set to the defined value UMS_SIGNAL and the reason code to UMR_INTR. In any event, an interrupt signal will always result in an immediate return with this status and reason code.

If no message is queued and no error has occurred (within the time specified by *timeout*), the function returns the defined status value UMS_NOMSG. In this case, the function *umaGetReason*( ) will return the reason code UMR_TIMEOUT.

sess_req identifies the session requested. If the defined value UMA_ANYSESSION is specified for *sess_req*, then, if a message is returned, it will be from the highest priority session having messages queued. If the call fails or is interrupted, the reason code obtained via the function *umaGetReason*( ) for any existing session will indicate the condition.

In any case, *sess_ret* will be set to the *sessid* for the session message actually returned. If no message is returned, *sess_ret* is set to UMA_NULLSESSION.

When a session end time is reached, *umaGetMsg*( ) will return an end of session condition message (UMA_EOS). A session whose destination is a file or UMADS will be automatically closed when this end time is reached.

When end-of-file is detected on the source for a session whose source is UMADS or a file, *uma_GetMsg*( ) will return an end-of-file condition message (UMA_EOF). A session whose destination is a file or UMADS will be automatically closed when this end-of-file condition is encountered.

If any of the required session attributes *STIME*, *ETIME*, or *INTERVAL* have not been set, *umaGetMsg*( ) will return a NULL message pointer (*msg_ptr*). In this case, the returned status code will be set to UMS_ATTR with reason code UMR_INCOMPLETE.

DIAGNOSTICS

The returned value of *umaGetMsg*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaGetMsg*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_ATTR	UMR_INCOMPLETE	Required attribute not specified
UMS_COMM	UMR_RECEIVE	Communications error when receiving messages from UMA
	UMR_SEND	Communications error when sending messages to UMA
	UMR_SYSERR	System error while communicating with UMA
UMS_NOMSG	UMR_TIMEOUT	Timeout occurred, no message is returned
UMS_SESSID	—	Invalid sessid specified
UMS_SESSION	UMR_RESOURCE	Insufficient system resources
UMS_SIGNAL	UMR_SYSERR	System error while setting signal handler
	UMR_INTR	Interrupt signal (SIGINT) caught
UMS_TIME	UMR_INVALID	Invalid timeout value specified

NAME

umaGetReason - Return the reason and status codes pertaining to an established session.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaGetReason(
    UMASessId       sessid,         /* in  */
    UMAStatusCode   *status,        /* out */
    UMAReasonCode   *reason         /* out */
);
```

DESCRIPTION

The function *umaGetReason*( ) returns a reason code (along with the status code for an established session (denoted by *sessid*). The reason code returned by *umaGetReason*( ) is for the most recent preceding MLI call for that session.

NOTE

umaCreate( ) and *umaReconnect*( ) functions return reasons through their parameter lists.

DIAGNOSTICS

Status and reason codes that apply to *umaGetReason*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_SESSID	—	Invalid sessid

NAME

umaReconnect - Re-establish the connection to a previously closed session.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaReconnect(
    char            *destination,       /* in  */
    char            *sysid,             /* in  */
    UMASessId       *sessid,            /* out */
    UMATimeVal      *timeout            /* in  */
    UMAReasonCode   *reason             /* out */
);
```

DESCRIPTION

umaReconnect( ) requests the re-establishment of control between the calling MAP and the specified destination on the specified *sysid*. The specified destination must be either "UMADS" or a private file and it must be currently active, that is, being reported to. The session that originally specified the reporting must have been created with the UMA_NOTERM property, and must no longer exist at the time of the *umaReconnect*( ) call. The new session, denoted by *sessid*, will have the same attributes and properties as the session previously active for this same destination.

If a connection to *sysid* and a session are not established within *timeout.tv_sec* seconds plus *timeout.tv_usec* microseconds, *umaReconnect*( ) will return with a status code UMS_COMM and reason code UMR_TIMEOUT.

The destination designator may be any of:

```
"[<location>:]UMADS[(dbid)]"    /* UMADS data only            */
"UMA_DFILE"                     /* environment-specified file */
"[<location>:]<path>"           /* private file               */
```

<location> identifies a specific UMADS or file location. It overrides the presumed location established by UMA administration.

The semantics of *sysid* in *umaReconnect*( ) are the same as for the *umaCreate*( ) call.

DIAGNOSTICS

The returned value of *umaReconnect*( ) indicates the general outcome (status code). The supplementary reason code for a failed status is returned as an argument to the function. Status and reason codes that apply to *umaReconnect*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_COMM	UMR_CONNECT	Attempt to connect to UMA failed
	UMR_NETWORK	Network error
	UMR_RECEIVE	Communications error when receiving messages from UMA
	UMR_RESOURCE	Insufficient system resources
	UMR_SEND	Communications error when sending messages to UMA
	UMR_SYSERR	System error while communicating with UMA
	UMR_TIMEOUT	Timeout while attempting to connect to UMA
UMS_DEST	UMR_ACTIVE	This destination is currently the target of another session
	UMR_INVALID	Specified destination invalid
UMS_PROTOCOL	UMR_HEADER	Control message header not recognised by UMA Data Services
	UMR_MESSAGE	Control message content not recognised by UMA Data Services
UMS_SESSION	UMR_MAX	Maximum resources exceeded for this MAP
	UMR_PERMISSION	Access denied
	UMR_RESOURCE	Resource not available
	UMR_SYSERR	System error while establishing the session
	UMR_UMADS	UMADS error while establishing the session

NAME

umaRelease - coordinate and start or stop reporting the requested classes and subclasses or activate threshold settings.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaRelease(
    UMASessId     sessid       /* in */
);
```

DESCRIPTION

umaRelease( ) informs the UMA facility to execute any assembled requests from previous *umaStart*( ), *umaStop*( ) and *umaSetThreshold*( ) calls for the indicated session(s). The parameter *sessid* may be assigned the defined symbol UMA_ALLSESSIONS to indicate that any such assembled requests MAP should be released and executed.

NOTE

umaRelease( ) will take effect only when all the required session attributes *STIME*, *ETIME*, and *INTERVAL* have been set.

DIAGNOSTICS

The returned value of *umaRelease*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaRelease*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_COMM	UMR_SEND	Communications error while sending messages to UMA (one or more sessions)
UMS_SESSID	—	Invalid session specified
UMS_SESSION	UMR_RESOURCE	Insufficient resource (one or more sessions)

NAME

umaRequestConfig - request the reporting of system and UMA configuration information to a MAP.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaRequestConfig(
    UMASessId       sessid,             /* in */
    UMAProvider     provider,           /* in */
    UMAClass        dclass,             /* in */
    UMASubClass     dsubcls             /* in */
);
```

DESCRIPTION

The function *umaRequestConfig*( ) requests the reporting of configuration data as specified by *provider*, *dclass* and *dsubcls* for the session specified by *sessid*. The defined symbols UMA_ALLCLASSES and UMA_ALLSUBCLASSES may be used for *dclass* or *dsubcls* respectively, to indicate that all classes and subclasses are to be selected.

STRUCTURES AND METADATA

The following terminology will be used in this discussion:

Instantiated Subclasses

MLI subclasses are equivalent to DCI classes that contain metric items. Such DCI subclasses will be designated "instantiated subclasses". Instantiated subclasses may not contain other classes.

Built-in Classes and Subclasses

The MLI supports static classes and subclasses that are defined to user applications by the inclusion of header files of predefined structures for the subclasses. Messages in these built-in classes and subclasses are available the "Canonical C" form of UMA UDUs and do not require metadata for interpretation. Metadata may, however, be optionally supplied in the form of "Class Attributes" and "Subclass Attributes" subclass messages in the "UMA Configuration Class"

Dynamic Classes and Subclasses

Dynamic classes and subclasses are those that a provider explicitly registers, typically with the UMA DCI. These classes and subclasses may come and go during the course of a provider instantiation and require that metadata provided by the "Subclass Attribute" subclass messages in the "UMA Configuration Class" be used to interpret them. Messages in dynamic classes and subclasses are made available in the "Canonical A" form of MLI UDUs. Built-in classes may not have dynamic subclasses.

The subclass ‘‘Implementation’’ of class ‘‘UMA Configuration’’ contains a class-type array. In addition to implementation status, this array contains two bit flags, UMA_BUILTIN and UMA_DYNAMIC. UMA_BUILTIN will be set if the class is available as a built-in class, UMA_DYNAMIC will be set if the class has been dynamically registered by its provider. If a class-type has UMA_DYNAMIC set then metadata in a ‘‘Subclass Attributes’’ subclass of class ‘‘UMA Configuration’’ will always be available to describe its subclasses.

In the case when both built-in and dynamic forms of a subclass are defined, the MAP may select which is to be delivered by setting the UMA_WORKLOAD_DYNAMIC flag in the *umaStart*( ) call. The MLI service layer will then construct UDU messages according to whichever form has been selected.

Class and Subclass Handles

Data Pool classes and subclasses are manipulated by MAPs (MLI consumers) using class and subclass handles (of types UMAClass and UMASubClass). The handles have static values for built-in classes and subclasses. For dynamic classes and subclasses, the class and subclass handles are dynamically assigned by the MLI service functions (the local UMA Data Services Layer serving the requesting MAP). A common DSL service to all local MAPs provides consistent handle values to all its local MAPs for the duration of the DSL's existence.

An MLI subclass handle is identical to the DPD (or DCI) class identifier of the deepest class in the class hierarchy, that is the class that directly contains metrics. On the other hand, the class handle is the representative name for all the elements in the naming hierarchy that define the DPD class name from "uma" up to (but not including) the terminal class identifier.

Providers

The MLI recognizes a provider as an explicit entity that registers and supplies metric classes and subclasses. As the DCI does not explicitly support this notion of provider, it is assumed, by convention, that the highest level class (or in some cases, classes) registered by a DCI provider be used to identify a provider entity (or entities).

The metadata classes, subclasses and structures returned by *umaRequestConfig*( ) are as follows:

UMAClassAttr

The UMAClassAttr structure provides the i18n (internationalization) and ASCII name labels for a class, identifies the metric-containing subclasses available in the class and provides labels for them. A set of structures for a single class is returned in a message of subclass ''Class Attributes'' in response to a MLI call umaRequestConfig() with a defined class handle and a subclass wildcard (UMA_ALLSUBCLASSES). It is also sent when there is a configuration change affecting the class if a umaStart() MLI call has been issued for class ''UMA Configuration'' subclass ''Class Attributes'' (or UMA_ALLSUBCLASSES).

The UMAClassAttr structure is defined as follows:

```
typedef struct UMAClassAttr {
  UMAClassId        class;          /* class handle              */
  UMAVarLenDescr    classLabel;     /* class label struct        */
  UMAArrayDescr     subClassId;     /* array of DPD sunclass ids */
  UMAArrayDescr     subClassStatus; /* subclass status array     */
  UMAVarArrayDescr  subClassLabel;  /* subcl label struct array  */
} UMAClassAttr;
```

This structure contains the class handle followed by a descriptor for its label (the actual label contents are in the UMA message UDU Basic Segment VLDS). These are followed by an array descriptor for the array of Data Pool subclass identifiers available in this class. (These subclass identifiers are the same as those registered by a DCI provider as class identifiers for classes that directly contain metrics.) For each subclass there is an element of a subclass-status array that describes the implementation status of the subclass, a value consisting of one of UMA_NOTIMPLEMENTED, UMA_DISABLED or UMA_ENABLED. Additional flag values in this array describe whether each subclass is available as INTERVAL data, EVENT data (or both).

The MLI service layer (the DSL) will compose a label to correspond to a class handle that is the concatenation, separated by periods, of the DCI class labels for each class in the path to the terminal, metric containing class.

Each subclass label is contained in a UMALabel structure defined as follows:

```
typedef struct UMALabel {
  UMAUint4         size;   /* size of this structure        */
  UMAVarLenDescr   ascii;  /* descriptor for the variable   */
                           /* UMATextString for ascii label */
  UMAElementDescr  i18n    /* descriptor for the variable   */
                           /* length data for i18n label    */
  UMAVarLenData    data;   /* label data for ascii and i18n */
} UMALabel;
```

UMASubClassAttr

The UmaSubClassAttr structure is the container for the metadata describing a UMA subclass. It includes descriptors for instance tags, work unit identifiers and metrics (data) corresponding to a given DCI metric class identifier. It is sent in a message of UMA class "Configuration", subclass "Subclass Attributes". This message is sent either when solicited by an *UMARequestConfig*( ) call or when there is a configuration change affecting the relevant UMA subclass and the "Configuration" subclass "Subclass Attributes" (or UMA_ALLSUBCLASSES) has been requested in a *umaStart*( ) call.

```
typedef struct UMASubClassAttr {
  UMASubClassHandles handles;      /* cl/subcl handles,flags    */
  UMAVarArrayDescr  instanceTags;  /* instance tag              */
                                   /* descriptor array          */
  UMAVarArrayDescr  workUnits;     /* work unit descr array     */
  UMAVarArrayDescr  dataBasic;     /* basic data desc array     */
  UMAVarArrayDescr  dataOptional;  /* optional data descr array */
  UMAVarArrayDescr  dataExtended;  /* extended data descr array */
} UMASubClassAttr;
```

UMASubClassHandles

The UMA class and subclass handles are contained in the UMASubClassHandles structure in messages of class ‘‘UMA Configuration’’, subclass ‘‘Subclass Attributes’’:

```
typedef struct UMASubClassHandles {
  UMAClass     class;          /* UMA class handle      */
  UMASubClass  subClass;       /* UMA subclass handle   */
} UMASubClassHandles;
```

The following figure shows how the UMASubClassAttr structure, which is contained in a UMA UDU of class ‘‘Configuration’’, subclass ‘‘Subclass Attributes’’ describes a UMA data UDU. In the figure, a data subclass is mapped that has its Data Pool Document (DPD) identifier provided in two components. The first component is an array of UMAClassId items that are the elements of the fully-qualified class name and the second component is just the subclass identifier by itself. These are mapped to the handles ‘‘X’’ and ‘‘Y’’, respectively.

Note that in the descriptor subclass ‘‘Subclass Attributes’’, the Basic Segment VLDS (Variable Length Data Section) contains variable length data that itself includes arrays of additional descriptive data structures. These structures will be defined in the ensuing discussion.

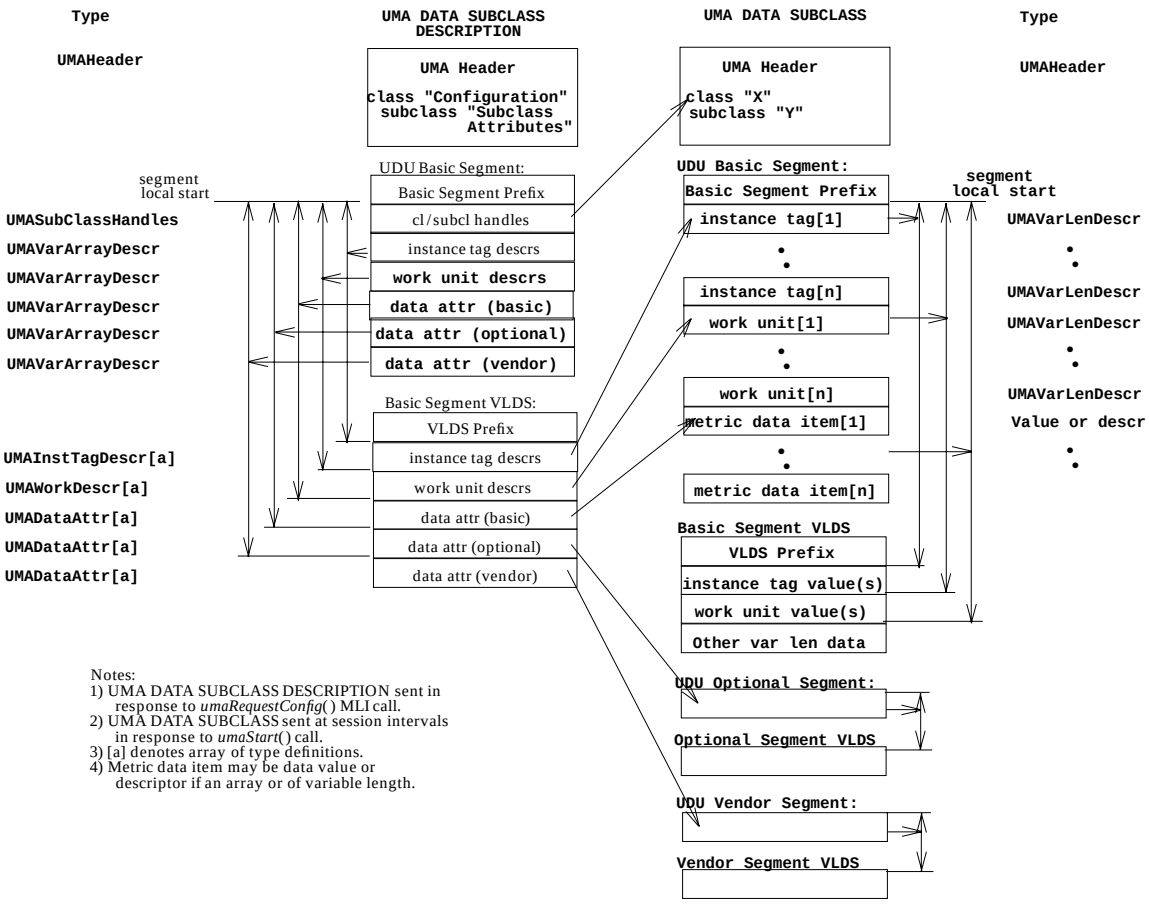


Figure 5-1 UMASubClassAttr Mapping to a Dynamic MLI Subclass

UMAInstTagDescr

UMA defines a fully qualified metric name as consisting of a class identifier, a metric instance identifier and a metric datum identifier. The metric class identifier is potentially multilevel and having multiple identifier components such as {data pool cpu per_thread}.

i. *Metric Class Identifier*

The mapping of the class identifier proceeds by assigning its lowest level (rightmost) class identifier component to the UMA subclass id in the UMA SubClassAttr structure and the remaining set of higher level components (to the left of it) to the array of identifier components for the UMA class id in this same structure. These metric class id components represent the segment of a complete path to instantiated subclasses that are defined within a UMA Data Pool.

ii. *Instance Levels*

The metric instance identifier uniquely identifies a metric class instantiation for a specific system object. Like the DCI metric class identifier, the metric instance identifier may have multiple levels, however, at each level the number of bytes in the identifier may vary. As an example, consider {chan1 bus2 cont1 disk2} which represents a specific channel, bus, controller and disk.

Metric instance identifiers are mapped level-by-level to instance tags in the UMA UDU and this mapping is described by the set of UMAInstTagDescr structures, one

per level. Instance tags are special data items that appear in an MLI basic data UDU immediately following header information.

```
typedef struct UMAInstTagDescr {
  UMAUint4        size;   /* size of this struct    */
  UMAUint4        flags;  /* indic mapped explicit, */
                          /* as data array indices  */
                          /* (lowest level only)    */
  UMADataType     type;   /* instance tag data type */
  UMAInstTagType  itType; /* instance tag type      */
  UMAUint4        itSize; /* tag size in bytes      */
  UMALabel        label;  /* ascii and i18n label   */
} UMAInstTagDescr;
```

When mapping to instance tags from instance identifiers, the order is preserved, meaning that the highest level component of the instance identifier appears (following the instance tag for the workload identifier) as the first instance tag, the next highest as the second instance tag, and so on. The instance tag data type determines whether the instance tag position in the metric data subclass fixed section for the segment is the tag data proper or if it is a data descriptor for variable length data, with the actual tag data in the data segment's VLDS pointed to by an offset in the descriptor. For example if the instance tag data type is UMA_UINT4, the tag data would be in the fixed data section, if the type is UMA_TEXTSTRING or UMA_OCTETSTRING, the fixed section would contain descriptors of type UMATextDescr or UMAElementDescr, respectively.

The UMAInstTagDescr structure describes levels of the metric instance identifier that are mapped explicitly to instance tags. However, the lowest level of the instance identifier may be mapped implicitly to an MLI UDU data array where this is deemed to be advantageous. In this case data type of the array index (corresponding to the instance tag data type) is always an integer. For example, the DCI_SYSCALL instance in the Data Pool Class "Processor", Subclass "Global System Call Counters" is mapped to an array with each index representing the numeric system call identifier.

UMAWorkDescr

The *UMAWorkInfo* definitions for a provider are a set of work units. These can be used jointly in a *umaStart*( ) specification to define filters and reporting granularities for per-work-unit subclasses. For example "Joe" might be a case of a work unit name corresponding to the *UMAWorkInfo* identifier "User Name" and "run_db" might be a case of a work unit name corresponding to the identifier "Command Name". When used jointly in the UMAWorkDefn of *umaStart*( ) structure they indicate that the Per Work-Unit subclass data is filtered for user "Joe" and the command "run_db". The available set of work unit identifiers depends on the provider type and is defined for for that provider type in the Data Pool.

There is some similarity between the UMAWorkInfo structure and instance identifiers and levels. A key difference is that a for a given provider type, a defined set of WorkInfo work unit identifiers is valid for all Per Work-Unit MLI classes and subclasses, while a fixed selection instance tags (UMA instance levels and types) are specifically defined for a particular MLI class and subclass. For example, only the instance types UMA_WORKID and UMA_PROCESSOR are valid in the Data Pool Subclass "Per Work Unit Processor Times", whereas any or all of the UMAWorkInfo identifiers defined for the provider may be used with this same subclass.

Another important difference between UMAWorkInfo and instance levels is that the implementation of UMAWorkInfo granularities is optional with any provider, while every provider must implement the instance levels as specified in the data pool. A provider has the following choices concerning the implementation of UMAWorkInfo:

a. Implement filtering as specified by a consumer's WorkInfo request for all or some of the UMAWorkInfo components,

b. Implement tagging only for all or some of the WorkInfo components,

c. Implement neither

The possible work unit identifiers supplied by a provider are defined by a set of UMAWorkDescr structures. This set of structures is contained in a message of UMA class ''UMA Configuration'', subclass ''UMA Work Units'' which is solicited with the *umaRequestConfig*( ) MLI call. The UMAWorkDescr structure is defined:

```
typedef struct UMAWorkDescr {
  UMAUint4        size;   /* size of this struct       */
  UMADataType     dType   /* Work Unit data type       */
  UMAWorkType     wType;  /* Work Unit type            */
  UMALabel        label;  /* ascii and i18n labels     */
} UMAWorkDescr;
```

UMADataAttr

The mapping of Data Pool metric values to the UMA Data UDU is described by the set of UMADataAttr structures. Data pool metric values are mapped to the same data types as defined in the Data Pool. Flag indicators in UMADataAttr indicate the implementation status of the item (UMA_NOTIMPLEMENTED, UMA_ENABLED or UMA_DISABLED) and whether the data is for the interval or is an absolute count. The descriptor type indicates whether the data is directly mapped at the offset or whether it is mapped through a descriptor to the UDU segment VLDS (Variable Length Data Section).

```
typedef struct UMADataAttr {
  UMAUint4        size;         /* size of this struct     */
  UMADataType     type;         /* data type of metric     */
  UMAUint4        status;       /* status: NOTIMPLEMENTED, */
                                /* DISABLED, ENABLED       */
  UMAUnit         units;        /* data units              */
  UMAUint4        dataFlags;    /* flags on units          */
                                /* rates, counts,          */
                                /* intervalization         */
  UMAUint4        offset;       /* to data item or descr   */
                                /* from segment start      */
  UMADescrType    descrType     /* Descriptor type         */
                                /* (or none)               */
  UMALabel        label;        /* ascii and i18n labels   */
} UMADataAttr;
```

DISCOVERY OF PROVIDERS, CLASSES, AND SUBCLASSES

This section summarizes the behavior of the MLI UMARequestConfig() call as used to discover what providers have registered and to discover what classes and subclasses are available for a given provider.

Wildcards

The defined values UMA_ALLCLASSES and UMA_ALLSUBCLASSES are wildcards for MLI classes and subclasses, respectively.

Available Providers

Executing the MLI call *umaRequestConfig*( ) with a class wildcard (UMA_ALLCLASSES), a subclass "UMA Providers", returns a message of class "UMA Configuration" and subclass "UMA Providers" that contains a list of providers available for this session's *destination*. The *provider*, *dclass* and *dsubcls* specifications are ignored. This class and subclass specification is also available to *umaStart*( ) as an event.

Available Classes for a Provider

Executing the MLI call *umaRequestConfig*( ) with a class wildcard (UMA_ALLCLASSES) and a subclass of "Implementation" returns a message of class ''UMA Configuration'' and subclass: ''Implementation'' that contains the status arrays for the classes and subclasses defined for the provider. These include both built-in and dynamic classes and subclasses. This class and subclass specification is also available to *umaStart*( ) as an event.

Available Subclasses for a Class

Executing the MLI call *umaRequestConfig*( ) with a defined class handle and a subclass "Class Attributes" returns the subclass ''Class Attributes'' of class ''UMA Configuration''. This returns the identifiers, labels and status of either built-in or dynamic subclasses. This class and subclass specification is also available to *umaStart*( ) as an event.

Retrieving Metadata for a Subclass

Executing the MLI call *umaRequestConfig*( ) with both class and subclass handles other than for the class "UMA Configuration" returns a message of class ''UMA Configuration'' and of subclass ''Subclass Attributes''.

Possible Work Units for a Provider

Executing the MLI call *umaRequestConfig*( ) with a class handle of UMA_ALL_CLASSES and a subclass of ''UMA Work Units'' returns a message of class ''UMA Configuration'' and of subclass ''UMA Work Units''.

umaRequestConfig( ) METADATA SUMMARY

The following table summarizes the use of the *umaRequestConfig*( ) call for obtaining UDU message descriptions. For, simplicity, the effects of the UMA_REPORT_DYNAMIC flag has been omitted from this table.

Specified Class	Specified Subclass	Returns "UMA Configuration" Subclass	For
UMA_ALLCLASSES	"UMA Providers"	"UMA Providers"	All available providers for this *sysid*
UMA_ALLCLASSES	"Implementation" "States" etc.	"Implementation" "States" etc.	All metric classes and subclasses for the provider
<class handle>	"Implementation"	"Implementation" "Class Attributes"	All metric subclasses in class denoted by Class Handle
<class handle>	"Subclass Attributes"	"Subclass Attributes	All metric subclasses in class denoted by Class Handle
UMA_ALLCLASSES	"Work Units"	"Work Units"	All UMAWorkInfo work units possible for this provider
UMA_ALLCLASSES	UMA_ALLSUBCLASSES	All of the above class/subclass information subclasses	All configuration and metric classes for the provider

NOTES

UMA_ALLCLASSES implies UMA_ALLSUBCLASSES.

umaRequestConfig( ) call responses are always returned to the originating MAP, regardless of the specified *destination*. These same status classes and subclasses may be directed to the session-specified *destination* as event data, by specifying them in *umaStart*( ) calls.

The *umaRequestConfig*( ) call differs from *umaStart*( ) in that it solicits an out-of-band response and does so only once for each solicitation.

DIAGNOSTICS

The returned value of *umaRequestConfig*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaRequestConfig*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_CLASS	UMR_PERMISSION	Access denied to this configuration data
	UMR_DISABLED	Specified configuration class not available
	UMR_INVALID	Specified configuration class invalid
	UMR_NOTIMPLEMENTED	Specified configuration class not implemented
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA
UMS_EOS	—	End of session encountered
UMS_PROVIDER	UMR_PERMISSION	Access denied to this provider
	UMR_UNKNOWN	Unknown data provider
UMS_SESSID	—	Invalid sessid specified
UMS_SESSION	UMR_RESOURCE	Lack of some system resources
UMS_SUBCLASS	UMR_DISABLED	Specified configuration subclass not available
	UMR_INVALID	Specified configuration subclass invalid
	UMR_NOTIMPLEMENTED	Specified configuration subclass not implemented

NAME

umaSeek - reposition to the interval of data with the specified timestamp.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaSeek(
    UMASessId      sessid,         /* in */
    UMAInt4        position,       /* in */
    UMAWhence      whence,         /* in */
    UMATimeSpec    *tstamp         /* in */
);
```

DESCRIPTION

The starting time for data reporting is nominally defined by the *stime* session parameter. Reporting normally continues forward from this point. However, it is also possible to position reporting by *seeking* to times either in Recent Data or UMADS facilities using *umaSeek*( ).

umaSeek( ) specifies that the next reporting interval corresponds to time *tstamp*, offset by number of intervals specified in the signed integer *position*.

1. The parameter *position* may be zero, negative, or positive.
2. The parameter *tstamp* may be any one of the following (subject to the values of the session attributes *RECENT_ONLY* and *HISTORY_ONLY*):
   - If the *whence* parameter is UMA_TSTAMP, the seek is relative to an arbitrary timestamp of type **UMATimeSpec** specified in the *tstamp* parameter.
   - A defined value UMA_CTIME, that specifies the session current time setting (timestamp of the last message retrieved by *umaGetmsg*( ),
   - A defined value UMA_LTIME, that specifies the latest time available from the *source*, subject to the constraint that the specified session end time is not exceeded,
   - A defined value UMA_STIME, that specifies the session start time.

If the session attribute *RECENT_ONLY* is set to TRUE, then seeks are confined to the data intervals in the Recent Data Facility.

If the session attribute *HISTORY_ONLY* is set to TRUE, then seeks are confined to UMADS.

All specified data *collection* (as opposed to *reporting*) continues during *umaSeek*( ) operations. The new data is not reported until it is reached in time sequence.

Note that the effect of a successful *umaSeek*( ) call is immediate, in that the next *umaGetMsg*( ) call will access data or status messages resulting from the repositioning. *umaSeek*( ) is defined only for sessions whose destination is the MAP.

If any of the session attributes *STIME, ETIME,* or *INTERVAL* have not been set, no positioning action will be taken and *umaSeek*( ) will return a UMS_ATTR status code with reason code UMR_INCOMPLETE.

NOTES

If a *umaSeek*( ) call is made to a time prior to the STIME, the next call to *umaGetMsg*( ) will return a message of class: UMA_CONDITION, subclass: UMA_INFO, source: UMA_DSL, id: UMA_STIME_BOUNDS. A subsequent call to *umaGetMsg()*( ) will return data from the first interval in the session.

f a *umaSeek*( ) call is made to a time prior to the ETIME, the subsequent calls to *umaGetMsg*( ) will return a message of class: UMA_CONDITION, subclass: UMA_INFO, source: UMA_DSL, id: UMA_EOS.

A seek with position 0 and UMA_CTIME will ‘‘reposition’’ to the current message. This will cause the next *umaGetmsg*( ) to return this same message.

A seek with UMA_LTIME must have a zero or negative position, otherwise an error return of status UMS_TIME, reason UMR_CONFLICT will result.

The following may result in unpredictable positioning:

i. A *umaSeek*( ) call with UMA_CTIME (current time) used prior to a *umaGetmsg*( ) call,

ii. Two consecutive calls to *umaSeek*( ) with no intervening successful call to *umaGetmsg*( ).

DIAGNOSTICS

The returned value of *umaSeek*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaSeek*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_ATTR	UMR_INCOMPLETE	All required attributes not specified
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA
	UMR_SYSERR	System error while communicating with UMA
	UMR_RECEIVE	Communications error while receiving messages from UMA
UMS_DEST	UMR_CONFLICT	Seek undefined with session destination
UMS_EOS	—	End of session encountered
UMS_SESSID	—	Invalid sessid specified
UMS_SESSION	UMR_RESOURCE	Insufficient system resources
UMS_TIME	UMR_CONFLICT	Conflicting time positions specified
	UMR_INVALID	Invalid time positions specified

NAME

umaSetAttr - specify (or change) a session's attributes.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaSetAttr(
    UMASessId       sessid,        /* in */
                    ...            /* in */
);
```

DESCRIPTION

Once a session is established (by a *umaCreate*( ) call), a MAP can issue a *umaSetAttr*( ) call to set or change the attributes for the session denoted by *sessid*.

The *sessid* parameter is followed by a null-terminated variable length list of name-value pairs (*attrpairs*), each pair consisting of a quoted attribute name followed by a comma and the attribute value. (Example: `"PRIO", 5`).

The session attributes and their defaults are:

Attribute	Type	Default by Source			
		File	UMADS	RECENT	UMADS+RECENT
STIME	UMATimeSec	Min Time	None	UMA_TIME_NOW	None
ETIME	UMATimeSec	Max Time	None	UMA_TIME_MAX	UMA_TIME_MAX
INTERVAL	UMATimeSec	File	UMADS	None	None
PRIO	UMAPrio	3	3	3	3
HISTORY_ONLY	UMABoolean	TRUE	TRUE	FALSE	FALSE
RECENT_ONLY	UMABoolean	FALSE	FALSE	TRUE	FALSE
PARTIAL	UMABoolean	FALSE	FALSE	FALSE	FALSE

The settings of these attributes affects the session context for other MLI calls in the following ways.

Data reporting for the specified session will be for intervals of size specified by attribute *INTERVAL* from *STIME* to *ETIME*.

Unless the session property UMA_NOTREGULAR has been set, intervals will be *regular* meaning:

1. If the number of seconds is greater than or equal to 3600 (1 hour), then the number of seconds must integrally divide 24*3600 and the equivalent number of hours must be a whole number. This is equivalent an interval specification of 1, 2, 3, 4, 6, 8, 12, or 24 hours.
2. If the number of seconds is greater than or equal to 60 but less than 3600, then the number of seconds must integrally divide 3600 and the equivalent number of minutes must be a whole number. This is equivalent to an interval specification of 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, or 30 minutes.
3. If the number of seconds is less than 60, then the number of seconds must integrally divide 60. This is equivalent to an interval specification of 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, or 30 seconds.

In any case, the interval duration must not be greater than 24 hours.

If the session start time or the reporting request (*umaStart*( )) occurs at a time *between* regular collection times, the first interval reported for the session may be for a shorter duration than that specified. All subsequent collections will be of the correct duration and at regular times.

The *STIME* and *ETIME* attributes are specified in units of calendar time measured in seconds since January 1, 1970 and are expressed as Coordinated Universal Time. The defined constant UMA_TIME_NOW may be used as a value for *STIME* or *ETIME* to specify an immediate start time or end time, respectively. The defined constant UMA_TIME_MIN may be used as a value for *STIME* to specify a session start time prior to any recorded data. The defined constant UMA_TIME_MAX may be used as a value for *ETIME* to specify that the end time is indefinite and that the session will be ended by an explicit *umaClose*( ) call.

If the session property UMA_NOTERM is set (in *umaCreate*( )) and the time specified by *ETIME* is reached, the session is automatically closed. If UMA_NOTERM is not set, a notification message is sent to the issuing MAP indicating *end-of-session* reached, in which case the MAP may either seek back in time or close the session. Note that setting the *ETIME* attribute to UMA_TIME_NOW results in an immediate end of reporting.

Note that the attributes *STIME* and *ETIME* must be specified prior to any *umaGetMsg*( ), *umaRequestConfig*( ) and *umaSetThreshold*( ) calls. In addition, *INTERVAL* must be specified prior to a *umaRelease*( ) that contains start or stop requests that have specified interval data.

The session start time *STIME* may be changed at any time before a previously specified start time has occurred. Once this time has passed, however, it can only be changed to an earlier time.

The effect of setting the STIME attribute when a session is initialized is to position the session's current time pointer to this value. Subsequent changes to the STIME attribute do not change the session's current time pointer.

The reporting end time *ETIME* may also be changed at any time before a previously specified time has occurred. Once this time has passed, it may be extended by a MAP destination (''UMA_MAP'') to a later time. For a file or UMADS destination, however, the session will be atomatically closed when this time is reached. See also the *umaGetMsg*( ) description for status and reason codes when the destination is a MAP.

Positive values of *INTERVAL* are taken to be the reporting interval size in seconds. A zero specification for *INTERVAL* is an error when the session source contains ''RECENT'' as part of its specification. When the source is UMADS or a file, a specified interval of zero is treated as a *wildcard* in that data is reported at the stored interval value in those sources; any positive integer interval specification is ignored in these cases. Alternatively, if the *INTERVAL* attribute is the defined constant UMA_TIME_MAX, a single interval will be defined from the time of first reporting to the session end time or the session close. This last capability is useful for collections that are to last the duration of a programmatically determined interval, as may happen in benchmarking activities.

The *INTERVAL* attribute does not affect data requested for reporting as events; reporting of events is on an individual basis for each event occuring in the start/stop window.

The relative priority of data delivery to the MAP, is designated by the *PRIO* attribute, a non-negative integer, with increasing values indicating decreasing priorities. This priority is in effect only when retrieving data with the *umaGetMsg*( ) call and specifying the defined value UMA_ANYSESSION for the requested session. (See *umaGetMsg*( )).

The session attributes *HISTORY_ONLY* and *RECENT_ONLY* limit searches to UMADS or to the Recent Data Facility, respectively, when *source* has been specified as "UMADS + RECENT". For further details, see the descriptions for these attributes under *attrparams* in Section 5.2.

When *regular* intervals are in effect, setting the session attribute *PARTIAL* to TRUE indicates that a requested change of interval size can take place at the next regular collection time for the shorter of the new and old intervals. This specification may result in a single truncated old or new interval at the time the change is made.

Summary of Data Source Specification

The following table shows the effect of various selections of source parameters and the session attributes *RECENT_ONLY* and *HISTORY_ONLY* in UMA. An attempt to set both *RECENT_ONLY* and *HISTORY_ONLY* to TRUE results in an error.

umaCreate( ) source	RECENT_ONLY	HISTORY_ONLY	Source Used
RECENT	FALSE	FALSE	Error
RECENT	TRUE	FALSE	Recent Data
RECENT	FALSE	TRUE	Error
UMADS	FALSE	FALSE	UMADS
UMADS	TRUE	FALSE	Error
UMADS	FALSE	TRUE	UMADS
UMADS + RECENT	FALSE	FALSE	Recent Data and UMADS
UMADS + RECENT	TRUE	FALSE	Recent Data
UMADS + RECENT	FALSE	TRUE	UMADS
Private File	FALSE	FALSE	Error
Private File	TRUE	FALSE	Error
Private File	FALSE	TRUE	Private File

EXAMPLE

The following example assumes that there is an active session with a sessid of sessid1. The function timec( ) in the example converts a string valued time specification to seconds since January 1, 1970).

```
#include <uma.h>

UMAStatusCode status;
UMASessId sessid1;

status = umaSetAttr(sessid1, "STIME", timec("2pm"),
         "INTERVAL", 120, "PRIO", 1,
         "RECENT_ONLY", TRUE, (char*)NULL);
```

DIAGNOSTICS

In the case of an error, no action will be taken on any specified attributes. The returned value of *umaSetAttr*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling *umaGetReason*( ). Status and reason codes that apply to *umaSetAttr*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_ATTR	UMR_INVALID	Invalid attribute name specified
	UMR_CONFLICT	Attributes conflict
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA
UMS_EOS	—	End of session
UMS_INTERVAL	UMR_INVALID	Invalid interval duration specified
	UMR_CONFLICT	Interval conflicts with source/destination
UMS_PRIORITY	UMR_INVALID	Invalid priority specified
UMS_SESSID	—	Invalid sessid
UMS_SESSION	UMR_RESOURCE	Resource not available
UMS_TIME	UMR_INVALID	Invalid time specified
	UMR_CONFLICT	STIME/ETIME conflict with other attribute

NAME

umaSetThreshold - establish or change UMA threshold values.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaSetThreshold(
    UMASessId       sessId,             /* in */
    UMAProvider     provider,           /* in */
    UMAClass        class,              /* in */
    UMASubClass     subClass,           /* in */
    UMASegFlags     segment,            /* in */
    char            *selectExpr         /* in */
    char            *workload           /* in */
);
```

DESCRIPTION

The function *umaSetThreshold*( ) formats a message to the UMA facility to report data for a *class*, *subClass* and *segment* from *provider* as specified in a prior *umaStart*( ) call in this same session, only if the selection expression *selectExpr* evaluates to TRUE. Filtering may be reset either by field or for an entire selection expression using the reset expression *resetExpr*. The action of *umaSetThreshold*( ) can be restricted to a previously defined constructed workload (see *umaStart*( )) by use of the *workload* parameter. A NULL pointer for *workload* indicates that the *umaSetThreshold*( ) action is to be global.

Note that *umaSetThreshold*( ) calls do not take effect until a *umaRelease*( ) call is made. This same *umaRelease*( ) call can contain *umaStart*( ) calls in its scope, as long as they precede the related *umaSetThreshold*( ) call(s). Multiple calls for the same *session*, *provider*, *class* and *subClass* are ORed, meaning that if any one of the selection expressions evaluates to TRUE, the designated subclass segments are reported to the calling MAP. If a *umaSetThreshold*( ) call is made referencing an unstarted class, subclass, or segment, no action is taken and the status UMS_SUBCLASS and reason UMR_NOTSTARTED are returned.

A select expression is defined as follows:

```
select_expr:
      <compare_expr>
    | <compare_expr> <lo> <select_expr>
    | <reset_expr>

lo:
      '|'  /* or */
    | '&'  /* and */

compare_expr:
      <field_des> <ro> <numeric_value>

reset_expr:
      RESET             /* Resume segment reporting    */
    | <field_des> RESET  /* Reset at the field level    */

ro:
      LT   /* less than               */
    | LE   /* less than or equal to    */
```

```
    | EQ   /* equal to                */
    | GT   /* greater than             */
    | GE   /* greater than or equal to */
    | NE   /* not equal to             */
    | NLT  /* not less than            */
    | NGT  /* not greater than         */

field_des:
      $<field_index>    /* ith scalar field in UMA data  */
                        /* message or any element,       */
                        /* if an array                   */
    | $<namespace_ref>  /* Data Pool namespace reference */

field_index:
      <integer>

namespace_ref:
      <DP  xref>  /* Fully qualified Data Pool xref      */
                  /* including the enclosing < > marks.  */
numeric value:
      a value convertible by %f input conversion in C.
```

As the logical operator '&' has higher precedence than '|', parentheses may be required to indicate the desired meaning. In addition, the relational operators (ro) have a higher precedence than the logical operators (lo).

EXAMPLES

```
"$3 GT 75"

"$5 LE 10 | $6 GT 50"

"$5 LE 10 & ($7 GT 60 | $9 GE 70)"

"$<2.1.93> GT 10 & $<2.1.94> GT 5)"   /* DCI xrefs */

"$5 RESET"  /* Remove $5 from filtering expression */
```

DIAGNOSTICS

The returned value of *umaSetThreshold*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaSetThreshold*( ) include:

STATUS	**REASON**	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_CLASS	UMR_DISABLED	Specified class not available
	UMR_INVALID	Specified class invalid
	UMR_NOTIMPLEMENTED	Specified class not implemented
UMS_COM	UMR_SEND	Communications error when sending to UMA
UMS_EXPRESSION	UMR_INVALID	Invalid selection expression
UMS_FIELD	UMR_DISABLED	Specified field disabled
	UMR_INVALID	Specified field invalid
	UMR_NOTIMPLEMENTED	Specified field not implemented
UMS_SESSID	—	Invalid session specified
UMS_SUBCLASS	UMR_DISABLED	Specified subclass disabled
	UMR_INVALID	Specified subclass invalid
	UMR_NOTIMPLEMENTED	Specified subclass not implemented

NAME

umaStart - specify the classes and subclasses of data to be reported.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaStart(
    UMASessId       sessid,             /* in */
    UMAProvider     provider,           /* in */
    UMAClass        class,              /* in */
    UMASubClass     subClass,           /* in */
    UMASegFlags     segFlags,           /* in */
    UMAWorkDefn     *workDefn           /* in */
);
```

[*Editor's note for the CAE Draft*:

There are two major areas of revision in *umaStart*( ). The first is to make specification of *UMAWorkInfo* and instance filters flexible and extensible in the DCI. This is now done by the use the WorkDefn structure. The second change is to be able to assign a identifier to a workload constructed by *UMAWorkInfo* filtering and/or summarization so that the workload can be manipulated or accessed during the same session or at a later time in a reference to UMADS.

Neither of these changes affect the ability to start classes and subclasses individually as described in the Preliminary Specification.]

DESCRIPTION

The function *umaStart*( ) specifies the class and subclass of data to be reported for the session denoted by *sessid*. Data reporting will be started for the specified *provider*, *class*, and *subClass*. (UMA will initiate data collection, if necessary). The defined symbols UMA_ALLCLASSES and UMA_ALLSUBCLASSES may be used for *class* and *subClass* respectively, to indicate that all classes and all subclasses are to be selected. Note that UMA_ALLCLASSES implies UMA_ALLSUBCLASSES.

Either interval or event forms of reporting may be selected for subclasses where these forms are defined to be valid in the Data Pool Definitions (see reference **DPD**). Events are usually reported out-of-band, that is they are reported ahead of any other currently queued data messages. In-band delivery can be specified using the UMAWorkDefn structure described below.

The Data Pool Definition (see reference **DPD**), describes three conceptual data segments or groupings within a UMA data subclass:

1. Basic - This is a segment of universally supplied data for the subclass as defined by the Measurement Data Pool.
2. Optional - This is a segment of data whose structure and contents are defined by the Measurement Data Pool but this segment may or may not be present in a particular implementation.
3. Extension - This is data whose structure and contents are defined by a specific vendor or reseller for the system. This segment may or may not be present.

segFlags indicates which data segments are to be reported (UMA_BSEG, UMA_OSEG and UMA_ESEG for basic, optional, and extension, respectively).

destination (set in the *umaCreate*( ) call) controls whether the data is reported to the requesting MAP or if it is to be recorded in UMADS or in an output file.

Normally, UMA will wait to start collection and reporting after a *umaRelease*( ) call is made subsequent to the *umaStart*( ), however the *umaStart*( ) will not actually take effect until the session attributes *STIME, ETIME, INTERVAL* have been set.

In addition, *umaStart*( ) requests will be validated and assembled within UMA but not acted upon until two conditions are satisfied:

1. the time specified by attribute *STIME* has occurred

   and

2. a *umaRelease*( ) call has been made.

This will allow a coordinated start time for a set of session measurements within the time window defined by attributes *STIME* and *ETIME*

Specification of *UMAWorkInfo* values and instances for reporting will be by use of the UMAWorkDefn structure defined as follows:

```
typedef struct UMAWorkDefn {
  UMAUint4          size;          /* size of this struct     */
  UMAUint4          rFlags;        /* reporting flags         */
  UMATextDescr      workIdSpec;    /* workload id offset/size */
  UMAUint4          granularity;   /* granularity request     */
  UMAUint4          wFlags;        /* workload  flags         */
  UMAVarLenDescr    workInfoSpec   /* offset to WorkInfo data */
  UMAVarLenDescr    instanceSpec   /* offset to instance data */
  UMAVarLenData     data;          /* WorkInfo, instance specs*/
} UMAWorkDefn;
```

The UMAWorkDefn structure provides the definition of a constructed workload and specification under the control of a number of flags described below. If the structure pointer for workDefn in the input parameters is NULL, then *UMAWorkInfo* selection, instance selection, workload selection by identifier, or summarization will not be requested.

The workload specification flags include:

UMA_WORKLOAD_ABSOLUTE
: Requests that this workload be defined by use of absolute (not intervalized) counters for this class and subclass.

UMA_WORKLOAD_COMPLEMENT
: Requests that this definition is of a complement constructed workload, meaning that a metric values in this workload consist of those available in the corresponding global metric class and subclass minus the values of the corresponding metrics in this per-work-unit class and subclass.

UMA_WORKLOAD_SUMMARIZED
: Requests that the data for this constructed workload be summarized to level specified in the UMAWorkDefn structure granularity flag even if the data provider to UMA does not directly support this granularity. Satisfying this request does however depend on the data provider's being able to tag this *UMAWorkInfo* level. This is indicated in the metadata provided by the UMA message of class "Configuration", subclass "Subclass Attributes" as a supported *UMAWorkInfo* level. A request for summarization at a granularity level that is not available is an error (UMS_WORKLOAD, UMR_GRANULARITY).

The reporting specification flags include:

UMA_REPORT_DYNAMIC
: Requests that the DCI return structure form of this subclass (Canonical A) should be reported by the MLI. Otherwise, the built-in structure form (Canonical C) is reported.

UMA_REPORT_EVENT
: Requests that the asynchronous event form of the specified class and subclass be reported to the MAP.

UMA_REPORT_WORKLOAD
: Requests that the workload data form of the specified class and subclass be reported to the MAP.

UMA_EVENT_FINALDATA
: Requests that the final data event for the specified class and subclass be reported to the MAP. Final data can be reported on components of the workload requested via the selection criteria to the extent supported by the data provider. For example, if commands ‘‘abc*’’ are selected for user 123, then, final data will reported whenever each such command teminates.

UMA_EVENT_INBAND
: Requests that event messages of the specified class and subclass be delivered to the MAP in_band, that is, they are to be reported in time sequence along with any queued data messages.

The UMAWorkSpec structure for specifying selection of *UMAWorkInfo* values is defined:

```
typedef struct UMAWorkSpec {
  UMUint4           wSpecSize;  /* size of this structure     */
  UMAUint4          wSelect;    /* WorkInfo level select bits */
  UMAVarArrayDescr  wSpecDescr; /* WorkInfo specs descr       */
  UMAVarLenData     data;       /* WorkInfo level spec value  */
} UMAWorkSpec;
```

And each WorkInfo level specification is in a UMAWorkLvlSpec structure:

```
typedef struct UMAWorkLvlSpec {
  UMAUint4        wLvlSize;    /* size of this structure    */
  UMAUint4        wLvlType;    /* enum type of WorkInfo spec*/
                               /* UMA_TEXTSTRING,           */
                               /* UMA_UINT4, etc assigned   */
  UMAVarLenData   data;        /* work spec/expr            */
} UMAWorkLvlSpec;
```

Note that the descriptor entries workIdSpec, workInfoSpec measure offsets from the start of their containing structure, UMAWorkDefn, and wSpecDescr measures offset from the start of UMAWorkSpec. Also, each structure’s initial size field (of type UMAUint4) is preceded by sufficient padding so that the size field begins on a 4-byte word boundary.

The *UMAWorkInfo* levels and their data types are described by the UMAWorkDescr structures contained in a UMA message of class "UMA Configuration", subclass "WorkInfo Attributes". If the data type in UMAWorkDescr is text, a text regular expression may be used for selection in UMAWorkLvlSpec [3]. If the *UMAWorkInfo* data type is integer, either an integer or a text regular

3. More specifically, a Basic Regular Expression as defined in the XSI Specification.

expression with integer semantics may used as a selection specification (e.g. a range such as [22-25]). A wildcard integer value `'0xffffffff'` specifies that all *UMAWorkInfo* values are to be selected.

The UMAInstSpec structure for specifying selection of instances is defined:

```
typedef struct UMAInstSpec {
  UMAUint4          iSpecSize;   /* size of this structure     */
  UMAUint4          iSelect;     /* instance level select bits */
  UMAVarArrayDescr  iSpecsDescr; /* instance level specs descr */
  UMAVarLenData     data;        /* instance level spec values */
} UMAInstSpec;
```

And each instance level specification is defined using the UMAInstLvlSpec structure:

```
typedef struct UMAInstLvlSpec {
  UMAUint4        iLvlSize;    /* size of this structure    */
  UMAUint4        iLvlType;    /* enum type of instance spec*/
                               /* UMA_TEXTSTRING,           */
                               /* UMA_UINT4, etc assigned   */
  UMAVarLenData   data;        /* instance spec/expr  */
} UMAInstLvlSpec;
```

Here again, the descriptor entries (instanceSpec, iSpecsDescr) measure offsets from the start of their containing structures (UMAWorkDefn and UMAInstSpec, respectively), and each structure's initial size field (of type UMAUint4) is preceded by sufficient padding so that the size field begins on a 4-byte word boundary.

The instance levels and their data types are described by the UMAInstTagDescr structures in a UMA message of class "UMA Configuration", subclass "Subclass Attributes". If the instance level data type is integer, either an integer or a text regular expression with integer semantics may used as a selection specification (e.g. a range such as [22-25]). A special wildcard integer value `'0xffffffff'` specifies that all instance values are to be selected.

As an example using the Unix Data Pool Definitions, consider a UMAWorkDefn structure that requests reporting granularity at the process level for command names commencing with the letters abc (*UMAWorkInfo* command name), executing on processors 2 through 4 (processor instances). The layout of the UMAWorkDefn structure would be as follows (assuming a compiler that aligns UMAUint4 on 4-byte word boundaries):

```
0x0000006c                /* size of this structure (108)      */
UMA_REPORT_WORKLOAD       /* report workload flag (rFlags)     */
0x00000020                /* offset to workload id (32)        */
0x00000007                /* size of workload id               */
UMA_WORKINFO_PROCESS_ID   /* report process granularity        */
0x00000000                /* workload is intervalized (wFlags  */
0x00000024                /* offset to WorkInfo data (36)      */
0x00000048                /* offset to instance data (72)      */
"work001"                 /* workload identifier               */
0x0                       /* padding (1 byte)                  */
0x00000021                /* size of UMAWorkSpec structure (36)*/
UMA_WORKINFO_COMMAND_NAME /* WorkInfo selection by command     */
0x00000010                /* offset to array of wrk selns (16) */
0x00000001                /* array count is 1 item (command)   */
0x00000017                /* size of UMAWorkLvlSpec struct     */
0x00000007                /* type is UMA_TEXTSTRING (enum)     */
0x00000009                /* size of text string struct (9)    */
"abc.*"                   /* regexpr string for command        */
0x000000                  /* padding (3 bytes)                 */
0x00000024                /* size of UMAInstSpec structure (36 */
UMA_PROCESSOR             /* bit value for processor instance  */
0x0000000f                /* offset to array of inst selections*/
0x00000001                /* array count is 1 item (processor) */
0x00000011                /* size of UMAInstLvlSpec struct (17)*/
0x00000007                /* type is UMA_TEXTSTRING (enum)     */
0x00000009                /* size of text string struct (9)    */
"[2-4]"                   /* processor 2 through 4 selection   */
0x000                     /* padding (3 bytes)                 */
```

Note that The double-quote (") is used here to indicate that the content of the data field is text, that is, the quote does not appear in the data.

DIAGNOSTICS

The returned value of *umaStart*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaStart*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_CLASS	UMR_DISABLED	Specified class not available
	UMR_INVALID	Specified class invalid
	UMR_NOTIMPLEMENTED	Specified class not implemented
	UMR_PERMISSION	Access denied to this class/subclass
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA
UMS_EOS	—	End of session
UMS_EVENT	UMR_NOTIMPLEMENTED	Event not implemented
UMS_FLAGS	UMR_INVALID	Specified flag is invalid
UMS_NODE	UMR_UNKNOWN	Node not recognised
UMS_PROVIDER	UMR_PERMISSION	Access denied to this provider
	UMR_UNKNOWN	Unknown data provider
UMS_SESSID	—	Invalid session specified
UMS_SESSION	UMR_RESOURCE	Resource not available
UMS_SUBCLASS	UMR_DISABLED	Specified subclass not available
	UMR_INVALID	Specified subclass invalid
	UMR_NOTIMPLEMENTED	Specified subclass not implemented
UMS_WORKLOAD	UMR_GRANULARITY	Workload granularity not supported
	UMR_REGEXP	Invalid regular expression

NAME

unaStop - stop reporting of data classes and subclasses.

SYNOPSIS

```
#include <uma.h>

UMAStatusCode umaStop(
    UMASessId       sessid,             /* in */
    UMAProvider     provider,           /* in */
    UMAClass        class,              /* in */
    UMASubClass     subClass,           /* in */
    UMASegFlags     segFlags,           /* in */
    UMAFlushFlags   flushFlags,         /* in */
    UMAWorkDefn     *workDefn;          /* in */
);
```

DESCRIPTION

umaStop( ) requests the UMA facility to stop reporting the data specified by *provider*, *class*, *subClass*, and, optionally, a workload as defined in workDefn over *sessid*. If the pointer to the UMAWorkDefn structure is not NULL, *umaStop*( ) terminates reporting on the specified class and subclass. This action can be limited to a specified workload and/or workload components by optionally specifying them in a UMAWorkDefn structure parameter. (Note that collection may continue if other sessions have requested the same subclass.)

The defined symbols UMA_ALLCLASSES and UMA_ALLSUBCLASSES may be used for *class* and *subclass*, respectively, to indicate that all classes and all subclasses are to be stopped. Note that the symbol UMA_ALLCLASSES implies UMA_ALLSUBCLASSES.

The *segFlags* parameter indicates which data segments are to be stopped (UMA_BSEG, UMA_OSEG, UMA_ESEG, UMA_ASEG for basic, optional, extension, and all, respectively). Not specifying *segflags* is equivalent to having specified UMA_ASEG.

The *flushFlags* parameter further specifies which previously issued *umaStart*( )s are cancelled. The flag settings include UMA_ALLSTARTED, UMA_HELD, or UMA_RELEASED to indicate whether all requests or only those held or only those released are to be stopped.

In addition to specifying a class and subclass to be stopped, the caller can specify that reporting of a previously defined workload or class/subclass components be terminated for this session using the UMAWorkDefn structure (see *umaStart*( ) for a full description of the structure). The fields in the UMAWorkDefn structure that are used and not used by *umaStop*( ) are shown here:

```
typedef struct UMAWorkDefn {
    UMAUint4        size;         /* size of this struct     */
    UMAUint4        rFlags;       /* reporting flags         */
    UMATextDescr    workIdSpec;   /* workload id offset/size */
    UMAUint4        granularity;  /* not used in umaStop()   */
    UMAUint4        wFlags;       /* not used in umaStop()   */
    UMAVarLenDescr  workInfoSpec  /* not used in umaStop()   */
    UMAVarLenDescr  instanceSpec  /* not used in umaStop()   */
    UMAVarLenData   data;         /* not used in umaStop()   */
} UMAWorkDefn;
```

Unused fields of type UMAVarLenData must have valid size components but any contained data will be ignored by *umaStop*( ). The workload identifier (work_id) limits the action of *umaStop* to the specified workload. If it is a null string, the reporting flags (rFlags) can still be used to stop components for all workloads:

UMA_REPORT_EVENT
: Requests that reporting of the asynchronous event form of the specified class and subclass be terminated (for this workload identifier).

UMA_REPORT_WORKLOAD
: Requests that the reporting of the workload data form of the specified class and subclass be terminated (for this workload identifier).

UMA_EVENT_FINALDATA
: Requests that the reporting of final data event for the specified class and subclass be terminated (for this workload identifier).

UMA_EVENT_INBAND
: Requests that in-band reporting of event messages of the specified class and subclass be changed to out-of-band reporting if event reporting is still to be continued (for this workload identifier).

DIAGNOSTICS

The returned value of *umaStop*( ) indicates the general outcome (status code). Supplementary reason code for a failed status can be obtained by calling the *umaGetReason*( ) function. Status and reason codes that apply to *umaStop*( ) include:

STATUS	REASON	EXPLANATION
UMS_SUCCESS	UMR_NOREASON	No error encountered
UMS_CLASS	UMR_DISABLED	Specified class not available
	UMR_INVALID	Specified class invalid
	UMR_NOTIMPLEMENTED	Specified class not implemented
UMS_COMM	UMR_SEND	Communications error when sending messages to UMA
UMS_EOS	—	End of session
UMS_EVENT	UMR_NOTIMPLEMENTED	Event not implemented
UMS_FLAGS	UMR_INVALID	Specified flag invalid
UMS_NODE	UMR_UNKNOWN	Specified node not recognised
UMS_PROVIDER	UMR_UNKNOWN	Unknown data provider
UMS_SESSID	—	Invalid session specified
UMS_SESSION	UMR_RESOURCE	Resource not available
UMS_SUBCLASS	UMR_DISABLED	Specified subclass disabled
	UMR_INVALID	Specified subclass invalid
	UMR_NOTIMPLEMENTED	Specified subclass not implemented

Chapter 6

UMA Message and Header Formats

UMA API messages, called UMA Data Units (UDUs), consist of a tagged header and either a control segment or one or more data segments. A limited degree of ASN.1/BER encoding has been provided by incorporating ASN.1 tags and length descriptors at the message, header, and segment levels.

6.1 UDU Message Headers

The class and subclass of a message are indicated in the message header. These are as defined by the UMA API unless flags (in the *mh_flags* field) indicate that they are specific to a provider. The provider identifier is specified in the header (in the *mh_provider* field). Thus, depending on these flag settings, a provider may be a source of either datapool-defined data or of data that is unique to that provider. An example of the difference would be a datapool-defined DBMS data class that is provided by the XYZ product designated by the integer nnn (flag setting off) versus a DBMS data class that is specific to XYZ (flag setting on). Both would have the integer nnn in the provider identifier field.

A UDU Indicators field in the message header includes specification of data encoding formats. Three canonical formats are defined:

1. Canonical A, which provides data in the native format of the processor where captured along with metadata (see reference **DCI**). In this form the metadata must be used to interpret subclass data.
2. Canonical B, which encodes all items following the UDU Indicators field according to ASN.1/BER tag-length-value format. Where tags and lengths are already specified in the message or segment headers, they would be incorporated into the BER encoding for Canonical B Format.
3. Canonical C, which permits data to be mapped to standardised C structures. In this format there is limited ASN.1/BER encoding at the *constructed types* level (for example, headers and data segments). In this format, metadata may also be made available.

6.2 UDU Control Segments

UDU control segments contain directives to UMA to take some action (for example, create a session, start measuring some class/subclass, query a status, etc.)

The UDU header indicates that the UDU contains a control segment by a flag and by the control segment class.

6.2.1 Compatibility Support

The UDU header optionally contains a ‘‘Protocol Section’’ that provides UMA-specific, vendor-specific, and platform-specific level and version information that may be needed for interoperable or vendor-optimized communications between UMA Data Services Layers.

The protocol parameters supported are:

UMA Communications Protocols (mhpro_umacps)
: Indicates communications protocols supported (indicated in the acknowledgement) - Flag `0x0001` is TCP/IP with socket interface

UMA Message Format Specification Level (mhpro_umamflvl)
: Indicates the UMA message header and UDU format specification level — Level 1 is the level specified by this document.

UMA+Message Specification Base Level (mhpro_umasblvl)
: Indicates the document specification base - Level 1 is the level specified by this document.

UMA Vendor Name
: Indicates vendor name in text form (mhpro_vndname).

UMA Vendor Protocol Level
: Indicates private vendor protocol level available (mhpro_vndplvl).

Protocol level indicators are positive integers, nominally 0-127.

6.2.2 Status Reporting

Control segments can report status back to the MAP. For all status messages, the class is the defined constant C_CONDITION. The subclass identifies the severity.

The source field, *cs_source*, identifies the UMA component affected:

- Data Capture Layer
- Data Services Layer
- UMADS
- Recent data facility.

The subclass of a message identifies the severity of the problem. The severity may be one of the following:

- Informational
- Warning
- Severe (the session has had an unrecoverable error)
- Fatal (UMA has had an unrecoverable error).

The body identifies the condition (through defined constants). For each source, there is a separate list of possible conditions. In addition, the message includes a textual description of the problem, for example:

```
"Data not collected for this interval"
"Requested timestamp not contained in any interval"
"End of session encountered"
```

The text description is useful for reporting the condition back to a user. It can be used, for example, in printing a status message to the user’s terminal.

6.2.3 UMA API Message Header Format for Control UDUs

The following table shows the proposed UMA message header format for control messages.

Table 6-1 UMA API UMA Control Message Header

Data Type	Name	Description
UMAOctetString[4]	mh_msgtag	BER-encoded UMA message indicator tag 0x7fd5cd41
UMAOctetString[1]	mh_msglenlen	ASN.1/BER length of mh_msglen
	1000 0011	Leading bit always 1; shows mh_msglen length of 3 octets.
UMAOctetString[3]	mh_msglen	Length of the message including the standard header starting at the next field
UMAOctetString[3]	mh_hdrtag	UDU Header Tag 0xbf1081 (includes length of mh_hdrlen in the low order 7 bits). The first octet of the string marks the global start of the message.
UMAOctetString[1]	mh_hdrlen	Length of UDU Header
UMAOctetString[3]	mh_indtag	UDU Indicators tag (for mh_flags) indicators - 0x9f3081 (includes length of mh_flags in the low order 7 bits).
UMAOctetString[1]	mh_indlen	Length of UDU Indicators
UMAMsgFlags	mh_flags	UDU Indicators
	1...	UDU contains control information
	010.	Canonical A format[4]
	001.	Canonical B format
	011.	Canonical C format
	1... 0...	16-bit byte order H to L[5]
	1... 1...	16-bit byte order L to H
	1... .0..	integer component order H to L
	1... .1..	integer component order L to H
	1... .... 1...	First message for this class, this system ID

4. Format designated (A, B, or C) is in effect for the remainder of the message following the optional Protocol Section.
5. Local DSL preferred byte order in Create and Reconnect protocol.
6. Specification Level indicators are positive integers, nominally 0-127

Data Type	Name	Description
	`1... .... .1.. ....`	Last message for this class, this system ID
	`1... .... ...0 ....`	dst not in effect
	`1... .... ...1 ....`	dst in effect
	`1... .... .... 1...`	Protocol Section present
UMAOctetString[3]	mh_protag	UDU Protocol Section tag (if section present) - 0xbf708n (includes length of protocol section, n, in the low order 7 bits).
UMAOctetString[3]	mhpro_wdsztag	Tag for this platform's wordsize - 0x9f7181 (includes length of mhpro_wdsize in the low order 7 bits).
UMAOctetString[1]	mhpro_wdsize	This platform's wordsize in bytes
UMAOctetString[3]	mhpro_cplvltag	Tag for UMA Communication Protocols - 0x9f7384 (includes length of mhpro_umaplvl in the low order 7 bits).
UMAUint4	mhpro_umacps	UMA Message Communications Protocols supported
UMAOctetString[3]	mhpro_mflvltag	Tag for UMA Message Format Level - 0x9f7581 (includes length of mhpro_umamflvl in the low order 7 bits)
UMAOctetString[1]	mhpro_umamflvl	UMA Message Format Specification Level[6]
UMAOctetString[3]	mhpro_sblvltag	Tag for UMA Specification Base Level - 0x9f7681 (includes length of mhpro_umamslvl in the low order 7 bits)
UMAOctetString[1]	mhpro_umasblvl	UMA Message Specification Base Level
UMAOctetString[3]	mhpro_vndtag	Tag for Vendor Name string - 0x9f778n (includes length of mhpro_vndname in the low order 7 bits, max length 64).
UMAOctetString[]	mhpro_vndname	Vendor Name - Text not null terminated
UMAOctetString[3]	mhpro_vndptag	Vendor Protocol Level tag - 0x9f7981 (includes length of mhpro_vndplvl in the low order 7 bits).
UMAOctetString[1]	mhpro_vndplvl	Vendor Protocol Level
UMATimeStamp	mh_time	Timestamp of message creation

Data Type	Name	Description
UMAClass	mh_class	UMA class of the message
UMASubclass	mh_subclass	UMA subclass of the message
UMAOctetString[8]	mh_address	Host network address that generated the data in the message
UMAOctetString[4]	mh_addr_family	Host network address type (e.g. internet, SNA, ...)[7]

6.2.4 UMA API Control Segment Format for Control UDUs

The following table shows the UMA control segment format for control messages (Canonical C format shown).

Data Type	Name	Description
UMAOctetString[4]	cs_segtag	BER-encoded control segment tag 0xbfd5c300 ("UC", counter 0x00)
UMAOctetString[1]	cs_seglenlen	ASN.1 / BER length of cs_seglen
	`1000 0011`	Leading bit always 1; indicates cs_seglen length of 3.
UMAOctetString[3]	cs_seglen	Length of the control segment that follows (not including this field); the next octet marks the local start position for this segment
	.	
	Control Segment Content	
	.	

Table 6-2 UMA API UDU Control Segment

7. The *provider* identifier is an integer which may refer to the operating system kernel, its subcomponents, or to subsystems such as DBMSs, transaction managers, or other applications. A default value of the identifier may be established by vendors of subsystems for their products. However these must be capable of being overridden by systems administrators at sites where they are in use. The provider identifier is mapped to text labels by the subclass ‘‘UMA Providers’’ of the class ‘‘UMA Configuration’’.

6.2.5 Hints

The message body provides three hint fields. The following table lists the conditions that use the hint fields.

Condition	Hint 1	Hint 2	Hint 3
U_EGAP RHIST_GAP	Start time of gap	Duration of gap	not used
DSL_NODATA	Start time of interval	Duration of interval	not used
DSL_GAP	Start time of gap	Duration of gap	not used
DSL_EOS	Session end time	not used	not used
DSL_INTVL	Closest available interval	not used	not used
U_EOF	Timestamp of last interval in data source. Either UMADS or file.	not used	not used

Table 6-3 Conditions Using Hint Fields

When provided, the MAP can use the hints to respond to the condition. Here are two examples:

1. If an interval of data does not exist, a gap message will be sent. The first two of the hint fields are used with gap messages: the first contains the gap start time and the second contains the gap duration. A MAP graphing the data can use the hints in a gap message to display the start time and duration of the gap.
2. Another situation that uses the hint fields is when a MAP requests data for an interval that UMA is unable to provide. This situation will happen if the interval is not already being collected and the site-configured maximum number of collectors are active. In this case, hint 1 provides the closest available interval size.

Each hint field is a union of the following data types:

- UMAInt4
- UMAInt8
- UMATimeSec
- UMATimeNsec
- UMATimeUsec
- UMATimeStamp.

Each hint field is preceded by a hint type field, which identifies the type actually used for the hint.

6.3 UDU Data Segments

UDU data segments contain either interval or event data (traces are treated as a form of event data). The presence of either category is indicated by a flag in the message header. Interval data is that which is specifically scheduled for capture at each expiration of a specified time interval. The data reported for the interval is the increment of the item values over the interval.

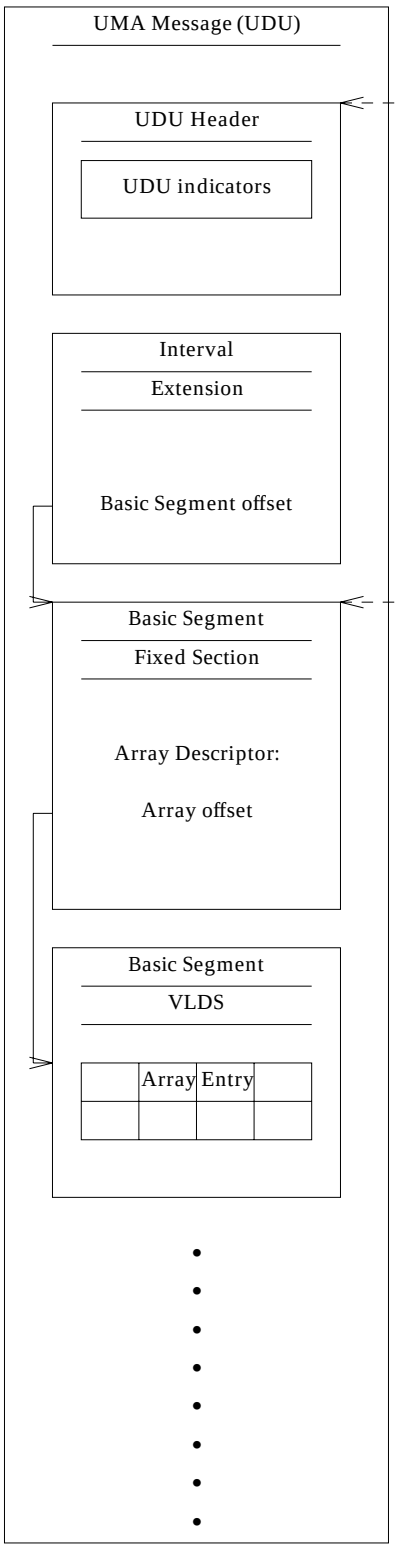
Within the categories of interval or event data, there are currently three kinds of data segments as defined by the Data Pool Definitions (see reference **DPD**):

1. Basic data, which each implementation must supply
2. Optional data, which is generally available in open operating systems but is not mandatory
3. Extension data, which is associated with a class and subclass, but is vendor and/or implementation-specific.

Following each fixed data segment section (Basic, Enhanced, Extension) there is a VLDS (Variable Length Data Section) that may contain one or more data items of varying length (for example, text strings, arrays). Each of these variable length items is pointed to and described by descriptors of the appropriate type (for example UMATextDescr, UMAArrayDescr) in the fixed length section.

Each of the UDU data segments begins with an ASN.1/BER tag-length prefix. The location of each segment is specified in the UDU header as an offset from a message global start position.

The following figure, UMA Data UDU Message Layout, illustrates the details of how a data UDU is assembled.




```
7fd5cd41      /* UMA UDU Message indicator "UMA" */
83nnnnnn      /* UMA UDU Message length = 0xnnnnnn */
bf0181nn      /* uduHdr tag (0x01) and length (0xnn) */
              <-- Global Start

9f038102      /* uduIndicators tag (0x03) and length (0xnn) */
0000          /* uduIndicators */

bf0581nn      /* uduHdrIntx tag (0x05) and length (0xnn) */

bfd5c200      /* uduUBS Basic Segment Indicator ("UB", counter 0x00) */
83nnnnnn      /* UDU Basic Segment Fixed Section length (0xnnnnnn) */
              <-- Local Start

              /* Note: Variable Length Data Section abbreviated "VLDS" */
bfd5c280      /* uduUBA Basic Segment VLDS indicator ("UB", counter 0x80) */
83nnnnnn      /* UDU Basic Segment VLDS length (0xnnnnnn) */

              /* OPTIONAL SEGMENT (not shown) */
bfd5cf00      /* uduUOS Optional Segment indicator ("UO", counter 0x00) */
83nnnnnn      /* UDU Optional Segment Fixed Section length (0xnnnnnn) */
bfd5cf80      /* uduUOS Optional Segment VLDS indicator ("UO", counter 0x80) */
83nnnnnn      /* UDU Optional Segment VLDS length (0xnnnnnn) */

              /* EXTENSION SEGMENT (not shown) */
bfd5c500      /* uduUES Extension Segment indicator ("UE", counter 0x00) */
83nnnnnn      /* UDU Extension Segment Fixed Section length (0xnnnnnn) */
bfd5c580      /* uduUES Extension Segment VLDS indicator ("UE", counter 0x80) */
83nnnnnn      /* UDU Extension Segment VLDS length (0xnnnnnn) */
```

Figure 6-1 UMA Data UDU Message Layout

6.3.1 UMA API Message Header Formats for Data UDUs

The following table shows the UMA message header format for data UDUs (Canonical C format shown).

Table 6-4 UMA API Data UDU Message Header

Data Type	Name	Description
UMAOctetString[4]	mh_msgtag	BER-encoded UMA message indicator tag 0x7fd5cd41
UMAOctetString[1]	mh_msglenlen	ASN.1/BER length of mh_msglen
	1000 0011	Leading bit always 1; indicates mh_msglen length of 3 octets.
UMAOctetString[3]	mh_msglen	Length of the message including the standard header starting at the next field
UMAOctetString[3]	mh_hdrtag	UDU Header Tag 0xbf0181 (includes length of mh_hdrlen in the low order 7 bits). The first octet of the string marks the global start of the message.
UMAOctetString[1]	mh_hdrlen	Length of UDU Header
UMAMsgFlags	mh_flags	UDU Indicators
	0...	UDU contains interval/event data
	010.	Canonical A format
	001.	Canonical B format
	011.	Canonical C format
	0... 0...	16-bit byte order H to L
	0... 1...	16-bit byte order L to H
	0... .0..	integer component order H to L
	0... .1..	integer component order L to H
	0... 1...	Interval data (interval header extension will be present)
	0... 0...	Event or trace data (event header extension will be present)
	0... ...0	dst not in effect
	0... ...1	dst in effect
	0... .... 1...	First message for this class, this system ID
	0... .... .1..	Last message for this class, this system ID
	0... ....1	Last message for this subclass
	0... .... 1...	Class specified is provider-specific

Data Type	Name	Description
	`0... .... .... .1..`	Subclass specified is provider-specific
	`0... .... .... ...1`	Threshold screening has been applied to this UDU
UMATimeStamp	mh_time	Timestamp of message creation (data received from Data Capture Interface)
UMAClass	mh_class	UMA class of the message
UMASubClass	mh_subclass	UMA subclass of the message
UMAOctetString[8]	mh_address	Host network address that generated the data in the message
UMAOctetString[4]	mh_addr_family	Host network address type (Internet, SNA, etc.)
UMAUint4	mh_provider	Identifier of the data provider that registered to supply this data.
UMAUint4	mh_provinst	The instance of this provider.

6.3.2 Interval Header Extension and Data UDU Basic Segment

The following table shows the interval header extension and basic segment for data UDUs (Canonical C format shown).

Table 6-5 UMA API Interval Header Extension and Data UDU Basic Segment

Data Type	Name	Description
	Interval Data Header Extension	(Type UMAIntExt)
UMAOctetString[3]	mhix_ixlenlen	BER-encoded header interval extension tag and length of mhix_ixlen (0xbf0581)
UMAOctetString[1]	mhix_ixlen	Length of the interval extension
UMAMsgFlags	mhix_flags	Interval extension flags
	`1... .... .... ....`	First message for interval, this system ID
	`.1.. .... .... ....`	Last message for interval, this system ID
	`..1. .... .... ....`	Source for this data was recent history
UMATimeStamp	mhix_schedtime	Datetime measurement scheduled as timestamp (nsec)
UMATimeStamp	mhix_intime	Actual timestamp for this interval (nsec)
UMATimeUsec	mhix_intlen	Duration of this interval in microseconds
UMAUint4	mhix_baseoff	Offset from global start to basic data segment, if it is present (zero otherwise)
UMAUint4	mhix_optoff	Offset from global start to optional segment, if it is present (zero otherwise)
UMAUint4	mhix_extoff	Offset from global start to extension segment, if it is present (zero otherwise)
	Data UDU Basic Segment	
UMAOctetString[4]	bs_segtag	BER-encoded basic data segment tag 0xbfd5c200 ("UB", counter 0x00). This and the next 2 fields are included in the type UMASegDescr.
UMAOctetString[1]	bs_seglenlen	ASN.1/BER length of bs_seglen
	`1000 0011`	Leading bit always 1; indicates bs_seglen length of 3.

Data Type	Name	Description
UMAOctetString[3]	bs_seglen	Length of the basic segment that follows, not including this field or the length of the Variable Length Data Section (VLDS); the next octet marks the local start position for this segment.
	. Basic Segment Data .	
	UDU VLDS	
UMAOctetString[4]	vlds_sectag	ASN.1/BER-encoded VLDS tag 0xbfuutt80 This and the next 2 fields are included in the type UMASegDescr.
	uutt = 0xd5c2	Basic segment
	uutt = 0xd5cf	Optional segment
	uutt = 0xd5c5	Extension segment
UMAOctetString[1]	vlds_seclenlen	ASN.1/BER length of vlds_seclen
	`1000 0011`	Leading bit always 1; gives vlds_seclen length 3 octets.
UMAOctetString[3]	vlds_seclen	VLDS length (not including this field)
	. Variable length data item .	

6.3.3 Event Header Extension and Data UDU Basic Segment

The following table shows the event header extension and basic segment for data UDUs (Canonical C format shown).

Table 6-6 UMA API Event Header Extension and Data UDU Basic Segment

Data Type	Name	Description
	Event Data Header Extension	(Type UMAEvtExt)
UMAOctetString[3]	mhex_exlenlen	BER-encoded event header extension tag and length of mhex_exlen (0xbf0681).
UMAOctetString[1]	mhex_exlen	Length of the event header extension
UMATimeStamp	mhex_evtime	Timestamp of this event (nsec)
UMAUint4	mhex_baseoff	Offset from global start to basic data segment, if it is present (zero otherwise)
UMAUint4	mhex_optoff	Offset from global start to optional segment, if it is present (zero otherwise)
UMAUint4	mhex_extoff	Offset from global start to extension segment, if it is present (zero otherwise)
	Data UDU Basic Segment	
UMAOctetString[4]	bs_segtag	BER-encoded basic data segment tag 0xbfd5c200 ("UB", counter 0x00). This and the next 2 fields are included in the type UMASegDescr.
UMAOctetString[1]	bs_seglenlen `1000 0011`	ASN.1/BER length of bs_seglen Leading bit always 1; indicates bs_seglen length of 3.
UMAOctetString[3]	bs_seglen	Length of the basic segment that follows, not including this field and not including the length of the VLDS; the next octet marks the local start position for this segment.
	. . Basic Segment Data .	

Data Type	Name	Description
	. *UDU VLDS*	
UMAOctetString[4]	vlds_sectag	ASN.1/BER-encoded VLDS tag 0xbfuutt80. This and the next 2 fields are included in the type UMASegDescr.
	uutt = 0xd5c2	Basic segment
	uutt = 0xd5cf	Optional segment
	uutt = 0xd5c5	Extension segment
UMAOctetString[1]	vlds_seclenlen	ASN.1/BER length of vlds_seclen
	`1000 0011`	Leading bit always 1; indicates vlds_seclen length of 3 octets.
UMAOctetString[3]	vlds_seclen	VLDS length (not including this field)
	. . Variable length data item .	

6.3.4 Optional and Extension Segments

The Data Capture Committee *Measurement Data Pool* document defines two data segments in addition to that for basic data. The *optional* segment contains standard data that is part of the pool, but may be implemented at an individual vendor's choice. The extension segment contains additional data items that a specific vendor considers useful.

Data Type	Name	Description
UMAOctetString[4]	os_segtag	BER-encoded optional data segment tag 0xbfd5cf00 ("UO", counter 0x00)
UMAOctetString[1]	os_seglenlen 1000 0011	ASN.1/BER length of os_seglen Leading bit always 1; indicates os_seglen length of 3.
UMAOctetString[3]	os_seglen . . Optional Segment Data . .	Length of the optional segment that follows, not including this field and not including the length of the VLDS; the next octet marks the local start position for this segment.

Table 6-7 Optional Segment Header

Data Type	Name	Description
UMAOctetString[4]	es_segtag	BER-encoded extension data segment tag 0xbfd5c500 ("UE", counter 0x00)
UMAOctetString[1]	es_seglenlen 1000 0011	ASN.1/BER length of es_seglen Leading bit always 1; indicates es_seglen length of 3.
UMAOctetString[3]	es_seglen . . Extension Segment Data . .	Length of the extension segment that follows, not including this field; the next octet marks the local start position for this segment.

Table 6-8 Extension Segment Header

6.3.5 Variable Length Data

In a UDU data segment (basic, optional, extension), a variable length data item is comprised of two parts. The first is a fixed-length descriptor giving the offset to the item from the segment local start position and the length of the item. The second part is the variable length data itself which is located in the segment's VLDS. Variable length text data, described by the UMA type UMATextDescr is an example of the use of this format.

Data Type	Name	Description
	.	
xxxxxx	*pppppp*	Fields preceding descriptor
	.	
UMAUint4	*yyy*_off	Offset from segment local start to variable length data item *yyy*
UMAUint4	*yyy*_len	Length of the variable length data item (in bytes)
	.	
xxxxxx	*ffffff*	Field(s) following variable length data descriptor
	UDU VLDS	
UMAOctetString[4]	vlds_sectag	ASN.1/BER-encoded VLDS tag 0xbfuutt80
	uutt = 0xd5c2	Basic segment
	uutt = 0xd5cf	Optional segment
	uutt = 0xd5c5	Extension segment
UMAOctetString[1]	vlds_seclenlen	ASN.1/BER length of vlds_seclen
	`1000 0011`	Leading bit always 1; indicates vlds_seclen length of 3 octets.
UMAOctetString[3]	vlds_seclen	VLDS length (not including this field)
	.	
	.	
	Variable length data item	
	.	

Table 6-9 Format for Variable Length Data Items

6.3.6 Array Data

In a UMA data message segment (basic, optional, extension), an array data entry is comprised of two parts. The first is a fixed-length descriptor (of type UMAArrayDescr) giving the offset to the array from the segment local start position, the count of array elements, and the size of each. The second part is the array itself which is located in the segment's VLDS.

Data Type	Name	Description
	.	
xxxxxx	*pppppp*	Fields preceding array descriptor
	.	
UMAUint4	*yyy*_off	Offset from segment local start to first element of array *yyy*
UMAUint4	*yyy*_elmtcount	Number of array elements in *yyy*
UMAUint4	*yyy*_elmtsize	Size of each array element in *yyy*
	.	
xxxxxx	*ffffff*	Field(s) following array descriptor
	.	
	.	
	UDU VLDS	
UMAOctetString[4]	vlds_sectag	BER-encoded VLDS tag 0xbfuutt80
	uutt = 0xd5c2	Basic segment
	uutt = 0xd5cf	Optional segment
	uutt = 0xd5c5	Extension segment
UMAOctetString[1]	vlds_seclenlen	ASN.1/BER length of vlds_seclen
	`1000 0011`	Leading bit always 1; indicates vlds_seclen length of 3 octets.
UMAOctetString[3]	vlds_seclen	VLDS length of following (not including this field)
	.	
	Array Data	
	.	
	.	

Table 6-10 Format for Array Data Items

Chapter 7

Distributed UMA

This chapter defines specifications that will allow multiple UMA implementations to interoperate in a distributed environment. The UMA MLI interface supports the semantics of distributed data sources and destinations. Examples include the explicit use of a remote node name for *sysid* in *umaCreate*( ), or references to (implicitly) remote UMADS areas.

The UMA Data Services Layer (DSL) provides this distributed support either directly or indirectly through the use of other service providers, but in any case certain DSL-to-DSL communications must be provided to implement these distributed functions. Ensuring interoperability between DSLs implemented by different vendors and on different hardware architectures will require common messages (logical protocol) and a common message transport. As the DSL is the coordinator of distributed services for UMA, the subsequent discussion will be focused on this functional layer.

For the purposes of this discussion, the *local DSL* is defined as being a UMA Data Services Layer that co-resides with a *requesting* MAP and MLI on a *local host*. A *remote DSL* will mean a responding Data Services Layer on a remote host that is to provide data to the requesting MAP and MLI via the local DSL. The remote host may or may not have MAPs executing, nor does it necessarily have a Data Capture Layer (meaning it may be a UMADS data provider only). The following figure illustrates the relationship:

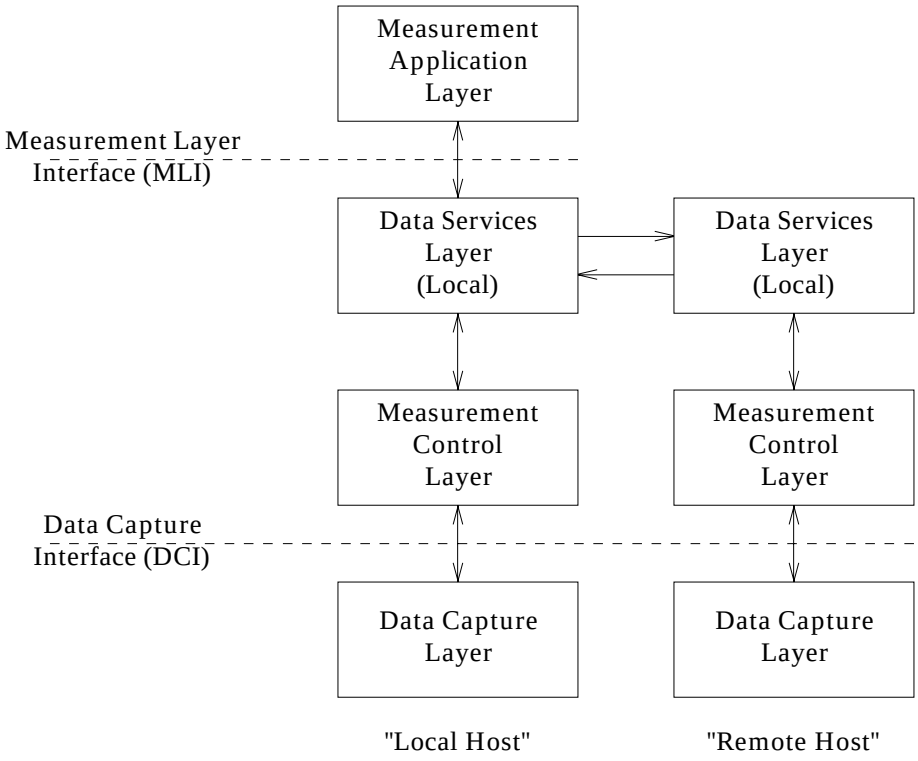


Figure 7-1 Distributed UMA - Host / DSL Relationships

7.1 Message Transport

The default message transport for UMA is TCP/IP with a socket interface[8]. This choice is based on the need for both efficiency and reliability in transporting UMA control and data messages. To fulfill the requirements for connection establishment, and in- and out-of-band message delivery, the following ports are defined:

uma — The service name for a registered port used to establish connections between local and remote Data Service Layers. Once the connection is established, this port is used to deliver command messages, interval data messages, and those event data messages that are to be sent in-band.

The constant UMA_PORT, giving the port number for the service defined, is defined in the header file **<mli.h>**.

uma_high — A port used to deliver status messages and event data messages that are to be sent out-of-band. See the *umaStart*( ) call description in a previous section for a further discussion of in-band and out-of-band reporting. This port is dynamically assigned and is returned in the *Connection ack* and *Reconnection ack* subclass messages defined by the logical protocol. See Section 7.4 on page 88 for further details.

7.1.1 Logical Buffer Sizing

When establishing a connection with a remote DSL, either as a consequence of a *umaCreate()* or a *umaReconnect()* call, a remote DSL uses a message buffer of size indicated by the ‘‘maximum buffer size to be used for communications’’ field in the *Create* or *Reconnect* control messages sent by the local DSL. This ensures compatible message exchanges governed by the socket interface.

7.1.2 Byte Ordering

In order to ensure that a remote DSL correctly interprets control and data messages sent by a local DSL host that has a different byte order data architecture for integers, the UMA message header contains indicator flags in field *mh_flags* to indicate the byte order of the host sending the message. These indicators are interpreted thus:

1. If the indicator for the 16-bit byte order is on (a binary 1), each 16-bit component of an integer quantity has internal byte order from low to high with increasing addresses and its low order byte is coincident with the component starting address (little endian). Otherwise, the component has byte order high to low with its high order byte at the component starting address (big endian).
2. If the indicator for the integer component order is on (a binary 1), the 16-bit components of the integer are ordered from low to high with increasing addresses; the start of the low order 16-bit component is coincident with the integer starting address. Otherwise, the 16-bit components are ordered from high to low with the start of the high order 16-byte component coincident with the integer starting address.

Furthermore, the default byte ordering for all integers or integer-based quantities in the UMA message header (including *mh_flags* itself) is *network standard byte order*. This is mandatory for

8. The OMG is in the process of defining message and transport specifications for CORBA that will ensure that different Object Request Broker (ORB) implementations will interoperate. In future, PMWG will evaluate these specifications when they are finalised, along with others, to determine their applicability for UMA message transport.

the **Create** and **Reconnect** logical messages, and in these instances the byte ordering indicators in *mh_flags* represent the local DSL's preferred byte ordering for header content. In the **Connection Acknowledement**, the DSL server indicates the header byte order it will use for the **Acknowledgement** and for all subsequent messages.

Lastly, UMADS data is always transported to a local DSL with the same byte ordering as for the host on which it originated. This means that a receiving DSL will always see the same byte ordering regardless of the architecture of the platform that may be acting as a data server.

7.2 Message Buffering - Normal Priority Channel

Messages sent on the normal priority channel — uma — may be blocked with other messages in a single buffer for the purposes of efficient transmission.

A token called a *seek number*, a monotonically increasing integer, is included in the buffer. The seek number is increased with each *umaSeek*( ) call to reflect that a change is to take place in the MAP's (and the MLI's) notion of current time for the session. The seek number is provided so that both local and remote data buffers may be resynchronised after the repositioning. This resynchronisation consists of discarding data buffers with lower seek numbers. The normal priority channel buffer format is:

UMAUint4	Byte count of data in buffer in bytes not including this field
UMAUint8	Seek number
	Message 1
	Message 2
	.
	.
	Message n

7.3 Message Buffering - High Priority Channel

Status messages and out-of-band condition messages from a source DSL are sent via the high priority channel, uma_high, and may be blocked with other messages in a single buffer. The buffer format is as follows:

UMAUint4	Byte count of data in buffer in bytes not including this field
	Message 1
	Message 2
	.
	.
	Message n

7.4 Logical Message Protocol

Messages sent from DSL to DSL include both *control* and *data* messages. Control messages include:

1. Requests resulting from calls to an MLI to be acted on by a remote DSL (class *Command*, subclass depends on MLI call, eg. subclass 'Seek')
2. Status messages from a remote DSL for the local DSL (for example, class *Connection Status*)
3. Condition messages from a remote DSL to be forwarded to a local MLI/MAP (class *Condition*, subclass depends on type, for example Informational, Severe)

Data messages from a remote DSL are sent to the local DSL, and depending on the MLI session destination, may either be written to UMADS or private files by the local DSL or may be forwarded to a MAP/MLI.

The general structure of UMA control and data messages has been described in a previous section. The remainder of this section describes specific messages used for DSL-to-DSL communications organised by the types defined above.

The following table summarises the relationship of MLI calls, messages emitted by the MLI's associated DSL, and remote DSLs in a distributed environment.

MLI Call	Local DSL 'Command' Message Request Subclass	Remote DSL Status/Condition/Data Response Subclass	Notes
umaClose( )	Close	-	-
umaCreate( )	Create	Connection Ack	Synchronous response
umaGetAttr( )	-	-	-
umaGetMsg( )	Request Data	Buffer (Data, Condition, Status)	Request to replenish buffer
umaGetReason( )	-	-	-
umaReconnect( )	Reconnect	Reconnect Ack	Synchronous response
umaRelease( )	Release	-	-
umaRequestConfig( )	Request Configuration	-	-
umaSeek( )	Seek	-	-
umaSetAttr( )	Set Attribute	-	One message per attribute
umaSetThreshold( )	Set Threshold	-	MLI may issue prior Request Configuration to verify class/subclass status
umaStart( )	Start	-	MLI may issue prior Request Configuration to verify class/subclass status
umaStop( )	*Stop*	-	-

Table 7-1 MLI Calls and Resulting DSL-to-DSL Messages

Note that condition messages (class *Condition*, subclasses *Informational*, *Warning*, *Severe* or *Fatal*) may be sent to the MLI as a result of any of the above MLI calls. These messages may be originated by either the remote or local DSL.

7.4.1 Forwarded Requests - Message Class - Command

These messages are in class *Command* and result from MLI requests. The subclass depends on the specific action requested.

7.4.1.1 Message Subclass - Create

This message requests that the UMA facility create a session.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAUint4	maximum buffer size to be used for communications.
UMAUint4	user id for remote authorisations.
UMAUint4	remote host I/P address for which data is to be reported.
UMATextDescr	source as specified in MLI *umaCreate*( ) call.
UMATextDescr	destination as specified in MLI *umaCreate*( ) call.
UMAProp	session properties as specified in MLI *umaCreate*( ) call.

7.4.1.2 Message Subclass - Reconnect

This message informs the UMA facility to reestablish session control to a previously shut down session with the *NOTERM* property.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAUint4	maximum buffer size to be used for communications.
UMATextDescr	destination as specified in MLI *umaReconnect*( ) call.

7.4.1.3 Message Subclass - Set Attribute

This message informs the UMA facility of the session attributes. One attribute is sent per message.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMATextDescr	attribute name. The following attribute strings are defined: "STIME", "ETIME", "INTERVAL", "PRIO", "HISTORY_ONLY", "RECENT_ONLY", "PARTIAL".
UMAInt4	attribute value.

7.4.1.4 Message Subclass - Close

This is a request to shut down a session. This message is composed of a header only. The subclass field in the header identifies the request.

UMAHeader	standard UMA message header.

7.4.1.5 Message Subclass - Start

This message informs the UMA facility to report data specified by the fields in this subclass (that is, class, subclass, . . .)

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1 / BER tags and lengths for the segment.
UMAProvider	data provider as specified in the *umaStart*( ) MLI call.
UMAClass	data class as specified in the *umaStart*( ) MLI call.
UMASubClass	data subclass as specified in the *umaStart*( ) MLI call.
UMASegFlags	data segments as specified in the *umaStart*( ) MLI call. The value will be one of UMA_BSEG, UMA_OSEG, UMA_ESEG, UMA_ASEG representing basic, optional, vendor extension, and all, respectively.
UMAUint4	length of workload definition structure or zero, if none specified.
UMAWorkDefn	workload definition structure as specified in the *umaStart*( ) call. This parameter is optional and will not be present if the preceding sentinel length indicator is zero.

7.4.1.6 Message Subclass - Set Threshold

This message informs the UMA facility to report data specified by the fields in this subclass (that is, class, subclass, . . .) only if the selection expression evaluates to true.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1 / BER tags and lengths for the segment.
UMAProvider	data provider as specified in the *umaSetThreshold*( ) MLI call.
UMAClass	data class as specified in the *umaSetThreshold*( ) MLI call.
UMASubClass	data subclass as specified in the *umaSetThreshold*( ) MLI call.
UMASegFlags	data segments as specified in the *umaSetThreshold*( ) MLI call. The value will be one of UMA_BSEG, UMA_OSEG, UMA_ESEG, UMA_ASEG representing basic, optional, vendor extension, and all, respectively.
UMATextDescr	selection expression as specified in the *umaSetThreshold*( ) MLI call. See the description for the *umaSetThreshold*( ) MLI call for expression syntax.
UMATextDescr	workload identifier as specified in the *umaSetThreshold*( ) call.

7.4.1.7 Message Subclass - Release

This message informs the UMA facility to begin the collection and reporting of started data classes and subclasses. This message is composed of a header only. The subclass field in the header identifies the request.

UMAHeader	standard UMA message header.

7.4.1.8 Message Subclass - Request Data

This message informs the UMA facility that a receiving buffer is empty and therefore able to receive more data.

UMAHeader	standard UMA message header
UMAUint4	buffer selection. The value will be one of UMA_HIGH_BUFF, UMA_NORM_BUFF, representing high and normal buffers, respectively.

7.4.1.9 Message Subclass - Stop

This message informs the UMA facility to stop reporting data specified by the fields in this subclass (that is, class, subclass, . . .)

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAProvider	data provider as specified in the *umaStop*( ) MLI call.
UMAClass	data class as specified in the *umaStop*( ) MLI call.
UMASubClass	data subclass as specified in the *umaStop*( ) MLI call.
UMASegFlags	data segments as specified in the *umaStop*( ) MLI call. The value will be one of UMA_BSEG, UMA_OSEG, UMA_ESEG, UMA_ASEG representing basic, optional, vendor extension, and all, respectively.
UMAFlushFlags	queue of data to flush as specified in the *umaStop*( ) MLI call. The value will be one of UMA_HELD, UMA_RELEASED, UMA_ALLSTARTED, to indicate whether *held* requests or only those *released* or all started requests are to be stopped.
UMAUint4	length of workload definition structure or zero, if none specified.
UMAWorkDefn	workload definition structure as specified in the *umaStop*( ) call. This parameter is optional and will not be present if the preceding sentinel length indicator is zero.

7.4.1.10 Message Subclass - Seek

This message informs the UMA facility to reposition the next reported interval.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAWhence	whence value: one of UMA_CTIME, UMA_LTIME, UMA_STIME or UMA_TSTAMP. Indicates what the seek is relative to.
UMATimeSpec	if the *whence* is UMA_TSTAMP, this field contains an arbitrary timestamp value which the seek is relative to.
UMAInt4	relative position (number of intervals of data messages) from timestamp as specified in the *umaSeek*( ) MLI call. The value may be zero, positive, or negative.

UMAUint4	port number for high priority channel (uma_high).
UMATimeStamp	session's current time (as known by the local DSL).
UMAUint4	session's last seek number (as known by the local DSL).

7.4.1.11 Message Subclass - Request Configuration

This message requests that the UMA facility return (as messages) the configuration status of classes and subclasses specified in the fields.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1 / BER tags and lengths for the segment.
UMAProvider	data provider as specified in the *umaRequestConfig*( ) MLI call.
UMAClass	configuration data class as specified in the *umaRequestConfig*( ) MLI call.
UMASubClass	configuration data subclass as specified in the *umaRequestConfig*( ) MLI call.
UMAUint4	port number for high priority channel (uma_high).

7.4.2 Message Class - Connection Status

7.4.2.1 Message Subclass - Connection Ack

This message is a status acknowledgment for a connection request.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAStatusCode	status code. This code identifies the general outcome of the connection request. Defined values are specified elsewhere in this document.
UMAReasonCode	reason code. This code provides more in depth identification for a status code. Defined values are specified elsewhere in this document.

7.4.2.2 Message Subclass - Reconnect Ack

This message is a status acknowledgment for a reconnection request.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAStatusCode	status code. This code identifies the general outcome of the connection request. Defined values are specified elsewhere in this document.
UMAReasonCode	reason code. This code provides more in depth identification for a status code. Defined values are specified elsewhere in this document.
UMATextDescr	source.
UMAProp	session properties.
UMAAttr	session attributes.

7.4.3 Message Class - Condition

7.4.3.1 Message Subclass - Informational

This message provides informational status regarding a condition that has occurred in UMA.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAInt4	source of the condition.
UMAInt4	condition identifier.
char[]	condition description.
UMAInt4	type of hint1.
UMAHint	value of hint1.
UMAInt4	type of hint2.
UMAHint	value of hint2.
UMAInt4	type of hint3.
UMAHint	value of hint3.

7.4.3.2 Message Subclass - Warning

This message provides warning status regarding a condition that has occurred in UMA.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAInt4	source of the condition.
UMAInt4	condition identifier.
char[]	condition description.
UMAInt4	type of hint1.
UMAHint	value of hint1.
UMAInt4	type of hint2.
UMAHint	value of hint2.
UMAInt4	type of hint3.
UMAHint	value of hint3.

7.4.3.3 Message Subclass - Severe

This message provides status regarding a severe condition that has occurred in UMA.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAInt4	source of the condition.
UMAInt4	condition identifier.

char[]	condition description.
UMAInt4	type of hint1.
UMAHint	value of hint1.
UMAInt4	type of hint2.
UMAHint	value of hint2.
UMAInt4	type of hint3.
UMAHint	value of hint3.

7.4.3.4 Message Subclass - Fatal

This message provides status regarding a fatal condition that has occurred or is imminent in UMA.

UMAHeader	standard UMA message header.
UMASegDescr	ASN.1/BER tags and lengths for the segment.
UMAInt4	source of the condition.
UMAInt4	condition identifier.
char[]	condition description.
UMAInt4	type of hint1.
UMAHint	value of hint1.
UMAInt4	type of hint2.
UMAHint	value of hint2.
UMAInt4	type of hint3.
UMAHint	value of hint3.

Chapter 8

The UMA Configuration Class

This chapter describes the ‘‘UMA Configuration’’ class and its subclasses. Messages in this class give data pertaining to the specific UMA implementation, data providers, and to the UMA configuration parameters on this particular system.

This class is an MLI-only class, created from the examination of the name space.

8.1 Subclass - System Description

A message of the subclass is produced on demand and when a change occurs in the parameters reflected in its content.

8.2 MLI Subclass Information

(H)UMAHeader	Standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAOctetString[8]	name of the operating system that created this message.
(B)UMAOctetString[8]	version of the operating system that created this message.
(B)UMAOctetString[8]	release of the operating system that created this message.
(B)UMAOctetString[8]	version of UMA that created this message.
(B)UMAMsgFlags	time representation indicators

```
.... .... .... 0000   duration is UMATimeStamp
.... .... .... 0001   duration is UMATimeVal
.... .... .... 0010   duration is UMATimeSpec
.... .... 0000 ....   timestamp is UMATimeStamp
.... .... 0001 ....   timestamp is UMATimeVal
.... .... 0010 ....   timestamp is UMATimeSpec
```

(B)UMATextDescr	descriptor for the standard time designator as defined for TZ in the System Interface Definitions (see reference **XSH**).
(B)UMATextDescr	descriptor for the standard time offset from UTC [+/-]hh[:mm[:ss]] as defined for TZ in the System Interface Definitions (see reference **XSH**). time offset is the value to be added to local time to arrive at UTC (Universal Coordinated Time).
(B)UMATextDescr	descriptor for the alternative time designator as defined for TZ in the System Interface Definitions (see reference **XSH**).
(B)UMATextDescr	descriptor for the alternative time offset from UTC [+/-]hh[:mm[:ss]] as defined for TZ in the System Interface Definitions (see reference **XSH**).
(B)UMATextDescr	descriptor for the rule to change to and from alternative time, date[/time],date[/time,] as defined for TZ in the System Interface Definitions (see reference **XSH**). Each time field describes when, in current local time, the change to the other time is made.

8.3 Subclass - UMA Providers

A message of the subclass is produced on demand and, if requested, when a change occurs in the parameters reflected in its content. It describes what providers are available.

8.3.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAArrayDescr	descriptor for the array of provider integer ids.
(B)UMAVarArrayDescr	descriptor for the array of provider labels of type UMALabel.

8.4 Subclass - UMA Work Units

A message of the subclass is produced on demand and, if requested, when a change occurs in the parameter reflected in its content. It defines the possible UMA Work Units that can be in effect with per-work-unit data subclasses.

8.4.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAVarArrayDescr	descriptor for the array of UMAWorkDescr structures.

8.5 Subclass - Implementation

A message of the subclass is produced on demand and, if requested, when a change occurs in the parameter reflected in its content. It describes which classes, subclasses and fields are implemented.

8.5.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAUint4	the highest class id implemented.
(B)UMAArrayDescr	descriptor for the class-status array.
(B)UMAArrayDescr	descriptor for the class-subclass count array.
(B)UMAArrayDescr	descriptor for the class-subclass index array.
(B)UMAArrayDescr	descriptor for the basic subclass status array.
(B)UMAArrayDescr	descriptor for the optional subclass status array.
(B)UMAArrayDescr	descriptor for the extension subclass status array.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic segment variable length data section.
(B)UMAUint4[]	the class-status array. This array, when indexed by the class id shows the implementation status of a given class. The value of each array element will be one of NOTIMPLEMENTED, DISABLED, or ENABLED. NOTIMPLEMENTED has the obvious meaning, DISABLED means that this class is implemented, but was omitted from the system at configuration time or disabled dynamically, ENABLED means that this class is available to be collected.
(B)UMAUint4[]	the class-subclass count array. This array, when indexed by class id gives the count of subclass ids implemented for this class.
(B)UMAUint4[]	the class-subclass index array. This array, when indexed by class id gives the index of the first subclass of this class in the subclass status array.
(B)UMAUint4[]	the subclass basic status array. This array, when indexed by the class-subclass index plus the subclass id gives the implementation status of basic segment of this subclass. The possible values are the same as in the class status array.
(B)UMAUint4[]	the subclass optional status array. This array, when indexed by the class-subclass index plus the subclass id gives the implementation status of optional segment of this subclass. The possible values are the same as in the class status array.
(B)UMAUint4[]	the subclass extension status array. This array, when indexed by the class-subclass index plus the subclass id gives the implementation status of extension segment of this subclass. The possible values are the same as in the class status array.
(B)UMAUint4[]	the subclass-field count array. This array, when indexed by class-subclass index gives the count of field sequence ids implemented for this subclass.

(B)UMAUint4[]	the subclass-field index array. This array, when indexed by class-subclass index gives the index of the first field of this subclass in the field status array.
(B)UMAUint4[]	the field basic status array. This array, when indexed by the subclass-field index plus the field sequence id gives the implementation status of this field in the basic segment. The possible values are the same as in the class status array.
(B)UMAUint4[]	the subclass optional status array. This array, when indexed by the subclass-field index plus the field sequence id gives the implementation status of this field in the optional segment. The possible values are the same as in the class status array.
(B)UMAUint4[]	the subclass extension status array. This array, when indexed by the subclass-field index plus the field sequence id gives the implementation status of this field in the extension segment. The possible values are the same as in the class status array.

8.6 Subclass - States

A message of the subclass is produced on demand and describes the current state of each segment type of a given class/subclass.

8.6.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAArrayDescr	descriptor for the class-subclass basic state array.
(B)UMAArrayDescr	descriptor for the class-subclass optional state array.
(B)UMAArrayDescr	descriptor for the class-subclass extension state array.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic segment variable length data section.
(B)UMAUint4[]	the class-subclass basic state array. This array, when indexed by the class-subclass index plus the subclass id shows the session's state of a given class/subclass basic segment. The value of each array element will be one of NOTSTARTED, RELEASED, and HELD. NOTSTARTED and RELEASED have the obvious meaning. HELD means that the session has started the specified class/subclass, and has not yet issued a *umaRelease*( ).
(B)UMAUint4[]	the class-subclass optional state array. This array, when indexed by the class-subclass index plus the subclass id shows the session's state of a given class/subclass optional segment. The value of each array element will be one of NOTSTARTED, RELEASED, and HELD. NOTSTARTED and RELEASED have the obvious meaning. HELD means that the session has started the specified class/subclass, and has not yet issued a *umaRelease*( ).
(B)UMAUint4[]	the class-subclass extension state array. This array, when indexed by the class-subclass index plus the subclass id shows the session's state of a given class/subclass extension segment. The value of each array element will be one of NOTSTARTED, RELEASED, and HELD. NOTSTARTED and RELEASED have the obvious meaning. HELD means that the session has started the specified class/subclass, and has not yet issued a *umaRelease*( ).

8.7 Subclass - Names

A message of the subclass is produced on demand and describes the class name, the subclass name and the abbreviated subclass name.

8.7.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event data header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMAArrayDescr	descriptor for the class name array.
(B)UMAArrayDescr	descriptor for the subclass name array.
(B)UMAArrayDescr	descriptor for the subclass abbreviated name array.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic segment variable length data section.
(B)char[][]	the class name array. This array, when indexed by the class-subclass index plus the subclass id gives the class name.
(B)char[][]	the subclass name array. This array, when indexed by the class-subclass index plus the subclass id gives the subclass name.
(B)char[][]	the subclass abbreviated name array. This array, when indexed by the class-subclass index plus the subclass id gives the abbreviated subclass name.

8.8 Subclass - UMA Restart

This subclass specifies the last UMA restart. In this context, UMA refers to an active data services entity capable of servicing MAPs and able to start data collection as needed.

8.8.1 MLI Subclass Information

(H)UMAHeader	standard UMA data header.
(H)UMAEvtExt	event header extension.
(B)UMASegDescr	ASN.1/BER tags and lengths for the basic data segment.
(B)UMATimeStamp	UMA restart time
(1)UMASegDescr	ASN.1 tags and lengths for the basic segment variable length data section.

Appendix A

C Language Header Files

The following two sections present the **<mli.h>** and **<uma.h>**, header files.

These two header files are included because they define the data types and structures more exactly than English text.

A.1 <mli.h>

```
#ifndef MLI_H
#define MLI_H

#include <limits.h>
#include <sys/uma.h>

/*
 ******************** NOTICE **********************************
 * The <dci.h>, <mli.h> and <uma.h> header files
 * introduce UMA symbols which may conflict with other
 * symbols defined in an application.  Symbols with the
 * following prefixes are therefore reserved to UMA:
 *           DCI
 *           dci
 *           UMA
 *           UMR
 *           UMS
 *
 * Note that the header files are provided as advisory
 * reference examples.
 ******************** END OF NOTICE ***************************
 */

/*********************************************************************/
/*                            UMA data types                         */
/*********************************************************************/
typedef UMAInt4         UMAEvent;
typedef UMAUint4        UMAFlushFlags;
typedef UMAInt4         UMAPrio;
typedef UMAInt4         UMAProp;
typedef UMAInt4         UMAReasonCode;
typedef UMAUint4        UMASegFlags;
typedef UMAInt4         UMASessId;
typedef UMAInt4         UMAStatusCode;
typedef UMAInt4         UMAChannelFlags;
typedef UMAInt4         UMAWhence;

/*********************************************************************/
/*               UMA Workload Definition Structures                  */
/*********************************************************************/

/* The UMAWorkDefn structure provides the definition of a constructed
 *  workload and specification:
 */

typedef struct UMAWorkDefn {
    UMAUint4          size;          /* size of this struct     */
    UMAUint4          rFlags;        /* reporting flags         */
    UMATextDescr      workIdSpec;    /* workload id offset/size */
    UMAUint4          granularity;   /* granularity request     */
    UMAUint4          wFlags;        /* workload flags          */
    UMAVarLenDescr    workInfoSpec   /* offset to WorkInfo data */
    UMAVarLenDescr    instanceSpec   /* offset to instance data */
    UMAVarLenData     data;          /* WorkInfo, instance specs*/
} UMAWorkDefn;
```

```
/* The UMAWorkSpec structure for specifying selection of
 *  UMAWorkInfo values:
 */

typedef struct UMAWorkSpec {
    UMUint4           wSpecSize;  /* size of this structure     */
    UMAUint4          wSelect;    /* WorkInfo level select bits */
    UMAVarArrayDescr  wSpecDescr; /* WorkInfo specs descr       */
    UMAVarLenData     data;       /* WorkInfo level spec value  */
} UMAWorkSpec;

/* And each WorkInfo level specification is in a UMAWorkLvlSpec
 *  structure:
 */

typedef struct UMAWorkLvlSpec {
    UMAUint4        wLvlSize;    /* size of this structure    */
    UMAUint4        wLvlType;    /* enum type of WorkInfo spec*/
                                 /* UMA_TEXTSTRING,           */
                                 /* UMA_UINT4, etc assigned   */
    UMAVarLenData   data;        /* work spec/expr            */
} UMAWorkLvlSpec;

/* The UMAInstSpec structure for specifying selection of instances is
 * defined:
 */

typedef struct UMAInstSpec {
    UMAUint4          iSpecSize;   /* size of this structure     */
    UMAUint4          iSelect;     /* instance level select bits */
    UMAVarArrayDescr  iSpecsDescr; /* instance level specs descr */
    UMAVarLenData     data;        /* instance level spec values */
} UMAInstSpec;

/* And each instance level specification is defined using the
 *  UMAInstLvlSpec structure:
 */

typedef struct UMAInstLvlSpec {
    UMAUint4        iLvlSize;    /* size of this structure    */
    UMAUint4        iLvlType;    /* enum type of instance spec*/
                                 /* UMA_TEXTSTRING,           */
                                 /* UMA_UINT4, etc assigned   */
    UMAVarLenData   data;        /* instance spec/expr  */
} UMAInstLvlSpec;

/*****************************************************************/
/*                Workload Reporting Flags                       */
/*****************************************************************/
#define UMA_REPORT_DYNAMIC    1<<0
#define UMA_REPORT_EVENT      1<<1
#define UMA_REPORT_WORKLOAD   1<<2
#define UMA_EVENT_FINALDATA   1<<3
#define UMA_EVENT_INBAND      1<<4

/*****************************************************************/
/*              Dynamic and builtin data availability            */
/*****************************************************************/
#define UMA_BUILTIN   0x0001   /* builtin data available */
```

```
#define UMA_DYNAMIC   0x0002   /* dynamic data available */

/*********************************************************************/
/*             UMA Configuration Description Structures              */
/*          Returned in Messages of Class UMA Configuration          */
/*********************************************************************/

/* The UMAClassAttr structure is returned in the message of subclass
 * "Class Attributes"  It provides the i18n, and ascii name labels for
 * a class, identifies the metric-containing subclasses available in
 * the class and provides labels for them.
 */

typedef struct UMAClassAttr {
    UMAClassId       class;          /* class handle             */
    UMAVarLenDescr   classLabel;     /* class label struct       */
    UMAArrayDescr    subClassId;     /* array of DPD subcl ids   */
    UMAArrayDescr    subClassStatus; /* subclass status array    */
    UMAVarArrayDescr subClassLabel;  /* subcl label struct array */
} UMAClassAttr;

/* The UMALabel structure is used to provide both ascii and i18n
 *  labels for UMA providers, subclasses, instance tags, work units,
 *  and for data elements.
 */

typedef struct UMALabel {
    UMAUint4         size;   /* size of this structure        */
    UMAVarLenDescr   ascii;  /* descriptor for the variable   */
                             /* UMATextString for ascii label */
    UMAElementDescr  i18n;   /* descriptor for the variable   */
                             /* length data for i18n label    */
    UMAVarLenData    data;   /* label data for ascii and i18n */
} UMALabel;

/* The UmaSubClassAttr structure is the container for the metadata
 * describing a UMA subclass.  It includes descriptors for instance
 * tags, work unit identifiers and metrics.  It is sent in a message
 * of subclass "Subclass Attributes".
 */

typedef struct UMASubClassAttr {
    UMASubClassHandles handles;      /* cl/subcl handles,flags    */
    UMAVarArrayDescr  instanceTags;  /* instance tag              */
                                     /* descriptor array          */
    UMAVarArrayDescr  workUnits;     /* work unit descr array     */
    UMAVarArrayDescr  dataBasic;     /* basic data desc array     */
    UMAVarArrayDescr  dataOptional;  /* optional data descr array */
    UMAVarArrayDescr  dataExtended;  /* extended data descr array */
} UMASubClassAttr;

/* The UMA class and subclass handles are contained in the
 * UMASubClassHandles structure in messages of subclass
 * ‘‘Subclass Attributes’’.
 */

typedef struct UMASubClassHandles {
    UMAClass     class;          /* UMA class handle      */
    UMASubClass  subClass;       /* UMA subclass handle   */
```

```
} UMASubClassHandles;

/* Metric instance identifiers are mapped level-by-level to
 *  instance tags in the UMA UDU and this mapping is described by
 *  the set of UMAInstTagDescr structures, one per level.
 */

typedef struct UMAInstTagDescr {
    UMAUint4        size;   /* size of this struct    */
    UMAUint4        flags;  /* indic mapped explicit, */
                            /* as data array indices  */
                            /* (lowest level only)    */
    UMADataType     type;   /* instance tag data type */
    UMAInstTagType  itType; /* instance tag type      */
    UMAUint4        itSize; /* tag size in bytes      */
    UMALabel        label;  /* ascii and i18n label   */
} UMAInstTagDescr;

/* The possible work unit identifiers supplied by a provider
 * are defined by a set of UMAWorkDescr structures.  This set
 * of structures is contained in a message of subclass
 * ''UMA Work Units''.
 */

typedef struct UMAWorkDescr {
    UMAUint4        size;   /* size of this struct       */
    UMADataType     dType   /* Work Unit data type       */
    UMAWorkType     wType;  /* Work Unit type            */
    UMALabel        label;  /* ascii and i18n labels     */
} UMAWorkDescr;

/* The mapping of Data Pool metric values to the UMA Data UDU is
 * described by the set of UMADataAttr structures.
 * Flag indicators show the implementation status of an item
 * (NOTIMPLEMENTED, ENABLED, or DISABLED) and whether the data is
 * for the interval or is an absolute count.  The UMADataAttr
 * are contained in messages of subclass ''Subclass Attributes''.
 */

typedef struct UMADataAttr {
    UMAUint4        size;        /* size of this struct     */
    UMADataType     dataType;    /* data type of metric     */
    UMAUint4        status;      /* status: NOTIMPLEMENTED, */
                                 /* DISABLED, ENABLED       */
    UMAUnit         units;       /* data units              */
    UMAUint4        dataFlags;   /* flags on units          */
                                 /* rates, counts,          */
                                 /* intervalization         */
    UMAUint4        offset;      /* to data item or descr   */
                                 /* from segment start      */
    UMADescrType    descrType    /* Descriptor type         */
                                 /* (or none)               */
    UMALabel        label;       /* ascii and i18n labels   */
} UMADataAttr;

/*****************************************************************/
/*                    UMA control classes                        */
/*****************************************************************/
#Define UMA_COMMAND     1
```

```
#define UMA_CONDITION   2
#define UMA_STATUS      3

/*********************************************************************/
/*              UMA message command subclasses                       */
/*********************************************************************/
#define UMA_CREATE      0
#define UMA_RECONNECT   1
#define UMA_SETATTR     2
#define UMA_CLOSE       3
#define UMA_START       4
#define UMA_SETTHRESH   5
#define UMA_RELEASE     6
#define UMA_GETDATA     7
#define UMA_STOP        8
#define UMA_SEEK        9
#define UMA_REQCONFIG   10

/*********************************************************************/
/*              UMA message status subclasses                        */
/*********************************************************************/
#define UMA_CONN_ACK    1
#define UMA_RECONN_ACK  2

/*********************************************************************/
/*               UMA message command segment types                   */
/*********************************************************************/
typedef struct  UMACreate {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMAUint4        cs_buff_size;   /* Maximum comm. buffer size  */
    UMAUint4        cs_uid;         /* For permission validation  */
    UMATextDescr    cs_src;         /* source of the data         */
    UMATextDescr    cs_dest;        /* destination of data        */
    UMAProp         cs_props;       /* Session properties         */
} UMACreate;

typedef struct  UMAReconnect {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMAUint4        cs_buff_size;   /* Maximum comm. buffer size  */
    UMATextDescr    cs_dest;        /* destination of data        */
} UMAReconnect;

typedef struct  UMASetAttr {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMATextDescr    cs_name;        /* Attribute name             */
    UMAInt4         cs_value;       /* Attribute value            */
} UMASetAttr;

typedef struct  UMAClose {
    UMAHeader       cs_header;      /* Message header             */
} UMAClose;

typedef struct  UMAStart {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMAProvider     cs_provider;    /* data provider identifier   */
```

```
    UMAClass        cs_class;       /* data class                */
    UMASubClass     cs_subclass;    /* data subclass             */
    UMASegFlags     cs_flags;       /* segment flags             */
    UMAUint4        cs_sentinel;    /* len of wkld defn struct   */
    UMAWorkDefn     cs_wklddefn;    /* workload defn struct      */
} UMAStart;

typedef struct  UMASetThreshold {
    UMAHeader       cs_header;      /* Message header            */
    UMASegDescr     cs_segdescr;    /* Segment descriptor        */
    UMAProvider     cs_provider;    /* data provider identifier  */
    UMAClass        cs_class;       /* data class                */
    UMASubClass     cs_subclass;    /* data subclass             */
    UMASegFlags     cs_flags;       /* segment flags             */
    UMATextDescr    cs_expression;  /* selection expression      */
    UMATextDescr    cs_wkldId;      /* workload identifier       */
} UMASetThreshold;

typedef struct  UMARelease {
    UMAHeader       cs_header;      /* Message header            */
} UMARelease;

typedef struct  UMARequestData {
    UMAHeader       cs_header;      /* Message header            */
    UMAUint4        cs_buffer;      /* buffer selection          */
} UMARequestData;

typedef struct  UMAStop {
    UMAHeader       cs_header;      /* Message header            */
    UMASegDescr     cs_segdescr;    /* Segment descriptor        */
    UMAProvider     cs_provider;    /* data provider identifier  */
    UMAClass        cs_class;       /* data class                */
    UMASubClass     cs_subclass;    /* data subclass             */
    UMASegFlags     cs_flags;       /* segment flags             */
    UMAFlushFlags   cs_flush;       /* which Q to flush          */
    UMAUint4        cs_sentinel;    /* len of wkld defn struct   */
    UMAWorkDefn     cs_wklddefn;    /* workload defn struct      */
} UMAStop;

typedef struct  UMASeek {
    UMAHeader       cs_header;      /* Message header            */
    UMASegDescr     cs_segdescr;    /* Segment descriptor        */
    UMATimeStamp    cs_tstamp;      /* Search timestamp          */
    UMAInt4         cs_pos;         /* relative position         */
    UMATimeStamp    cs_currtime;    /* session's current time    */
    UMAUint4        cs_lastseek;    /* last seek number          */
} UMASeek;

typedef struct  UMARequestConfig {
    UMAHeader       cs_header;      /* Message header            */
    UMASegDescr     cs_segdescr;    /* Segment descriptor        */
    UMAProvider     cs_provider;    /* data provider identifier  */
    UMAClass        cs_class;       /* data class                */
    UMASubClass     cs_subclass;    /* data subclass             */
} UMARequestConfig;

/*****************************************************************/
/*               UMA message status segment types                */
/*****************************************************************/
```

```
typedef struct  UMAConnectAck {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMAStatusCode   cs_status;      /* Status of session creation */
    UMAReasonCode   cs_reason;      /* Reason of the failure      */
} UMAConnectAck;

typedef struct  UMAReconnectAck {
    UMAHeader       cs_header;      /* Message header             */
    UMASegDescr     cs_segdescr;    /* Segment descriptor         */
    UMAStatusCode   cs_status;      /* Status of reconnection     */
    UMAReasonCode   cs_reason;      /* Reason of the failure      */
} UMAReconnectAck;

/*********************************************************************/
/*                 UMA message condition                             */
/*********************************************************************/
typedef struct  UMACondition {
    UMAHeader       cs_header;      /* header                     */
    UMASegDescr     cs_segdescr;    /* segment descriptor         */
    UMAInt4         cs_source;      /* Source of the condition    */
    UMAInt4         cs_cond_id;     /* Condition identification   */
    UMATextDescr    cs_cond_descr   /* Condition description      */
    UMAInt4         cs_hint1_type;  /* Type of cs_hint1           */
    union  cs_hint1  {
        UMAInt4    hint1_int4;
        UMAInt8    hint1_int8;
        UMATimeSec hint1_timesec;
        UMATimeUsec hint1_timeusec;
        UMATimeNsec hint1_timensec;
        UMATimeStamp hint1_timestamp;
    } cs_hint1;
    UMAInt4 cs_hint2_type;          /* Type of cs_hint2           */
    union  cs_hint2 {
        UMAInt4    hint2_int4;
        UMAInt8    hint2_int8;
        UMATimeSec hint2_timesec;
        UMATimeUsec hint2_timeusec;
        UMATimeNsec hint2_timensec;
        UMATimeStamp hint2_timestamp;
    } cs_hint2;
    UMAInt4 cs_hint3_type;          /* Type of cs_hint3           */
    union  cs_hint3 {
        UMAInt4    hint3_int4;
        UMAInt8    hint3_int8;
        UMATimeSec hint3_timesec;
        UMATimeUsec hint3_timeusec;
        UMATimeNsec hint3_timensec;
        UMATimeStamp hint3_timestamp;
    } cs_hint3;
} UMA_Condition;

/*********************************************************************/
/*             UMA message condition subclasses                      */
/*********************************************************************/
#define UMA_INFO        1
#define UMA_WARNING     2
#define UMA_SEVERE      3
#define UMA_FATAL       4
```

```
/*********************************************************************/
/*          UMA message condition codes for source                   */
/*********************************************************************/
#define DCL             1
#define DSL             2
#define UMADS           3
#define RECENT_HIST     4

/*********************************************************************/
/*             UMA Condition Message Identifiers                     */
/*********************************************************************/
#define UMA_GAP           -1  /* Timestamp in a gap in recent history */
#define UMA_NODATA        -2  /* Requested segments not in current    */
                              /* interval                             */
#define UMA_EOS           -3  /* End time has been reached for this   */
                              /* session                              */
#define UMA_EOF           -4  /* End of source file has been reached  */
#define UMA_INTVL         -5  /* Requested interval is not available  */
#define UMA_OPEN          -6  /* Error opening a source file          */
#define UMA_FLOCK         -7  /* Source file is locked by another     */
                              /* session                              */
#define UMA_PERMISSION    -8  /* Permission error opening a source    */
                              /* file                                 */
#define UMA_FILE_TYPE     -9  /* Source file is of invalid file type  */
#define UMA_BAD_FILE     -10  /* Source file is corrupted             */
#define UMA_READ         -11  /* Error reading a source file          */
#define UMA_WRITE        -12  /* Error writing to a source file       */
#define UMA_ALLOC        -13  /* Error while allocating memory        */
#define UMA_STIME_BOUNDS -14  /* Seek to time prior to session start  */
                              /* time                                 */

/*********************************************************************/
/*         UMA message condition codes for hint types                */
/*********************************************************************/
#define UMA_HINT_INT4       1
#define UMA_HINT_INT8       2
#define UMA_HINT_TIMESEC    3
#define UMA_HINT_TIMEUSEC   4
#define UMA_HINT_TIMENSEC   5
#define UMA_HINT_TIMESTAMP  6

/*********************************************************************/
/*                  Miscellaneous constants                          */
/*********************************************************************/
#ifdef  NULL
#undef  NULL
#endif
#define NULL              0

#ifdef  TRUE
#undef  TRUE
#endif
#define TRUE              1

#ifdef  FALSE
#undef  FALSE
#endif
#define FALSE             0
```

```
#define UMA_ANYSESSION         INT_MAX
#define UMA_ALLSESSIONS        INT_MAX
#define UMA_ALLCLASSES         INT_MAX
#define UMA_ALLSUBCLASSES      INT_MAX
#define UMA_TIME_MAX           INT_MAX
#define UMA_TIME_MIN           INT_MIN
#define UMA_NULLSESSION        -1
#define UMA_TIME_NOW           -999
#define UMA_ANY_BAND           0x0001
#define UMA_IN_BAND_ONLY       0x0002
#define UMA_OUT_OF_BAND_ONLY   0x0004

#define UMA_PORT               1797     /* Port number for UMA service */

/*************************************************************************/
/*                         UMA session flags                             */
/*          The following flags override the default (opposite)          */
/*************************************************************************/
#define UMA_NOTERM         0x0001
#define UMA_NOTREGULAR     0x0002
#define UMA_SYNCH          0x0008
#define UMA_EVENT          0x0010
#define UMA_COHERENT       0x0020

/*************************************************************************/
/*                       UMA seek tstamp                                 */
/*************************************************************************/
#define UMA_STIME          -1
#define UMA_CTIME          -2
#define UMA_LTIME          -3
#define UMA_TSTAMP         -4

/*************************************************************************/
/*                         UMA flush flags                               */
/*************************************************************************/
#define UMA_ALLSTARTED      1
#define UMA_HELD            2
#define UMA_RELEASED        3

/*************************************************************************/
/*                  UMA Class and Subclass Status                        */
/*************************************************************************/
#define UMA_ENABLED          1      /* Class/Subclass available        */
#define UMA_DISABLED         2      /* Class/Subclass implemented,     */
                                    /* but omitted from the system     */
                                    /* at configuration time           */
#define UMA_NOTIMPLEMENTED   3      /* Class/Subclass not implemented  */

/*************************************************************************/
/*                       Network Family                                  */
/*************************************************************************/
#define UMA_INET             1      /*  Internet Protocol              */

/*************************************************************************/
/*                      UMA macro operations                             */
/*************************************************************************/
#define UMA_SET(FIELD, FLAG)    ((FIELD) |=  (FLAG))
#define UMA_RESET(FIELD, FLAG)  ((FIELD) &=  (FLAG))
#define UMA_ISSET(FIELD, FLAG)  ((FIELD) &   (FLAG))
```

```
#define UMA_CLEAR(FIELD)        ((FIELD) &= 0)

/*********************************************************************/
/*                      UMA Status Codes                             */
/*********************************************************************/
#define UMS_SUCCESS      0      /* No error                          */
#define UMS_NODE        -1      /* Invalid/Unknown host name         */
#define UMS_TIME        -2      /* Invalid time                      */
#define UMS_SOURCE      -3      /* Invalid source                    */
#define UMS_DEST        -4      /* Invalid destination               */
#define UMS_CLASS       -5      /* Invalid class                     */
#define UMS_SUBCLASS    -6      /* Invalid subclass                  */
#define UMS_INTERVAL    -7      /* Invalid interval                  */
#define UMS_PROPERTY    -8      /* Invalid property                  */
#define UMS_SESSID      -9      /* Invalid session identifier        */
#define UMS_PRIORITY    -10     /* Invalid Priorit                   */
#define UMS_PROTOCOL    -11     /* Protocol error                    */
#define UMS_COMM        -12     /* Communication failure             */
#define UMS_SESSION     -13     /* Error has occurred in this session */
#define UMS_NOMSG       -14     /* No message received from UMA      */
#define UMS_SIGNAL      -15     /* A interrupt has occurred          */
#define UMS_EOS         -16     /* Session is about to end           */
#define UMS_ATTR        -17     /* Invalid attribute specified       */
#define UMS_UID         -18     /* Invalid UID                       */
#define UMS_FLAGS       -19     /* Invalid flags                     */
#define UMS_EXPRESSION  -20     /* Invalid selection expression      */
#define UMS_EVENT       -21     /* unsolicited event unsupported     */

/*********************************************************************/
/*                      UMA Reason Codes                             */
/*********************************************************************/
#define UMR_NOREASON     0        /* No reason                       */
#define UMR_INVALID     -1        /* An invalid parameter specified  */
#define UMR_UNKNOWN     -2        /* Unknown src, dest, node, etc    */
#define UMR_NETWORK     -3        /* Network unreacheable            */
#define UMR_TIMEOUT     -4        /* A timeout has occured           */
#define UMR_MAX         -5        /* Max number of sessions reached  */
#define UMR_NOTIMPLEMENTED  -6    /* Class, subclass or event not    */
                                  /* implemented                     */
#define UMR_DISABLED    -7        /* Class or Subclass disabled      */
#define UMR_CONNECT     -8        /* Error in connecting to UMA      */
#define UMR_RECEIVE     -9        /* Error when receiving from UMA   */
#define UMR_SEND        -10       /* Error when sending to UMA       */
#define UMR_CONFLICT    -11       /* Invalid Comb. of arguments      */
#define UMR_INCOMPLETE  -12       /* A required attribute not spec.  */
#define UMR_PERMISSION  -13       /* Permission denied               */
#define UMR_ACTIVE      -14       /* Session already active          */
#define UMR_SYSERR      -15       /* UMA has encountered a system err */
#define UMR_RESOURCE    -16       /* Lack of system resource         */
#define UMR_UMADS       -17       /* Error accessing UMADS           */
#define UMR_INTR        -18       /* An interrupt has occured        */
#define UMR_HEADER      -19       /* An invalid header encountered   */
#define UMR_MESSAGE     -20       /* An invalid message encountered  */
#define UMR_NOTSTARTED  -21       /* Subclass not started            */

#endif  /* MLI_H */
```

A.2 <uma.h>

```
#ifndef UMA_H
#define UMA_H

#include <sys/types.h>

/*
 * ****************** NOTICE *********************************
 *  The <dci.h>, <mli.h> and <uma.h> header files
 *  introduce UMA symbols which may conflict with other
 *  symbols defined in an application.  Symbols with the
 *  following prefixes are therefore reserved to UMA:
 *            DCI
 *            dci
 *            UMA
 *            UMR
 *            UMS
 *
 *   Note that the header files are provided as advisory
 *   reference examples.
 * ****************** END OF NOTICE ***************************
 */

/*****************************************************************/
/*                            UMA data types                     */
/*****************************************************************/

typedef int             UMAInt4;
typedef unsigned int    UMAUint4;
typedef longlong_t      UMAInt8;
typedef ulonglong_t     UMAUint8;

typedef UMAInt4         UMABoolean;
typedef UMAUint4        UMAProvider;
typedef UMAUint4        UMAClass;
typedef UMAUint4        UMAMsgFlags;
typedef UMAUint4        UMANetAddr;

#ifdef ORIGINALUMA
typedef unsigned char UMAOctetString;
#else
/* make sure that unsigned char in MLI definition is used correctly */
#endif

typedef UMAUint4        UMASchedClass;
typedef UMAUint4        UMASubClass;

typedef struct UMASegDescr {
    unsigned char  segtag[4];
    unsigned char  seglenlen;
    unsigned char  seglen[3];
} UMASegDescr;

typedef time_t          UMATimeSec;
typedef UMAInt8         UMATimeNsec;
typedef UMAInt8         UMATimeUsec;
```

```
typedef UMAInt8         UMATimeStamp;

#ifdef timespec
typedef struct timespec UMATimeSpec;
#else
typedef struct UMATimeSpec {
    UMAUint4        tv_sec;
    UMAUint4        tv_nsec;
} UMATimeSpec;
#endif

#ifndef timeval
typedef struct timeval  UMATimeVal;
#else
typedef struct UMATimeVal {
    UMAUint4        tv_sec;
    UMAUint4        tv_usec;
} UMATimeVal;
#endif


/* Support for random access to variable length members of structures
 * requires that the address of these members is derived from fixed
 * size structures at known offsets within the main structure
 * definition.  There are several types of variable length member descriptor
 * structures, all of which contain the offset required to locate the
 * variable length data.  The 'offset' is considered relative to the
 * base address of the parent structure of the variable length member
 * descriptor. Extra information concerning the variable length data
 * may also be available.
 *
 * Note that it is not possible to determine whether a variable length
 * data has been initialized before it is referenced.  As a convention,
 * the offset plus length equal 0 could be used to indicate an
 * uninitialized variable length data item.
 */

/* used as a place holder for variable length data in structs.
/* ------------------------------------------------------------ */
typedef unsigned char UMAVarLenData;    /* Variable-length-date   */
                                        /*     container          */

/* descriptor for a single variable length element which contains
its own size
/* ------------------------------------------------------------ */
typedef struct UMAVarLenDescr {
    UMAUint4        offset;       /* offset to beginning of data */
} UMAVarLenDescr;

/* descriptor for variable length element (which doesn't contain */
/* its own size)                                                 */
/* ------------------------------------------------------------ */
typedef struct UMAElementDescr {
UMAUint4        offset;         /* offset to beginning of data   */
UMAUint4        size;           /* size of the whole structure   */
} UMAElementDescr;
```

```
/* descriptor for variable length text                          */
/* ------------------------------------------------------------ */
typedef struct UMATextDescr {
UMAUint4        offset;         /* offset to beginning of data   */
UMAUint4        count;          /* count of elements in the text */
} UMATextDescr;

/* struct for a variable length string
/* ------------------------------------------------------------ */
typedef struct UMAString {
    UMAUint4        size;       /* size of the entire structure  */
    char            string[1];  /* the variable length           */
                                /* text string                   */
} UMATextString, UMAOctetString;

/* descriptor for variable length array of fixed size elements   */
/* ------------------------------------------------------------ */
typedef struct UMAArrayDescr {
    UMAUint4        offset; /* offset to beginning of data       */
    UMAUint4        count;  /* count of elements in the array    */
    UMAUint4        size;   /* size of each element of the array */
} UMAArrayDescr;

/* descriptor for variable length array of variable sized elements
/* ------------------------------------------------------------ */
typedef struct UMAVarArrayDescr {
    UMAUint4        offset;     /* offset to beginning of data   */
    UMAUint4        count;      /* count of elements in the array*/
} UMAVarArrayDescr;


/* ------------------------------------------------------------ */
 * Constants for common units in UMA
 * Units are organised into 5 types:
 * size, time, count, derived and info units
 *
 */
#define  UMA_UNITS_TIME         0x10000
#define  UMA_UNITS_COUNT        0x20000
#define  UMA_UNITS_SIZE         0x30000
#define  UMA_UNITS_DERIVED      0x40000
#define  UMA_UNITS_INFO         0x50000

enum UMAUnit {
    /* time units */
    UMA_SECS            = 0x01 | UMA_UNITS_TIME,
    UMA_MILLISECS       = 0x02 | UMA_UNITS_TIME,
    UMA_MICROSECS       = 0x03 | UMA_UNITS_TIME,
    UMA_NANOSECS        = 0x04 | UMA_UNITS_TIME,
    UMA_PICOSECS        = 0x05 | UMA_UNITS_TIME,
    UMA_TICKS           = 0x06 | UMA_UNITS_TIME,

    /* count units */
    UMA_COUNT           = 0x01 | UMA_UNITS_COUNT,
    UMA_EVENT           = 0x02 | UMA_UNITS_COUNT,
    UMA_PAGES           = 0x03 | UMA_UNITS_COUNT,
    UMA_BLOCKS          = 0x04 | UMA_UNITS_COUNT,
    UMA_CHARACTERS      = 0x05 | UMA_UNITS_COUNT,
    UMA_QLENGTH         = 0x06 | UMA_UNITS_COUNT,
```

```
UMA_PROCESSES       = 0x07 | UMA_UNITS_COUNT,
UMA_TASKS           = 0x08 | UMA_UNITS_COUNT,
UMA_THREADS         = 0x09 | UMA_UNITS_COUNT,
UMA_JOBS            = 0x0a | UMA_UNITS_COUNT,
UMA_USERS           = 0x0b | UMA_UNITS_COUNT,
UMA_TRANSACTIONS    = 0x0c | UMA_UNITS_COUNT,
UMA_MESSAGES        = 0x0d | UMA_UNITS_COUNT,
UMA_SESSIONS        = 0x0e | UMA_UNITS_COUNT,
UMA_STREAMSMODULES  = 0x0f | UMA_UNITS_COUNT,
UMA_STREAMSHEADS    = 0x10 | UMA_UNITS_COUNT,
UMA_STREAMSMSGS     = 0x11 | UMA_UNITS_COUNT,
UMA_PACKETS         = 0x12 | UMA_UNITS_COUNT,
UMA_INODES          = 0x13 | UMA_UNITS_COUNT,
UMA_FILES           = 0x14 | UMA_UNITS_COUNT,
UMA_FILESYSTEMS     = 0x15 | UMA_UNITS_COUNT,
UMA_READS           = 0x16 | UMA_UNITS_COUNT,
UMA_WRITES          = 0x17 | UMA_UNITS_COUNT,
UMA_SEEKS           = 0x18 | UMA_UNITS_COUNT,
UMA_IOCTLS          = 0x19 | UMA_UNITS_COUNT,
UMA_CONNECTIONS     = 0x1a | UMA_UNITS_COUNT,
UMA_RETRIES         = 0x1b | UMA_UNITS_COUNT,
UMA_MOUNTS          = 0x1c | UMA_UNITS_COUNT,
UMA_REWINDS         = 0x1d | UMA_UNITS_COUNT,
UMA_POSITIONINGS    = 0x1e | UMA_UNITS_COUNT,
UMA_MARKS           = 0x1f | UMA_UNITS_COUNT,
UMA_PORTS           = 0x20 | UMA_UNITS_COUNT,
UMA_PROCESSORS      = 0x21 | UMA_UNITS_COUNT,
UMA_DISKS           = 0x22 | UMA_UNITS_COUNT,
UMA_NETS            = 0x23 | UMA_UNITS_COUNT,
UMA_SLINES          = 0x24 | UMA_UNITS_COUNT,
UMA_BUSSES          = 0x25 | UMA_UNITS_COUNT,
UMA_CHANNELS        = 0x26 | UMA_UNITS_COUNT,
UMA_NOUNITS         = 0x27 | UMA_UNITS_COUNT,

/* size units */
UMA_BYTES           = 0x01 | UMA_UNITS_SIZE,
UMA_KBYTES          = 0x02 | UMA_UNITS_SIZE,
UMA_MBYTES          = 0x03 | UMA_UNITS_SIZE,
UMA_GBYTES          = 0x04 | UMA_UNITS_SIZE,
UMA_TBYTES          = 0x05 | UMA_UNITS_SIZE,

/* derived data units.                                          */
/* The values from 0x01-0x9f are reserved.                      */
/* Values from 0xa0-0xfe may be used for vendor extensions.     */
/* 0x00 and 0xff are unavailable for use                        */
UMA_DERIVED_SUM2            = 0x01 | UMA_UNITS_DERIVED,
UMA_DERIVED_SUM3            = 0x02 | UMA_UNITS_DERIVED,
UMA_DERIVED_DIFFERENCE      = 0x03 | UMA_UNITS_DERIVED,
UMA_DERIVED_AVERAGE         = 0x04 | UMA_UNITS_DERIVED,
UMA_DERIVED_PERCENT         = 0x05 | UMA_UNITS_DERIVED,
UMA_DERIVED_PRODUCT         = 0x06 | UMA_UNITS_DERIVED,
UMA_DERIVED_VARIANCE        = 0x07 | UMA_UNITS_DERIVED,

/* info units */
UMA_CPU                     = 0x01 | UMA_UNITS_INFO,
UMA_MEMORY                  = 0x02 | UMA_UNITS_INFO,
UMA_TASKID                  = 0x03 | UMA_UNITS_INFO,
UMA_THREADID                = 0x04 | UMA_UNITS_INFO,
UMA_PRECEDENCE              = 0x05 | UMA_UNITS_INFO,
```

```
    UMA_ORDER                   = 0x06 | UMA_UNITS_INFO,
    UMA_DATA                    = 0x07 | UMA_UNITS_INFO,
    UMA_TRUEFALSE               = 0x08 | UMA_UNITS_INFO,
    UMA_MODEL                   = 0x09 | UMA_UNITS_INFO,
    UMA_POSITION                = 0x0a | UMA_UNITS_INFO,
    UMA_SECSMILLI               = 0x0b | UMA_UNITS_INFO,
    UMA_ADDR                    = 0x0c | UMA_UNITS_INFO,
    UMA_SIZE                    = 0x0d | UMA_UNITS_INFO,
    UMA_PROTECT                 = 0x0e | UMA_UNITS_INFO,
    UMA_OBJECTNAME              = 0x0f | UMA_UNITS_INFO,
    UMA_MEMOFFSET               = 0x10 | UMA_UNITS_INFO,
    UMA_BYTESIZE                = 0x11 | UMA_UNITS_INFO,
    UMA_MODEL_ID                = 0x12 | UMA_UNITS_INFO,
    UMA_STATE                   = 0x13 | UMA_UNITS_INFO,
    UMA_PROCESSOR_SPEED         = 0x14 | UMA_UNITS_INFO
};
typedef enum UMAUnit UMAUnit;

/* ------------------------------------------------------------ */
/* Instance type definitions.                                   */
enum UMAInstTagType {
    UMA_SINGLEINST          = 1,    /* A single instance of     */
                                    /* value '0' exists         */
    UMA_WORKINFO            = 2,    /* UMA_WORKINFO enumeration */
    UMA_WORKID              = 3,    /* Data associated          */
                                    /* with UMA_WORKINFO        */
    UMA_MSG_QUEUE           = 4,
    UMA_SEMAPHORE           = 5,
    UMA_SHR_SEGMENT         = 6,
    UMA_PROCESSOR           = 7,    /* processor number         */
    UMA_FSGROUP             = 8,
    UMA_MOUNTPOINT          = 9,
    UMA_INODE               = 10,   /* inode number             */
    UMA_DISKID              = 11,   /* disk device number       */
    UMA_BUCKET_NO           = 12,
    UMA_DISKPARTITION       = 13,
    UMA_ACCESS_PORT         = 14,
    UMA_DEVICE              = 15,   /* generic device number    */
    UMA_KERNEL_TABLES       = 16,
    UMA_CHANNEL             = 17,   /* channel number           */
    UMA_IOP                 = 18,   /* IO processor number      */
    UMA_PATH                = 19,
    UMA_SYSCALL             = 20,   /* system call number       */
    UMA_ENUMERATION         = 21,
    UMA_STREAMS             = 22,
    UMA_CONTROLLERID        = 23,   /* controller number        */
    UMA_SCHED_CLASS         = 24,   /* scheduling class type    */
    UMA_LOGICALVOL          = 25,
    UMA_REMOTE_FSTYPES      = 26,
    UMA_IPADDR              = 27,
    UMA_FILESERVER_COMMAND  = 28,
    UMA_FILECLIENT_COMMAND  = 29,
    UMA_SERVER_COMMAND      = 30,
    UMA_CLIENT_COMMAND      = 31,
    UMA_MEMOBJECT_ID        = 32,
    UMA_HOSTPORT            = 33,
    UMA_TASKPORT            = 34,
    UMA_THREADPORT          = 35,
    UMA_DPGRPORT            = 36,
```

```
    UMA_PRCSCTLPORT         = 37,
    UMA_VMADDRESS           = 38
};
typedef enum UMAInstTagType UMAInstTagType;


/* ------------------------------------------------------ */
/* Default UMAWorkInfo definitions:                       */
#define  UMA_WORKINFO_PROJECT                   1<<0
#define  UMA_WORKINFO_GROUP_ID                  1<<1
#define  UMA_WORKINFO_EFFECTIVE_GROUP_ID        1<<2
#define  UMA_WORKINFO_USER_ID                   1<<3
#define  UMA_WORKINFO_EFFECTIVE_USER_ID         1<<4
#define  UMA_WORKINFO_SESSION_ID                1<<5
#define  UMA_WORKINFO_TTY                       1<<6
#define  UMA_WORKINFO_NQS                       1<<7
#define  UMA_WORKINFO_SCHEDULING_CLASS          1<<8
#define  UMA_WORKINFO_SCHED_GRP                 1<<9
#define  UMA_WORKINFO_TRANSACTION_ID            1<<10
#define  UMA_WORKINFO_PROCESS_GRP               1<<11
#define  UMA_WORKINFO_PARENT_PROCESS_ID         1<<12
#define  UMA_WORKINFO_COMMAND_NAME              1<<13
#define  UMA_WORKINFO_PROCESS_ID                1<<14
#define  UMA_WORKINFO_THREAD_ID                 1<<15

/* fundamental data types */
enum UMADataType {
    UMA_INT4        = 1,
    UMA_INT8        = 2,
    UMA_UINT4       = 3,
    UMA_UINT8       = 4,
    UMA_BOOLEAN     = 5,
    UMA_OCTETSTRING = 6,
    UMA_TEXTSTRING  = 7,
    UMA_TIMEVAL     = 8,
    UMA_TIMESPEC    = 9,
    UMA_TIMESTAMP   = 11,
    UMA_DERIVED     = 12,
    UMA_CLASSDATA   = 13
};
typedef enum UMADataType UMADataType;

/* descriptor types */
enum UMADescrType {
    UMA_NODESCR       = 0,
    UMA_ELEMENTDESCR  = 1,
    UMA_TEXTDESCR     = 2,
    UMA_ARRAYDESCR    = 3,
    UMA_VARARRAYDESCR = 4,
    UMA_VARLENDESCR   = 5
};
typedef enum UMADescrType UMADescrType;

/*********************************************************************/
/*                      Standard UMA message header                  */
/*********************************************************************/

typedef struct UMAHeader {
    unsigned char   mh_msgtag[4];   /* BER encoded indicator tag  */
```

```
    unsigned char   mh_msglenlen;   /* ASN.1/BER length of length */
    unsigned char   mh_msglen[3];   /* msg length from next field */
    unsigned char   mh_hdrtag[3];   /* msg header tag             */
    unsigned char   mh_hdrlen;      /* msg header length          */
    UMAMsgFlags     mh_flags;      /* Data Modes                 */
    UMATimeUsec     mh_time;       /* timestamp of msg creation  */
    UMAClass        mh_class;      /* UMA Class of the message   */
    UMASubClass     mh_subclass;   /* UMA Subclass of the message */
    unsigned char   mh_address[8];  /* Host network address      */
    unsigned char   mh_addr_family; /* Host network address type */
    UMAUint4        mh_provider;   /* Data provider              */
    UMAUint4        mh_provinst;   /* Data provider instance     */
} UMAHeader;

/*********************************************************************/
/*                    UMA interval message header extension          */
/*********************************************************************/

typedef struct  UMAIntExt {
    unsigned char   mhix_ixlenlen;  /* Extension tag and length   */
    unsigned char   mhix_ixlen[3];  /* Extension length           */
    UMAMsgFlags     mhix_flags;     /* Interval extension flags   */
    UMATimeSec      mhix_schedtime; /* Time of sched. measurement */
    UMATimeUsec     mhix_intime;    /* Actual interval start time */
    UMATimeUsec     mhix_intlen;    /* Actual interval duration   */
    UMAUint4        mhix_baseoff;   /* Global to basic offset     */
    UMAUint4        mhix_optoff;    /* Global to optional offset  */
    UMAUint4        mhix_extoff;    /* Global to extension offset */
} UMAIntExt;

/*********************************************************************/
/*                    UMA event message header extension             */
/*********************************************************************/

typedef struct UMAEvtExt {
    unsigned char   mhex_exlenlen;  /* Extension tag and length   */
    unsigned char   mhex_exlen[3];  /* Length of event extension  */
    UMAMsgFlags     mhex_flags;     /* Event extension flags      */
    UMATimeUsec     mhex_evtime;    /* Timestamp of event         */
    UMAUint4        mhex_baseoff;   /* Global to basic offset     */
    UMAUint4        mhex_optoff;    /* Global to optional offset  */
    UMAUint4        mhex_extoff;    /* Global to extension offset */
} UMAEvtExt;

/*********************************************************************/
/*               UMA message header data modes                       */
/*********************************************************************/

#define UMA_THRESH      0x0001  /*  Threshold screening applied       */
#define UMA_UNUSED1     0x0002  /*  Available for future use          */
#define UMA_PROV_SCLASS 0x0004  /*  Subclass is provider-specific     */
#define UMA_PROV_CLASS  0x0008  /*  Class is provider-specific        */
#define UMA_L_SUB       0x0010  /*  Last message for this subclass    */
#define UMA_UNUSED2     0x0020  /*  Available for future use          */
#define UMA_L_CLASS     0x0040  /*  Last message for this class       */
#define UMA_F_CLASS     0x0080  /*  First message for this class      */
#define UMA_DST         0x0100  /*  Daylight Saving times in effect   */
#define UMA_UNUSED4     0x0200  /*  Available for future use          */
#define UMA_EVTHDR      0x0400  /*  Event header extension present    */
```

```
#define UMA_INTVAL      0x0800  /*  Interval header extension present */
#define UMA_UNUSED5     0x1000  /*  Available for future use          */
#define UMA_R_L         0x2000  /*  R-L byte ordering                 */
#define UMA_B_FORM      0x4000  /*  Canonical B format                */
#define UMA_CNTRL       0x8000  /*  Control message                   */

/*********************************************************************/
/*        UMA interval/event message header extension flags          */
/*********************************************************************/

#define UMA_UNUSED6     0x0001  /*  Available for future use          */
#define UMA_UNUSED7     0x0002  /*  Available for future use          */
#define UMA_UNUSED8     0x0004  /*  Available for future use          */
#define UMA_UNUSED9     0x0008  /*  Available for future use          */
#define UMA_UNUSED10    0x0010  /*  Available for future use          */
#define UMA_UNUSED11    0x0020  /*  Available for future use          */
#define UMA_UNUSED12    0x0040  /*  Available for future use          */
#define UMA_UNUSED13    0x0080  /*  Available for future use          */
#define UMA_ASEG        0x0100  /*  All segments requested            */
#define UMA_ESEG        0x0200  /*  Extension segment present         */
#define UMA_OSEG        0x0400  /*  Optional segment present          */
#define UMA_BSEG        0x0800  /*  Basic segment present             */
#define UMA_UNUSED14    0x1000  /*  Available for future use          */
#define UMA_SRC_RECENT  0x2000  /*  Source for this msg: recent hist  */
#define UMA_L_INT       0x4000  /*  Last message for interval         */
#define UMA_F_INT       0x8000  /*  First message for interval        */

#endif  /* UMA_H */
```

Appendix B

Future Directions

B.1 UMA Generalized Command Interface

An additional MLI capability is currently being prototyped to assist in remote administration of UMA-instrumented platforms by creating and administering remotely available services. This capability, called the "UMA Generalized Command Interface" or UMA-CCI, extends the available message classes with the class "GCI". UMA-GCI currently has the following subclasses defined:

INIT
: Initialize all GCI variables and build GCIService array

LIST
: Return GCIService structure with data_err_size set

GETDATA
: Returns public GCIService array

ATTRIBUTES(register)
: Sets a GCIService available for use (administrative only)

EXEC
: Executes an available GCIService

GETSTATUS
: Check execution status of GCIService

GETERROR
: Retrieve error output of GCIService

ATTRIBUTES(remove)
: Sets a GCIService unavailable for use (administrative only)

ATTRIBUTES(hide)
: Hides a GCIService from other users (administrative only)

SAVE
: Save the GCIService array (administrative only)

END
: Free GCIService array

Glossary

ascii
Term used in this specification for referencing the American Standard Code for Interchange of Information: 8-bit (128-character) code set.

ASN.1/BER
Abstract Syntax Notation One / Basic Encoding Rules - The ASN.1 language describes all abstract syntaxes in the OSI architecture. An abstract syntax is a named group of types. BER, the Basic Encoding Rules, is a *transfer syntax* used to communicate data between open systems. It includes those aspects of the rules used in the formal specification of data which embody a specific representation of that data. BER is capable of encoding any abstract syntax that can be described using ASN.1.

API
Application Program Interface - In general, a standard interface for programmatic access to services; MLI, a service API for UMA, is defined in this document.

Collection Interval
The time between successive captures of a specific set of data items. The term *interval* is also used to mean the data for a collection interval having a certain time stamp and duration.

Constructed Workload
A named workload data collection that results from a joint specification of any one or more of: Multiple class/subclass specifications, a granularity/summarization specification, an instance or UMAWorkInfo filtering specification, a complement workload specification.

DCI
Data Capture Interface - A standard UMA interface to access data sources such as kernel and subsystem data structures, hardware dependent data, and data which is event-generated.

Data Capture Layer
A UMA entity concerned with the collection of raw data from the operating system kernel and from other sources. Data is considered collected when it exists assembled into data structures of predefined class and subclass in storage controlled by services contained in the measurement model.

Data Acquisition Node
A physical entity (for example, a processor) that executes a UMA Data Capture and a UMA Data Services Layer.

Data Class
The general system measurement entity to be collected. For example, the data classes for UMA include system configuration information, processor and memory usage information, and other like categories.[9]

Data Provider
A logical entity in the Data Capture Layer that makes data available to UMA and its users,

9. Data classes and subclasses for the Universal Measurement Architecture are described in the accompanying document Data Pool Definitions (see reference **DPD**).

usually through the DCI.

Data Services Layer
The layer responsible for data distribution to measurement applications (which use the MLI for archival data storage), management of services and resources required for distributed measurement access and control, measurement requesting, and data format transformations required for recording and transmission.

Data Subclass
A specific grouping of data within a data class. Each data class may have several data subclasses. For instance, the Data Pool class *processor* contains subclasses such as *Global Measured Processor Times* and *Global System Call Counters*, etc.

Event Data
In the context of UMA, this represents the reporting of one or more system events (for example, process termination, creation, signal delivery, logon, etc.).

i18n
abbreviation used in this specification for the term *internationalization* (which has 18 letters between its first and last letters)

Measurement Application Layer
This functional layer contains the application primitives and tools used to report currently captured and archival performance data to the end-user (or to an automated stand-in). These applications are called Measurement Application Programs (MAPs).

Measurement Application Node
A physical entity that executes a MAP and a UMA Data Services Layer.

MAP
Measurement Application Program

Measurement Control Layer
The layer responsible for managing the capture of data, including its synchronisation, and for providing any necessary buffer or queue management for data assembled by the data capture mechanism.

Measurement Interval
A continuous time interval during which measurement activity and reporting is requested by a MAP.

Message
In UMA, a basic unit of control or data information. Each UMA message contains a header portion which identifies the class and subclass of the information contained in the rest of the message.

MAP
Measurement Application Program - A UMA-based application program providing end-user services.

MLI
Measurement Layer Interface - The MLI comprises the Application Programming Interface (API) for UMA, and the management of UMA message transport.

Presumed Location (UMADS)
The location of historical data for a node that UMA determines through administrative policy.

Recent Data Facility
A UMA storage entity that caches the most current data captured by DCL data providers.

Regular Expression
In the context of the MLI, a text matching pattern constructed according to the rules described for Basic Regular Expressions (BREs) in the System Interface (XSH) CAE Specification (see reference **XSH**).

Reporting Interval
The union of one or more contiguous collection intervals to be seen by a MAP or by UMADS. Thus the reporting interval may be identical to a collection interval or it may have a duration that is (nominally) a multiple of the collection interval duration.

Sampling Interval
The time between successive samples during data capture.

Session
In UMA, a logical communications channel between a MAP and the UMA facility. A MAP can establish multiple concurrent sessions.

Trace Data
In the context of UMA, reported trace data is data for a set of selected related events.

UDU
UMA Data Unit - The contents of a UMA API message. The UDU consists of a header portion and either a control segment or one or more data segments.

UMA
Universal Measurement Architecture - A common, flexible measurement control and data delivery mechanism.

UMADS
UMA Data Storage - An archive that stores historical performance, resource usage, and accounting information. (In UMA, *historical* data is that for which the time of capture is earlier than time *now*.)

Index

CAE Specification

Part 3:
UMA Data Capture Interface (DCI)

The Open Group

Contents

List of Figures

List of Tables

Preface

This Document

This document is a CAE Specification. It defines the requirements for a programming interface for the lowest layer in the Universal Measurement Architecture (UMA) — the interface between the data capture layer and the measurement control layer. It includes sufficient background for the reader to understand the problem being solved and the source of the programming interface requirements, as well as defining an interface (the DCI) which meets these requirements.

There are two associated UMA specifications which along with the DCI specification define the UMA system:

- UMA Measurement Layer Interface (MLI) specification (see Part 2 of this specification).
  This defines the functional characteristics of the MLI, and the underlying semantics and function calls that implement them. It also defines a format for headers appended to measurement data captured through the DCI.
- UMA Data Pool Definitions (see Part 4 of this specification).
  The data pool defines a set of performance metrics which may be accessed by the two UMA interfaces.

The UMA Guide (see Part 1 of this specification) reviews the issues surrounding performance measurement in Open Systems, describes the general UMA architecture, and discusses user considerations in adopting the UMA.

Audience

The target audience for this document is both system designers, who need to implement this interface, and performance professionals, who need to understand how this interface can be used.

Structure

- **Chapter 1, Introduction** — provides a high level overview of the requirements
- **Chapter 2, DCI Architectural Description** — describes the overall architecture of the Data Capture Interface, presenting in tutorial style a general description of the problem being solved and the requirements for a solution
- **Chapter 3, Overview of the DCI Specification** — describes the metrics name space, metric attributes and data types, and a summary of the groups of routines decribed in the following chapters. In the case of any conflicts between the material presented in Chapter 2 or Chapter 3 the reader should consider Chapter 3 to be authoritative.
- **Chapter 4, DCI Routines Overview** — provides an overview of the DCI routines described in more detail in the subsequent chapter
- **Chapter 5, Metrics Consumer Routines** — describes the interfaces used by Metrics Consumers
- **Chapter 6, Metrics Provider Routines** — describes the interfaces used by Metrics Providers
- **Chapter 7, Event Routines** — describes the interfaces which handle events

- **Appendix A, C Language Header Files** — presents the **<dci.h>** header file.

Acknowledgements

This specification was developed by the Performance Management Working Group (PMWG). The PMWG was originally part of UNIX International, and is now part of the Computer Measurement Group.

the Open Group gratefully acknowledges the work of the PMWG in the development of this specification and in the review process for this publication.

Major contributors to the Data Capture Interface specification include:

Sara Abraham	Amdahl Corporation	Peter Benoit	Digital Equipment Corp.
Robert Berry†	IBM Corporation	Niels Christiansen	IBM Corporation
Paul Curtis	Hitachi Computer Products (America), Inc.	Paul Douglas	Digital Equipment Corp.
Janice Dumont	AT&T Bell Laboratories	Mark Feldman	Sequent Computer Systems, Inc.
Ken Gartner	Hitachi Computer Products (America), Inc.	Javad Habibi	Amdahl Corporation
Suzanne John	IBM Corporation	Bill Laurune	Digital Equipment Corp.
Jee-Fung Pang	Digital Equipment Corp.	David Potter	Open Systems Performance, Inc.
Jim Van Sciver††	Open Software Foundation	Jaap Vermeulen	Sequent Computer Systems, Inc.
Ping Wang	Open Software Foundation		

Participants who have made contributions to the process of developing these specifications are listed below along with their corporate affiliation at the time of their contribution. Our sincere apologies to anyone whom we may have missed.

Subhash Agrawal	BGS Systems	Barrie Archer	ICL
Tom Beretvas	IBM Corporation	Wolfgang Blau	Tandem Computers, Inc.
Jim Busse	NCR Corporation	David Butchart	Digital Equipment Corp.
David Chadwick	Performance Awareness Corp.	Ram Chelluri	AT&T Global Information Solutions
Danny Chen	AT&T Bell Laboratories	Ansgar Erlenkoetter	Tandem Computers, Inc.
Paul Farr	Aim Technology	Jerome Feder	UNIX System Laboratories
Thierry Fevrier	Hewlett-Packard	Lewis T. Flynn	Amdahl Corporation
Tony Gaseor	AT&T Bell Laboratories	Joseph Glenski	Cray Research, Inc.
Dave Glover	Hewlett-Packard	Jay Goldberg	UNIX System Laboratories
William Hidden	Open Software Foundation	Liz Hookway	NCR Corporation
John Howell	Amdahl Corporation	Ken Huffman	Hewlett-Packard
Mario Jauvin	Bell Northern Research	Chester John	IBM Corporation
Suzanne John	IBM Corporation	Rebecca Koskela	Cray Research, Inc.
Ted Lehr	IBM Corporation	Greg Mansfield	Instrumental
Shane McCarron	UNIX International	Michael Meissner	Open Software Foundation
Marge Momberger	IBM Corporation	Bernice Moy	Open Software Foundation
Henry Newman	Instrumental	James Pitcairn-Hill	Open Software Foundation
Melur K. Raghuraman	Digital Equipment Corp.	O. T. Satyanarayanan	Amdahl Corporation
Yefim Somin	BGS Systems	Jim Richard	Amdahl Corporation
Steve Sonnenberg	Landmark Systems	Douglas R. Souders	UNIX System Laboratories
Leon Traister	Amdahl Corporation	Michael Wallulis	Digital Equipment Corp.
Steve Whitney	Boeing Computer Services	Elizabeth Williams	Super Computer Research
Willie Williams	Open Software Foundation	Neal Wyse	Sequent Computer Systems, Inc.
Seung Yoo	Amdahl Corporation		

† Editor

†† Past Editor

Chapter 1

Introduction

1.1 Purpose

This document is one of a family of documents that comprise the Universal Measurement Architecture (UMA), which define interfaces and data formats for performance measurement. UMA was originally defined by the Performance Management Working Group (PMWG) and subsequently adopted by The Open Group.

This document defines the requirements for a programming interface for the lowest layer in the UMA performance metrics architecture. The purpose of this document is twofold. The first is to provide sufficient background for the reader to understand the problem being solved and the source of the programming interface requirements. The second is to provide a specification of an interface that would meet these requirements.

The UMA is defined in the following documents:

- Guide to the Universal Measurement Architecture (see reference **UMA**). This document provides an overview of the UMA.
- UMA Measurement Layer Interface Specification (see reference **MLI**). This document defines functional characteristics for a high-level open Application Program Interface (API) to be used by Measurement Application Programs (MAPs) to request and receive data. It also defines header formats to be appended to the data captured by a low-level Data Capture Interface (DCI).
- UMA Data Capture Interface Specification (this document).
- UMA Data Pool Definitions (see reference **DPD**). This document defines a performance data pool (a set of operating system metrics) for the analysis and management of computer systems, and an organisation to facilitate the collection and use of such data. The Data Pool specification describes the metric data types, and groups metrics into classes and subclasses.

1.2 Scope

Performance and capacity management of operating systems have been considered 'internal' to the operating system and as such differ from operating system to operating system and from implementation to implementation. Most operating systems have, as a matter of necessity, performance analysis modules, narrowly targeted at the type of hardware, software and networking facilities implemented within the system.

Most operating systems provide ad-hoc developed or tailored performance metrics and tools. Some of these tools are developed as internal support tools for benchmarking, or on demand of performance analysts and capacity planners. These tools are generally also confined to one machine only and can not be interrogated remotely.

The new era of networking and interoperability views performance management and capacity planning from the user's perspective. Multiple machines and operating systems can be involved in the interaction with the user. This approach requires the capture and reporting of performance metrics to be clearly defined and portable between platforms and operating systems.

Historically, many operating systems have not had a well defined interface for acquiring information about system performance. These systems typically provide a number of user commands for obtaining the status of system resources, such as **ps**, **sar** and the **stat** commands of UNIX. However, programs which require instrumentation beyond that provided by these utilities, or applications which require higher metric acquisition rates, have resorted to interrogating the operating system themselves. In these cases, privileged access to system memory (for example, kernel memory in UNIX, system control blocks for MVS) is the only available mechanism. Such access is notoriously slow, and can require the development and maintenance of specialised subsystems linked in with the operating system. Clearly, a flexible, extensible interface to system metrics is required.

The Performance Management Working Group has provided a solution to this problem. This solution, the Universal Measurement Architecture (UMA), specifies two architectural layers for system performance metrics. The uppermost layer specifies an API for the transfer of formatted measurement data. The lower layer specifies the interface for acquiring raw measurement data. The term *formatted* implies that the upper layer provides measurement services that modify the content and format of the data on behalf of the user. For details, see the Measurement Layer Interface specification (reference **MLI**). *Raw* implies that the measurement data is acquired in its original, unchanged form.

The Data Capture Layer is the term used for this lowest layer in this performance metrics architecture. The programming interface for the Data Capture layer is called the Data Capture Interface, or DCI. Measurements provided by the DCI are intended to be collected, interpreted, or distributed by higher level applications or services before their consumption by a user. The DCI provides an alternative to, and eventually a replacement for, older mechanisms such as reading the kernel memory device on UNIX systems.

The DCI is intended for anyone writing system performance monitoring or management tools. A correctly specified Data Capture Interface provides several benefits to the performance community. The first is that they will be able to write portable measurement applications because they will not have to adapt the application to each operating system variant and release. Another benefit is that the DCI will allow toolmakers to provide services beyond those that could be supported by the current state of metric acquisition. Also, by establishing a solid foundation, higher levels of standardisation, as described by the UMA, can be specified. Finally, DCI can become part of the lowest layers of the emerging effort to standardise distributed system management.

1.2.1 Goals

There are two classes of goals for the DCI. The first class consists of those goals that meet the needs of the performance community. The second class consists of those goals that must be met to satisfy requirements that the system imposes on any new interfaces.

The DCI application programming interface must meet the following goals to satisfy the needs of the performance community. (A detailed description of each goal follows this section.)

Performance
: The addition and use of the DCI should not significantly alter the performance of an existing system.

Standardisation
: Standardise the interface for retrieving performance data.

Portability
: The DCI should be implementable on a wide variety of systems.

Multiple Metrics Sources
: The DCI should allow for metric providers other than the operating system.

Extensibility
: The DCI should have little or no inherent knowledge of the structure of the data being provided.

Efficient Enablement for Multi-system Measurement
: The DCI, while limiting its scope to metrics provided by a single system, must allow for an efficient multi-system measurement solution.

Polled Metrics and Events
: Both passive and interrupt driven metrics acquisition must be supported.

Modification of Configuration Data
: The DCI should allow for the possibility that metrics, such as configuration values, be modified by a properly privileged MAP. It is implementation defined whether such modification requests are honoured.

The following are requirements imposed by the current state of operating system software.

Security
: The interface should not preclude a secure implementation.

Multiprocessor Support
: The interface should not preclude implementation on multiprocessor architectures.

Internationalisation
: Any textual information should be capable of supporting an internationalised implementation.

Interoperability
: The interface should fit into a distributed environment.

1.2.2 Performance

The addition of any metrics acquisition subsystem should not noticeably affect the performance of the measured system. (Many performance tool builders assert that system performance should not be altered by more than 5% when there is measurement activity.) Although it is beyond this specification to stipulate a performance degradation figure, that figure belongs in an implementation's design specification, the performance goal does impose a requirement that the programming interfaces specified in this document be capable of being implemented in the most efficient manner possible on the target operating system.

1.2.3 Standardisation and Portability

The reason for these two goals stems from a desire to have this API be accepted as a standard for system metrics acquisition. The intent is to propagate the Data Capture Interface across a wide range of systems so that performance applications can themselves be ported to those systems. To do so, the specification of the programming interface cannot impose constraints which would make implementations difficult. An example of such a constraint would be a requirement that the entire DCI be implemented in either the system or user address space. Such a requirement unnecessarily limits the implementor's options, possibly precluding implementation on some systems, and would thus decrease DCI availability to the performance community.

1.2.4 Multiple Metrics Sources

The goal of allowing non-operating system information providers arises from practical considerations. Metrics collection and management is a system problem. Not all metrics necessarily reside in the operating system. Many metrics also exist in user space server programs or in applications. This decentralised information is exaggerated in the case of microkernel operating system architectures, although it also exists in operating systems with traditional structures. This goal allows the API to not only cover a wide range of system types but also allows for the introduction of vendor supplied metrics to a local system. This goal requires that loadable subsystems, such as device drivers, streams modules, and transaction processing applications, be able to register and provide their metrics.

1.2.5 Extensibility

One of the keys to extensibility is to allow variable representations of metric data. If the API requires that applications have prior knowledge of the availability of metrics and their data types, then this information would have to be maintained separately. When the metadata (information that describes a data structure) and the data it describes are disjoint, then possible version skew problems are introduced. Therefore it is a requirement that the API specify "self-describing" data structures. The solution should allow applications to read both the metrics metadata and the data it describes and not require prior knowledge of the metadata contents.

1.2.6 Efficient Enablement for Multi-system Measurement

The limitation of this API's scope to the information coming from a single system is motivated by practicality. (In this document, a single system refers to any machine or machines which use a single operating system image and could be uniquely identified on a network.) The Data Capture API provides the most basic of services: locating system information and providing that information to a higher level application. More complex services, such as distributing the information across a network of machines or data reduction, will use the information gathered by this API and should be provided by higher layers. Limiting the scope of this specification to a single system has the added benefit of deferring the problems of heterogeneous data representation to higher layers, where they belong. However, defining the scope to cover a

single system should not prevent solutions of multiple system issues at the higher layers.

1.2.7 Polled Metrics and Events

Much of performance monitoring consists of polling activity counts. However it is also useful to provide event driven information. An application should be able to request notification of one or more events. The event features should not preclude supporting a wide range of event throughput requirements; from low speed events such as disk mounting and unmounting to the high bandwidth required by trace packages.

1.2.8 Modification of Configuration Data

Often the application measuring performance may wish to take steps to address a perceived performance problem. This might take the form of configuration changes (e.g., widening network timeout windows). The DCI can offer a front-end to simple configuration management, if the underlying implementation supports it.

1.2.9 Security

It is now a computer industry requirement that security issues be addressed whenever specifying the collection and distribution of information. The information specified by the Data Pool document (see reference **DPD**) covers the activity of an entire system. A secure system implementation must concern itself with discretionary and mandatory access control between objects (metrics) and subjects (metrics requesters) before allowing the metrics to be propagated. This specification has been designed to be implemented on systems with the full range of security requirements: from no security to the highest security levels. Note that although this specification must allow for highly secure implementations the choice of security level is dependent on the target system.

1.2.10 Multiprocessor Systems

Multiprocessor systems are becoming commonplace in the computing world. Any programming interface must take into account these systems and allow for the provision of a *thread-safe* environment. When designing with multiprocessor systems in mind, one should avoid the introduction of global variables. Global variables work perfectly well in single threaded environments but require synchronised access in multi-threaded environments. Beyond taking care to not introduce functions which require undue synchronisation, multiprocessor support should be relatively straight forward.

1.2.11 Internationalisation

It is no longer acceptable in the world of computing to limit textual output to English. Should the specification call for the use of any text fields, these fields should be capable of providing the native system's ability to support internationalised text. As with security, the decision to provide internationalised text is dependent on the target implementation.

Such is the frequency of use of the word *internationalization*, that it it commonly abbreviated in this and other documents to *i18n* or *I18n*.

1.2.12 Interoperability

Although this API is intended to be a single machine interface, it should provide the metrics in a format that does not make it difficult to transport those values in a heterogeneous, networked environment. Furthermore, application software should be capable of operating upon the data regardless of the machine that is eventually used for the operation. It is intended that this interface work well as low level support for a range of distributed management and performance applications.

1.2.13 Non-goals

This section describes features which are considered to be out of the scope of this specification.

- It is not within the scope of this document to provide specific implementation details, such as the mechanism that must be used to access operating system metrics. Those details would be found in a design specification. In fact, this specification attempts to leave the designer as much room as possible in the selection of the implementation.
- This specification does not cover management services for information providers. This work focuses on the lowest level services: data transfer from providers to users. Operations which control a system's collection of information providers belong in a higher level design.

1.3 Definitions, Acronyms and Abbreviations

Terms, acronyms and abbreviations used in this specification are defined in the Glossary.

1.4 Conformance

A conformant implementation must support the basic support functions as listed in Table 4-2 on page 70 of this specification. In addition, an implementation may support any of the following DCI API subsets:

- Multiple Providers
- Event Support
- Set Capability Support
- Access Control.

These are also listed in Table 4-2 on page 70.

For each of these additional capabilities supported in an implementation, all DCI APIs in the corresponding subset must be implemented in their entirety.

Functions in unsupported subsets must be implemented to the extent that they return the [DCI_NOIMPLEMENTATION] error code.

Chapter 2

DCI Architectural Description

2.1 Overview

The Data Capture Interface is the lowest layer in the UMA performance metrics architecture. The problem the DCI must solve is how to transmit metrics efficiently from the various sources or providers to the metric consumers or upper architectural layers.

In addition to the responsibility of transferring system metrics, the DCI design must also meet the goals laid down in the introduction. The design of the interface must be especially concerned with the goals of extensibility, portability, and with gathering of metrics from multiple sources.

The problem to be solved by the DCI is a communications problem, in this case, the communication of system metrics between multiple information providers and multiple information consumers where all of the providers and consumers reside on a single system. Although metrics information comes from a single system, the ‘‘system’’ is not a monolithic entity, but a collection of services. These services include the operating system (what is traditionally thought of as the ‘‘system’’), user space server programs, and applications. Thus it is useful, in spite of the DCI single system scope, to describe the solution in networking terms.

The DCI solution can be seen in Figure 2-1.

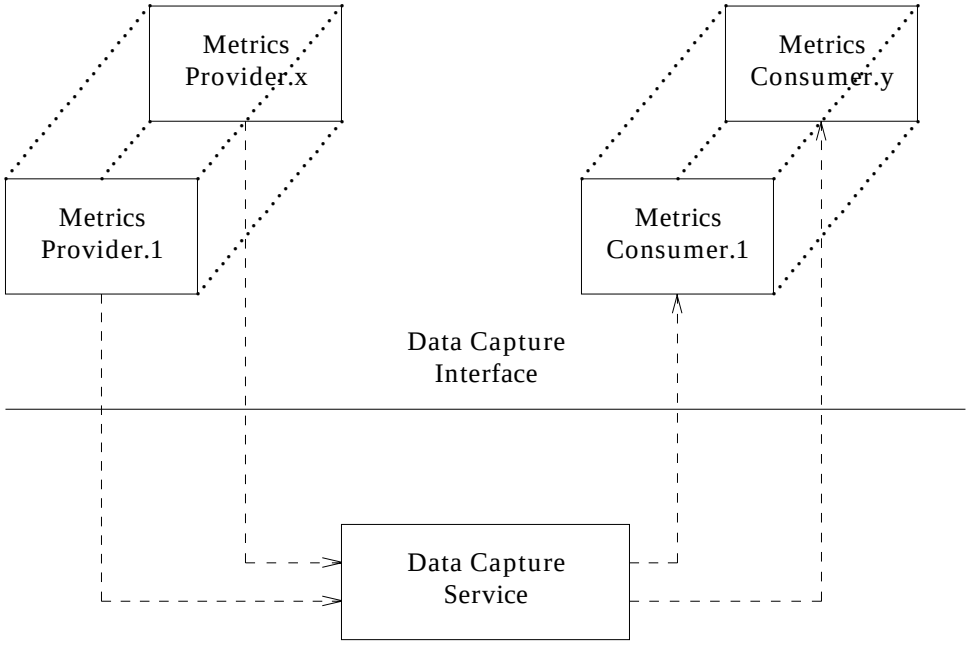


Figure 2-1 Generalised Data Capture Architecture

In Figure 2-1, the collection of system metrics are made available by multiple metric providers. The set of providers for a system is the total supply of that system’s performance metrics. In addition to the providers, there are a variable number of applications that are potential metric consumers. The consumer applications could be the final consumer for the performance

information or they could be acting as an intermediary. These metric consumers could form the lower level of system performance display or collection programs. They could also be network applications which forward and/or accumulate performance information from multiple systems. As indicated by the arrows, metrics flow from the providers to the consumers.

Both the providers and consumers use the same interface — the DCI — to perform their tasks, although they use different aspects of that interface. The ‘‘service’’ provided by the DCI is a set of functions that can be used by the metrics providers and consumers. These functions give providers the ability to transmit metrics and consumers to receive metrics without prior knowledge of each other's existence or of the underlying transport mechanism. This work is carried out invisibly by the underlying service, referred to as the DCI metrics service or server.

The concepts of ‘‘provider’’ and ‘‘consumer’’ refer to roles adopted with respect to the interface. Any particular software entity is free to adopt each of these roles as required. In particular, whilst a provider may obtain performance data by ‘‘private’’ means, it may also obtain such data by being a consumer. Such a situation may arise, for example, where a provider used basic performance data to provide performance information which related to higher level entities within a system.

Thus the Data Capture Interface can be completely described by its set of functions, the structure of the transmitted data, and the behaviour of the DCI Server. The details of this specification can be found in Chapter 3. The rest of this chapter provides sufficient background information to enable the reader to understand the context for this specification. This section describes the services provided by the Data Capture Interface, the means by which those services are provided, the components that make up the DCI, and where those components reside.

The relationships between consumers, providers, and the DCI service can be seen in Figure 2-2. In comparison to Figure 2-1 on page 7, this diagram identifies the main DCI elements. It shows that the Data Capture Interface corresponds to a service boundary layer for providers and consumers. This boundary represents the scope of the DCI specification.

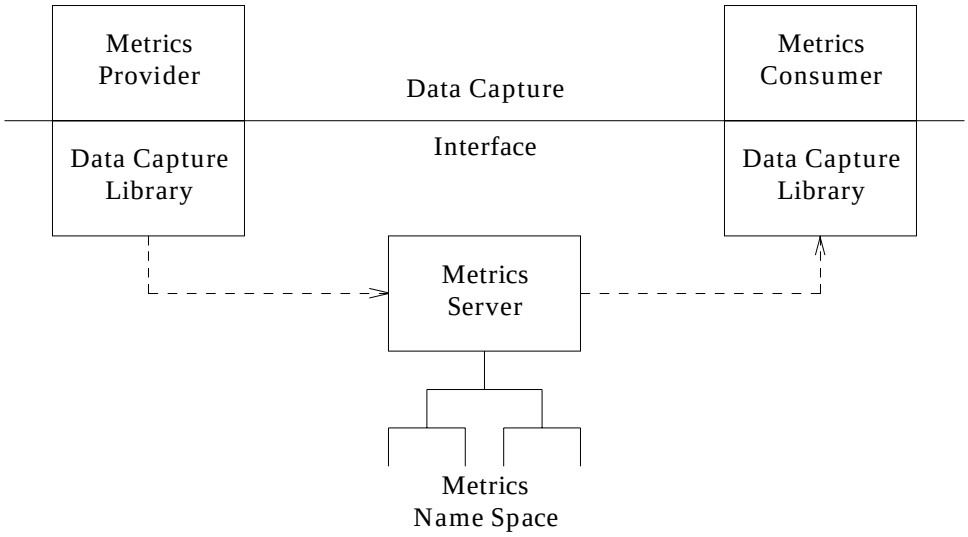


Figure 2-2 Data Capture Architecture

2.2 DCI Services

The DCI provides two primary services:

- one makes the connection between metrics providers and consumers
- the other maintains a metrics name space on behalf of the providers and consumers.

In its role as a metrics transport mechanism, the DCI allows consumers to acquire metrics without having any inherent knowledge of their source. Likewise, providers can transmit metrics without any knowledge of their destination. This metrics transport service is analogous to a layer in a network architecture.

Note: The reader should note that the scope of the DCI is limited to transporting metrics between providers and consumers on a single system.

Most systems have multiple transport mechanisms that can be used to pass data between the metrics providers and the metrics consumers. These mechanisms can be divided into two classes. The first class requires the metrics provider to actively participate in the transport. The second class requires no action on the part of the metric provider other than the registration and maintenance of the statistics.

An example of the latter class is shared memory: the metrics provider indicates the location of its metric to the DCI service, and the service then provides this address to interested metrics consumers. An example of a transport mechanism which requires provider action is sockets: the metrics provider would have to wait for and reply to consumer requests for its metrics.

The type of transport is unspecified and completely up to the DCI implementor. A sample of possible transport mechanisms are shared memory, proc file system, Streams, Mach messages, sockets, pipes, remote procedure calls, and files. An implementation could even use more than one type of transport mechanism. An example is the use of system calls to acquire kernel metrics and a user space IPC mechanism, such as shared memory, to acquire application metrics.

It is very important to realise that the type of transport mechanism is invisible to the providers and consumers. What is specified by the DCI is a small set of generic *methods* that providers can use to send their metrics. When a provider registers its metrics in the name space, it specifies the method to be used by that provider to supply metric values. When a consumer queries a metric's value, the particular method registered by the metric provider is invoked to obtain the desired data.

The second service provided by the DCI is maintenance of a metrics name space. The name space consists of a hierarchical structure which lists the metrics that have been registered by all providers. The programming interface specifies the structure of this name space, the provider routines for registering and unregistering new metrics in this space, and the consumer routines to interrogate the name space. Additional information maintained within the name space includes attribute structures describing metric data layout and type, as well as implementation defined access control information.

The DCI Server stores metric attribute structures. These structures describe the characteristics of each metric and are used to allow the metrics to be *self-identifying*. Metrics consumers do not have to have any prior knowledge about metric characteristics such as its data type, units, etc.

The DCI Server stores access control information. Different systems have varying requirements for how strictly system information should be restricted. The specification of a hierarchical name space and the storage and use of access control information in that name space allows the DCI to meet the range of requirements. The choice of how much access control is used is entirely up to the implementation, this specification simply makes access control possible. This

subject is described in more detail in Section 2.5 on page 14, which covers secure implementation strategies.

The metric server's maintenance of a name space implies that a long term relationship has been established between a provider and the server. The extent to which this relationship is preserved in the event of subsystem failure is implementation defined. In particular, the impact of the abnormal termination of the server (or any provider) on metrics that have been registered in the namespace must be defined by the implementation.

2.3 DCI Components

As can be seen in Figure 2-2 on page 8, the entire metrics acquisition architecture consists of:

- a set of metrics providers which, upon request, transmit collections of metrics
- a set of metrics consumers who query for the existence of desired metrics and subsequently fetch metric values
- a library which is part of the provider's or consumer's address space and makes the DCI available
- a DCI Server which provide metric directory services, registration services, and establishes connections between providers and consumers
- consumers may communicate with the DCI Server to alter the configuration of some metrics, or to establish new values. The DCI Server may pass these or other requests to the providers, if the providers are designed to receive them.

The drawn boundary very definitely does not indicate the division of functions between system and user address spaces. Any system specific mechanisms necessary to provide the service functions and metrics transport are not within the scope of this specification. This restriction of the scope of this specification makes it possible to meet the goals of implementation on a wide range of systems and support for application metrics.

Also the existence in Figure 2-2 on page 8 of a box labeled *DCI Server* does not impose a requirement that all server functions reside within a single software module. DCI implementations can divide the server functions between the library and any service provider in the manner appropriate for the target system. As an example, the DCI services could be implemented with multiple DCI Servers, for performance reasons. Alternatively, there could be no DCI Server but the Data Capture services could be implemented entirely in the library routines.

2.4 Metrics Name Space

Metrics are individual units of information. They correspond either to data acquired in a polled manner (for example, statistical information such as *dispatch_count*, *transaction_count*, or configuration/status information such as *number_of_disks*), or to data delivered in the form of an event (for example, thread_termination).

Metrics are grouped into metric classes. For example, all statistics relating to per-thread cpu statistics could be grouped into a single metric class. A metric class is only a template: it must be instantiated by a metric provider before the actual data available for the class's metrics can be identified. For example, the provider of per-thread cpu statistics would instantiate this class for each thread. The thread_id would then provide the additional information required to identify a specific thread's cpu metrics.

The purpose of the DCI metric name space is to establish a unique name for a metric. This name consists of three parts:

- a metric class identifier
- a metric instance identifier
- a metric datum identifier.

A fully qualified metric name consists of a metric *class* identifier, a metric *instance* identifier and a metric *datum* identifier. Once a name for a metric has been registered by a metrics provider, this name can be returned by the library routines which list registered metrics, and can also be used as an argument to routines that return values for the metric. The textual form of the metric name can be written as:

{ {DCIClassId} {DCIInstanceId} {DCIDatumId} }

The metric class identifier names an abstraction, meaning that the name has no physical representation in the system. Metric class identifiers indicate a unique location in the metric class hierarchy. For example, a metric class identifier of {datapool cpu per_thread} might indicate the class of per-thread cpu statistics. This name does not indicate any particular data (since the metric instance identifier, thread_id, has not been specified). Nor is a specific datum indicated (for example, dispatch_count, queue_length). The metric class identifier identifies a class (or *template*, *record* or *struct*) containing metric datum identifiers. This class is instantiated by the provider for a particular set of instance identifiers, that is, threads.

When used with a metric class identifier, the metric instance identifier uniquely identifies a specific metric class instantiation. The instance identifier can correspond directly to some system object, such as a process identifier, a device number, etc. Like the metric class identifier, the instance identifier can have multiple levels. This flexibility allows for multi-dimensional metric classes, for example, per-thread/per-disk I/O statistics. The metric class identifier for such a class might be {datapool io per_thread per_disk}; a metric instance identifier would consist of two parts: a thread_id and a disk_id — for example, {8145 disk0}.

Note that there is a special kind of instance identifier that is used for classes that have a single instantiation. Such classes typically include metrics of a global (or system-wide) nature, for example, refer to the global physical I/O counters class in the Data Pool Definitions specification. Such classes are registered as having UMA_SINGLEINST instance types. The consumer then provides an instance identifier of 0 to reference this single instance. See Section 3.4.1.2 on page 43 for elaboration on the instance level structure (DCIInstLevel); it is this structure that describes the attributes of instance identifiers. See also the **<uma.h>** file for specifics regarding instance types.

When used with a metric *class* identifier and a metric *instance* identifier, the metric *datum* identifier serves to uniquely name a metric. This metric corresponds either with a statistic (for example, dispatch_count), or an event (for example, thread termination). A special value (or wildcard) for the metric datum identifier can be provided to indicate that all metrics within an instantiated class are involved in a particular DCI operation.

Although DCI operations are optimised for transfer of information at the instantiated class level, one can also perform operations at the metric datum (within an instantiated class) level. This ability is used, for example, to wait for individual events or collections of individual events which are not members of the same class. The datum identifier specifies a metric within a fully qualified class and instance identifier.

Consider the following complete name space example illustrating the class ''I/O Device Data'' and subclass ''Disk Device Data''. This example is drawn from the companion specification, **UMA Performance Measurement Data Pool Definition** (see reference **DPD**). The disk device data subclass could have a metric datum that returns ''Number of Blocks Read''. This metric class could be represented symbolically as:

{ datapool io_data disk_data }

or the corresponding numeric name, such as 1.11.2. Note that only a single 4 byte integer is used for each level in the metric class hierarchy.

If one were interested in collecting only the number of blocks read for disk0, then the metric identifier used in the DCI routines must explicitly list the class, instance, and datum. This can be represented symbolically as:

{ { datapool io_data disk_data } { disk0 } { blocks_read } }

Like the metric class identifier, the instance identifier can have multiple levels. Unlike the metric class identifier, each instance identifier level represents an instance of an existing system value and can be multiple bytes long.

Extending the above example, if a machine has a complicated I/O system that consists of a channel/bus/controller/disk hierarchy, then the number of blocks read on the first channel, second bus, first controller, and second drive would be identified by:

{ { datapool io_data disk_data_by_busaddress } { chan1 bus2 cont1 disk2 } { blocks_read } }

These examples serve to introduce but not explicitly define the DCI name space. Refer to the specification in Chapter 3 for more detail.

2.5 Secure Implementation

The issue of secure implementations of this programming interface must be addressed since the problem being solved by the DCI Server is in the class of applications that are most affected by security policy: applications that manage decentralised information on a single system. Also, the design of a secure programming interface cannot be an afterthought since it is exceedingly difficult to retrofit security into an existing design without changing the nature of the design. If we then couple these requirements with the fact that the trend is for higher system security levels to be increasingly important in procurement specifications, the motivation for secure implementations is clear.

There is an important point to keep in mind here. Although this specification describes how to implement a secure version of the programming interface, it does not mandate a particular security level. The choice of how secure a particular implementation should be is for the designer to decide. This programming interface should be flexible enough so that security levels can range from no security checks at all to the highest mandated standards without any modification to the interfaces.

In summary, a secure implementation must be able to allow for discretionary and mandatory access control, and it must prevent the creation of covert timing and storage channels. To this end, this specification adapts a key principle in security: economy of mechanism. This principle means that a system's existing security mechanisms should be used for the implementation of secure access control to metrics. The implication is that this specification will allow for the use of those controls but not specify what those controls are nor how they are implemented. The advantage to this approach, from a security perspective, is that this interface will use known, proven, access control subsystems.

Several features have been added to the design to explicitly support secure implementations. The class/subclass hierarchy fits nicely into a security model. It allows the hierarchy to be ordered from least to most privileged information. There are some side effects to this hierarchical ordering. First, there must be separate metrics hierarchies for each subject. To allow multiple subjects with different access levels to register metrics at the same hierarchical level would create a security nightmare. This implies that the total metrics name space consists of a root, a level containing each class of provider's metrics, and sublevels for the metrics' classes and subclasses.

Each request to obtain or modify metrics, classes or instances is subject to a security check, if the implementation supports it. The consumer's access is checked against the access registered for the requested metrics class and instance. If wildcards are used to request multiple classes or instances then access is checked against the entire branch of metrics.

Another aspect of the design that is affected by security is the requirement that a chosen transport mechanism must be capable of asynchronously notifying the DCI Server that a provider has exited without unregistering metrics. This notification is necessary to allow the server to collect and discard defunct branches of the metrics name space.

From a practical point of view, most designers will choose to use file system access control mechanisms for their secure implementations. The file system used for the metrics hierarchy should be modelled after mechanisms used to implement secure temporary file systems. The latter solves the problem of how to handle file access to the same root by providing a multi-level directory access mechanism. For example, subjects with the highest access level could see all the files in the temporary file system while those with lower levels could only see those files appropriate to their level. (Some secure systems mandate that not only can a subject not have access to an illegal object but it cannot even know that the object exists.)

A final point in this section is that even though an existing file system is used for access control, there is no requirement that the file system also be used for data transport. There is a separation between the need to verify a consumer's access to metrics and how those metrics are delivered from the provider to the consumer.

2.6 Operating System Interaction

On implementations which supply the C library routines **fork**, **exec** and **exit**, certain requirements are made. It is implementation-defined whether a process which is currently using an initialised DCI subsystem and which may have open handles, pending events, or other DCI state, will transmit this state to its forked child. It is implementation-defined whether the forked child will be able to issue DCI calls without error.

A process that overlays itself with a new image by using the **exec** system call does not inherit any DCI state that the previous image had; all DCI handles the previous image had open are closed, instances provided by the previous image are implicitly removed, classes registered by the previous image which have no remaining instances are implicitly unregistered, and all DCI resources that the previous image held are returned to the system. Resources which have been explicitly defined as persistent will not be reclaimed due to the **exec** system call. An example of persistent state which will continue to exist in the system after an image overlayed itself using the **exec** system call, is class attribute information with the DCI_PERSISTENT_CLASS flag set which was registered with the *dciRegister*( ) call.

Upon termination of a process, for example, by using the **exit** system call, all its DCI handles are closed, all its pending event information is lost, all instances provided by the process are implicitly removed, all classes registered by the process which have no remaining instances are unregistered, and all DCI resources that this process held are returned to the system. Resources which have been explicitly defined as persistent will not be reclaimed at process termination. An example of persistent state which will continue to exist in the system after a process terminated, is class attribute information with the DCI_PERSISTENT_CLASS flag set which was registered with the *dciRegister*( ) call.

2.7 Overview of DCI Functions

There are four primary classifications of DCI functions:

- polled metrics consumer functions
- polled metrics provider functions
- event functions
- other functions.

The consumer functions allow metric consumers to traverse the DCI name space, acquire metric class and data attributes, and obtain data. Metric class attributes consist of that data which describes the metrics to be collected, and these are commonly known as *metadata*. Metric class attributes consist of, among other things, a text label, access control information, units, data type, and offset within the collected data buffer.

The provider functions allow metric providers to modify the name space by registering or unregistering metric classes, adding or deleting instances of those classes. These functions also allow providers to communicate with the DCI server to provide polled metrics to consumers. The set of communication functions used depends on the method the provider has registered with the DCI service when adding an instance of a metric class to the name space.

The event functions cover both the provider and consumer categories, and are used to transmit event information. The event functions transmit events directly from provider to consumer.

The other functions consist of configuration requests, security management and metric modification. These requests may be handled entirely by the DCI Server or made visible to the provider, if it has indicated an interest.

2.8 Typical Use of DCI Functions

This section is a tutorial introduction to how the DCI functions can be used by applications. It is intended to illustrate the use of the DCI in support of various performance measurement functions.

As noted in Section 1.2 on page 2, the DCI provides the programming interface for the lowest layer of a metrics architecture. As such, there are many different applications for which the DCI is an important interface. Below are a few examples:

- a simple application reporting a small subset of metric values at a regular interval (as would a DCI-based implementation of the traditional UNIX **iostat**, **vmstat** commands)
- a subsystem provider extending the name space with metrics relevant to that provider's operation (for example, an illustration of how a database vendor could use the DCI to surface transaction metrics)
- an application that starts collection for a subset of events and directs these events to a file for subsequent postprocessing
- a profiling application using events to dynamically track system cpu activity
- an application that navigates through the data of several metric classes to satisfy a particular thread of analysis
- a (UMA) Data Services layer implementation that manages requests for data at different rates for different applications, translates these requests into appropriate DCI calls, manages the collected data and reports the requested information to the appropriate requesters.

2.8.1 Polled Data Acquisition

This simple application collects disk configuration information. Data is collected once every 10 seconds, and interval statistics are computed and printed. The application knows the location of the desired metrics in the name space since the programmer consulted the Data Pool Definition documentation for the target system. The steps this application takes are:

1. Initialise a connection to the DCI subsystem.
2. Open a set of metrics desired and use the resulting handle for the *dciGetData*( ) call. (This handle can be used subsequently to obtain informational messages about new instances that have been created within the metric id list specified.)
3. Obtain the class attributes structure. This will be needed for subsequent parsing of instance structures and returned data for metrics in this class.
4. Poll ''forever'', collecting system statistics. After each poll, the return status is checked to ensure that the call succeeded. If successful, the data is passed along to a routine which will check each return value, and extract the data to be computed and printed. After the data is printed, the data buffer is freed and the poll restarted.
5. When complete (a fatal interruption of the *dciGetData*( ) in the case of this program), close the handle and disconnect from the DCI Server.

Of course, error checking has been minimised to produce a simple coding example.

This example uses macros or function calls which are not part of the API, but if such functionality was available, a sample output for a system with a single disk would look like the following:

```
****** Polled the metrics @ Wed Sep 14 13:44:45 1994
capacity                        c3780
sector_size                      1000
track_size                       c000
addr                             e002
major                               1
minor                            6200
channel_paths                       4
status                              1
vendor                       The Disk Vendor
vendor_designation           The Disk Model
cu_vendor_designation        The Controller Model


#include <sys/time.h>
#include <sys/dci.h>
extern int errno;

extern DCIMetricId *makemetricid();

/* first level */
#define DATAPOOL    2   /* from Data Pool Definition documentation */
/* second level */
#define SYSTEM_CONFIG  11
/* third level */
#define DISK        10

/* macro to extract the classid from a metric id */
#define getDCIClassIdfromDCIMetricId(x) ((DCIClassId*)((char*)(x) + (x)->classId.offset))

main()
{
    DCIMetricId *midp=0;
    DCIClassId *cidp;               /* class for time metrics */
    DCIReturn  *returneddata=0;     /* generic return buffer */
    DCIReturn  *classattrdata=0;    /* classattribute data */
    DCIRetval  *rtvl;               /* individual return status */
    DCIStatus status;               /* return status of functions */
    DCIClassAttr *classattr;        /* pointer the class attributes */
    DCIHandle handle;               /* descriptor returned from dciOpen */
    int class[] = { DATAPOOL, SYSTEM_CONFIG, DISK}; /* metric class desired */
    int polling = 1;        /* if 1, continue to poll */
    void *thedata;          /* pointer to the returned data */

    /* initialise the connection to the DCI Server */
    status = dciInitialize(DCIVersion *) NULL, (DCIVersion *) NULL);
    if (!(status & DCI_SUCCESS)) {
        dciPerror (status, errno, 0, "dciInitialize failed");
        exit(1);
        }

    /* make a metric id using an application provided function and
     * the specified class. The metric id will contain wildcards
     * in all instance levels.
     */
    midp = makemetricid(class, 3);

    /* open up the metric. Let the library allocate the return data buffer */
    status = dciOpen(&handle, midp, 1, &returneddata, 0, 0);
    if (status & DCI_FATAL)
        goto quit;

    /*
     * obtain the class attributes. These are used to extract particular
```

```
     * metrics when a whole class of metric data is returned. The DCI
     * server will automatically allocate the correct size return
     * buffer. Extract the actual class attribute super structure
     * from the request returned data.
     */
    cidp = getDCIClassIdfromDCIMetricId(midp);
    status = dciGetClassAttributes(handle, cidp, 1, &classattrdata, 0);
    if (status & DCI_FATAL)
        goto quit;
    rtvl = (DCIRetval *)(&classattrdata->retval);
    classattr = (DCIClassAttr *)((char *)rtvl + rtvl->dataOffset);

    while (polling) {
        /* free the return buffer for reuse */
        dciFree(returneddata);  returneddata = 0;

            /* poll for the data placing the data and return
             * status in the same buffer.
             */
            status = dciGetData(handle, midp, 1, &returneddata, 0,
                        0, 0, (UMATimeVal *)0);

            if (status & DCI_FATAL) {
            /* application provided error printing call */
            dciPerror(status, errno, 0, "dciGetData failed");
                goto quit;
            }

            /* the poll was successful. Call a compute-and-print
             * routine.  The routine takes the class attribute
             * structure and the data returned and derives the
             * data desired.
             */
            rtvl = (DCIRetval *)(&returneddata->retval);
            if (rtvl->dataSize) {
                thedata = (char *)((char *)returneddata + rtvl->dataOffset);
                computeandprint(classattr, thedata);
            }
            sleep(10);      /* the polling rate */
    }

quit:
    /* free any buffers the library allocated, close the handle and
     * shutdown the connection to the DCI Server.
     */
    if (midp)
        dciFree(midp);
    if (returneddata)
        dciFree(returneddata);
    if (classattrdata)
        dciFree(classattrdata);
    status = dciClose (handle);
    dciTerminate();
    exit(0);
}


/* create a metric Id for the given class with wildcarded datumId
 * and instances.  Note that memory allocation has been trivialised
 * for the example. The caller must free the metric id when done.
 */
DCIMetricId
*makemetricid(int *classarray, int numclasses)
{
    DCIClassId *cidp;
```

```
    DCIInstanceId *iidp;
    DCIMetricId *midp;
    int size, i;

    /* for ease of example, overallocate a DCIMetricId structure */
    midp = (DCIMetricId *)malloc(128);
    if (!midp)
        return(midp);

    /* fill in the classid, using macros not part of the DCI API.
     * The data for the classid will be appended to the end of the
     * base DCIMetricId structure.
     */
    midp->classId.offset = sizeof(DCIMetricId);
    cidp = (DCIClassId *)((char *)midp + midp->classId.offset);
    cidp->identifier.offset = sizeof(DCIClassId);
    cidp->identifier.count = numclasses;
    cidp->size = dcisizeof (cidp) + (numclasses * sizeof(UMAUint4));
    for (i=0;i<numclasses;i++)
        dciclassidlevel(cidp,i) = classarray[i];

   /* fill in the instanceid, using macros not part of the DCI API */
    midp->instanceId.offset = midp->classId.offset + cidp->size;
    iidp = (DCIInstanceId *)((char *)midp + midp->instanceId.offset);
    iidp->inputMask = DCI_ALL_INSTANCES;    /* wildcard overrides actual instances */
    iidp->outputMask = DCI_ALL;
    iidp->size = sizeof(DCIInstanceId) + 4;

    /* fill in the metric id now */
    midp->datumId = DCI_ALL;        /* all metrics */
    midp->size = 128;               /* the size allocated */
    return(midp);
}

/* Takes a class attribute structure and the data and prints the
 * current values.  For ease of example, assume all the data is
 * 4 bytes in size (except for textstrings) -- normally one
 * must check the data type and correctly print the data.
 */
computeandprint(DCIClassAttr *ca, char *databuf)
{
    int numdatums, i, currentdata;
    char *labeltext, *currentstr;
    struct timeval tv;
    DCIDataAttr *da;

    gettimeofday(&tv, (struct timezone *)0);
    printf("****** Polled the metrics @ %s", ctime((time_t *)&tv));

    /* number of datums in the class */
    numdatums = ca->dataAttr.count;
    da = (DCIDataAttr *) dciclassattrdataattr(ca);

    /* for each datum, print the label and current value of data */
    for (i=0; i<numdatums; i++) {
        labeltext = (char *) &
            ((UMATextString *)(dcilabelascii(dcidataattrlabel(da))))->string;
            currentdata = *(int *)(databuf + da->offset);
            if (da->type == UMA_TEXTSTRING) {
                /* the offset actually points to a UMATextString */
                currentstr = (char *) &
                    ((UMATextString *)(databuf + currentdata))->string;
                printf("%-25.25s    %s0, labeltext, currentstr);
            } else {
                printf("%-25.25s    %8x0, labeltext, currentdata);
```

```
                }
                /* next data attribute */
                da = (DCIDataAttr *)((char *)da + da->size);
        }

        return(0);
}
```

2.9 Possible Implementation Strategies

There are three major decisions to make when implementing the Data Capture Interface. The implementor must decide:

- the degree of centralisation
- the underlying transport mechanisms
- the security level.

These three factors, centralisation, transport, and security, dictate the size and difficulty of the implementation.

Firstly, consider centralisation. The initial architecture diagram shows all interactions going through a centralised DCI Server; however this is not an implementation requirement. The specification consists of the routines provided at the boundary layer and the behaviour of the underlying transport mechanism, and not the implementation details. Figure 2-3 shows an implementation which uses a library to implement the programming interface. The outermost provider/consumer pair are using the DCI Server to list available metrics or register new metrics. The innermost pair are transmitting metrics in a connection that was previously established by information provided by the server. Note that the location of the DCI Server is not specified. It could be implemented in either user or system space, or in a combination of the two. An implementation could go further and eliminate a centralised DCI Server, performing all services as part of library routines.

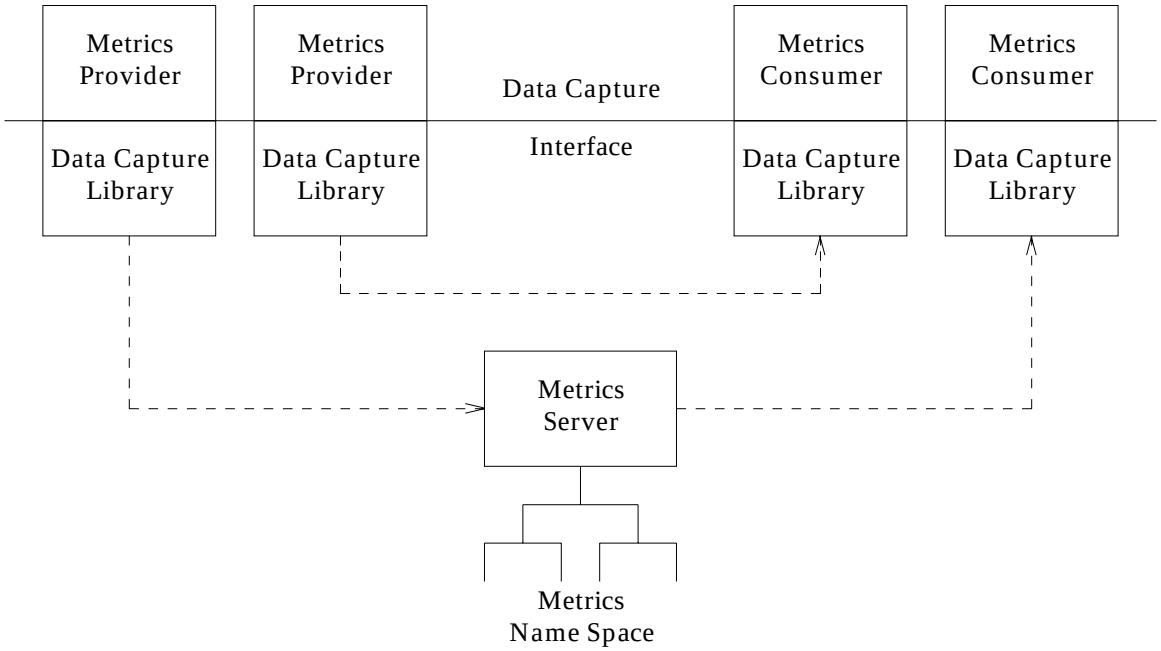


Figure 2-3 Decentralised DCI Implementation Example

When implementing the DCI, one or more underlying transport mechanisms must be chosen. The choice of transport can be any system specific communications mechanism, such as sockets, shared memory, shared files, and various IPC mechanisms. When choosing a transport mechanism, the designer must consider the number of concurrent operations, speed, portability, if the implementation is intended for more than one operating system type, and the ability to

detect inadvertent provider and consumer termination. The latter feature is important if the implementation saves connection related state, such as whether a consumer passed access control.

Finally, an implementor must choose the implementation's security level. This has the largest effect on a secondary choice, the DCI name space implementation. As mentioned in Section 2.5 on page 14, secure implementations must implement the name space using a previously implemented secure facility, such as a file system. If the implementor chooses a lower security level, then the name space can be implemented without this constraint.

Chapter 3

Overview of the DCI Specification

This is the specification for the Data Capture Interface. The material in this chapter is authoritative and overrides any possibly conflicting material in the first two chapters of this document. This and the next chapter cover the following material:

- definitions for the metrics name space
- definition of metric class attributes, including data type and unit values
- a summary of the routines and a guide for subset implementations
- a list and explanation of routine status values
- descriptions of the basic routines used by metrics consumers
- descriptions of the basic routines used by metrics providers
- descriptions of the extended routines for event support.

3.1 Conventions

3.1.1 Naming Conventions

This section follows a naming convention for the three types of programming language constructs defined here: constants, type definitions, and routine names. (The rules followed are the same as those followed by the MLI layer (see reference **MLI**).) The purpose of this convention is to clearly identify the Data Capture facilities in a program, to determine what type of object a particular DCI name represents, and to avoid name space conflicts.

To this end:

- constant names have an uppercase ‘‘DCI_’’ (or ‘‘UMA_’’) prefix followed by an all uppercase name
- type definition names have an uppercase ‘‘DCI’’ (or ‘‘UMA_’’) prefix followed by a mixed case name
- DCI library routines have a lowercase ‘‘dci’’ prefix followed by a mixed case name.

An additional naming convention is that only English alphabetic characters are used. For type definitions and routine names, non-alphabetic characters, such as underscore, are not used as delimiters within a name.

3.1.2 Data Type Conventions

A set of data types has been defined for the DCI. The primary purpose of these data type definitions is to ensure that different DCI implementations support the same data types. All DCI data types are defined in the **<uma.h>** header file in **Part 2** of this publication; they have the uppercase ‘‘UMA’’ prefix, followed by a mixed case name (for example, UMAInt4, UMAUint4, UMAInt8).

3.1.3 Treatment of Variable Length Structures

Many structures in the Data Capture Interface have variable length. For example, metric class identifiers are unconstrained in the number of levels. Further, many variable length structures may themselves be contained in other variable length structures. The DCI has adopted a consistent mechanism for the layout and description of such structures.

Most DCI structures have the same organisation:

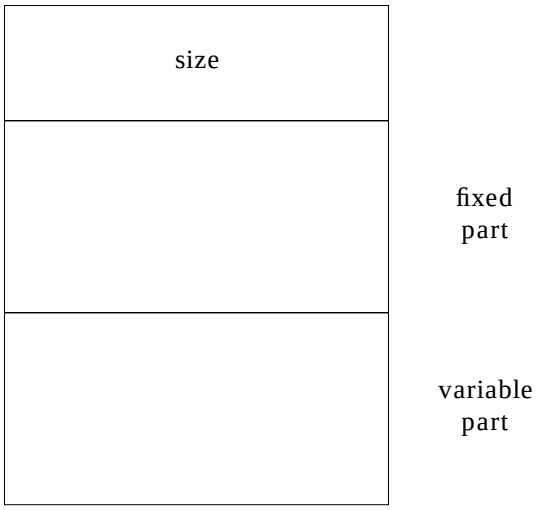


Figure 3-1 DCI Structure Organisation

Size is the size of the entire structure (including any variable part). The fixed part contains only simple fixed size entities (for example, UMAUint4, DCIDatumId), as well as fixed sized descriptors for any variable length data in the rest of the structure (for example, UMAVarLenDescr). The variable part contains all variable data. It may include many variable structures, each of which is described by a fixed size descriptor in the fixed part of the overall structure.

There are no constraints on the placement of the different variable length structures within the variable part of the overall structure; each descriptor ''points'' to its corresponding variable part using an offset mechanism - every descriptor contains an *offset* field which is a byte offset to be applied to the base of the structure in question. Thus the *offset* field is always a small positive integer. Because the offset is added to the base address of the structure in question and then indirected, it must always produce a correct address within the structure. Variable length data which is optional must have a valid offset to data of 0 size to prevent improper address indirection.

There are different types of descriptors for different types of variable length entities (for example, array of fixed size elements, string, array of variable length elements). Each of these descriptor types is presented in the **<uma.h>** file, and DCI structures are presented in the **<dci.h>** file. Each file is a valid/compilable C-language include file. In each of these structures, the variable part is described uniformly as being of type UMAVarLenData; this is simply a placeholder for the entire variable section.

It is worthwhile noting two interesting properties of variable sized DCI structures that result from using descriptors. First is that the order of the data in the variable part of a structure may not be the same as the order of the fixed sized fields that point to it. So for example, if a structure contains descriptors *field1* and *field2*, in that order in the fixed part, it is valid for the data pointed

at by *field2* to precede the data pointed at by *field1* in the variable length area. (It is NOT valid for fields to overlap.) Second is that it is also valid for the variable part of a structure to contain ‘‘holes’’ which are never referenced. A hole would be accounted for in the size field of the enclosing structure, but would not be described by any of the descriptors. If individual variable length structures are properly located using the *offset* field located in their descriptors (located in the fixed part of the overall structure), both these interesting properties will be transparent.

Note that the foregoing discussion applies to most variable length structures; there are some exceptions involving primitive structures (for example, **UMATextString**) that do not require the flexibility of the offset mechanism, and thus are optimised to be as small and simply-organised as possible.

3.2 Metrics Name Space

The Data Capture Interface supports a hierarchical name space that is used to uniquely identify specific instances of available metrics. Metric providers make additions and deletions to the contents of the name space as they make metrics available and unavailable to metrics consumers. Metric consumers query the name space to obtain metric names; these names are used to obtain actual metric values.

The other purposes of this name space are to provide a mechanism for storage of the descriptions of what type of data is available, the metric class attributes, and to provide a hierarchical structure suitable for implementing an access control mechanism.

The metrics name space is grouped into two layers: one layer to identify a metric class, or metric datum within a metric class, and another layer to identify the available instances for a metric class. The first layer is represented by the metric class identifier combined with the metric datum identifier. The second layer is represented by the metric instance identifier. Together the two name space layers uniquely identify a metric instance. The components associated with this name space are the DCIClassId, DCIInstanceId, DCIDatumId. The DCIMetricId is a combination of all three components.

File systems typically store any of their supported object types, such as files or devices, at any position in their name space. The DCI imposes more structure upon its name space by identifying different levels with different object types and restricting the positions at which objects can be stored. The reason behind these restrictions is that the DCI name space is not a general purpose file system but is intended to support a very specific purpose. The structure and function of the DCI name space are described in the following sections.

3.2.1 Mapping DCI Name Space to a Network Representation

One of the primary goals of the DCI name space design is that it map well to a network representation. This section describes how one might approach such a mapping and how this approach affected the name space design.

A fully qualified metric name consists of a metric class identifier, a metric instance identifier and a metric datum identifier. A metric class identifier is a sequence of class levels, where each class level is a 4 byte integer. A metric instance identifier is a sequence of instance levels, where each instance level is a multiple of 4 bytes. Instance levels are intended to represent *natural* values found within a system; as such, they are not arbitrarily limited in size or range. For a given class, however, the number of instance levels is defined, as is the size of each instance level.

The DCI interfaces allow metrics to be selected individually, or as a group within an instantiated class. The metric datum identifier is a 4 byte integer, with values 0 and 2^32 - 1 reserved. Thus, there are at most 2^32 - 2 unique metric data within any instantiated class.

To summarise, the metrics name space features:

- subdivision of name space into a class/datum identifier hierarchy and instance identifier hierarchy
- reserving values 0 and 2^32 - 1 in the class/datum name space for use as a delimiter and wildcard, respectively.

3.3 DCI API Data Types

DCI structures do not need the delimiter necessary in the network representation as they can rely on the structure boundaries themselves. The C structures are presented first and then described in subsequent sections. The detailed descriptions include code fragments for how these structures are used.

The structures are defined as follows:

3.3.1 DCIClassId

```
typedef struct DCIClassId {
    UMAUint4        size;
    UMAVarLenData   data;
} DCIClassId;
```

The usage of the structure elements is as follows:

size The total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

data Variable length data

3.3.2 DCIDatumId

```
typedef UMAUint4 DCIDatumId;
```

The usage of the structure elements is as follows:

DCIDatumId Datum identifier

3.3.3 DCIInstanceId

```
typedef struct DCIInstanceId {
    UMAUint4        size;
    UMAUint4        inputMask;
    UMAUint4        outputMask;
    UMAVarLenData   data;       /* Type: defined by DCIClassAttr */
} DCIInstanceId;
```

The usage of the structure elements is as follows:

size The total nutotal number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

inputMask A bitmap indicating which levels are included in this instance id. Only included levels will have instance identifiers in the data area of this structure. Levels not included (i.e., those for which their corresponding inputMask bit is off, set to 0), are interpreted as wildcarded. Bit 0 corresponds to the first instance level registered for the class; bit 1 for the second, and so on.

outputMask A bitmap indicating which levels in the instance id to be filled in by the provider / DCI Server. At least one level must be selected (set to 1).

data Variable length structure holding the single instance identifier. The instance identifier contains a sequence of instance levels; the number of instance levels and the size of each instance level can be found in the DCIClassAttr structure

3.3.4 DCIMetricId

```
typedef struct DCIMetricId {
    UMAUint4        size;
    UMAVarLenDescr  classId;    /* Type: DCIClassId */
    UMAVarLenDescr  instanceId; /* Type: DCIInstanceId */
    DCIDatumId      datumId;
    UMAVarLenData   data;
} DCIMetricId;
```

The usage of the structure elements is as follows:

size The total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

classId Indirection to class identifier (variable size)

instanceId Indirection to instance identifier (variable size)

datumId Datum identifier

data Variable length data (DCIClassId and DCIInstanceId structures)

3.3.4.1 Use of DCI Name Space Structures

The following diagram gives an example of how the name space structures are used. The solid boxes identify individual classes and the dashed boxes instances of those classes. The root of this DCI name space is labeled metrics and is an uninstantiated class. There are two labeled subclasses, one being the data pool and the other an application. The data pool is a predefined group that lists operating system metrics used to describe system behaviour. See the companion document, Data Pool Definitions (see reference **DPD**), for a list of those metrics.

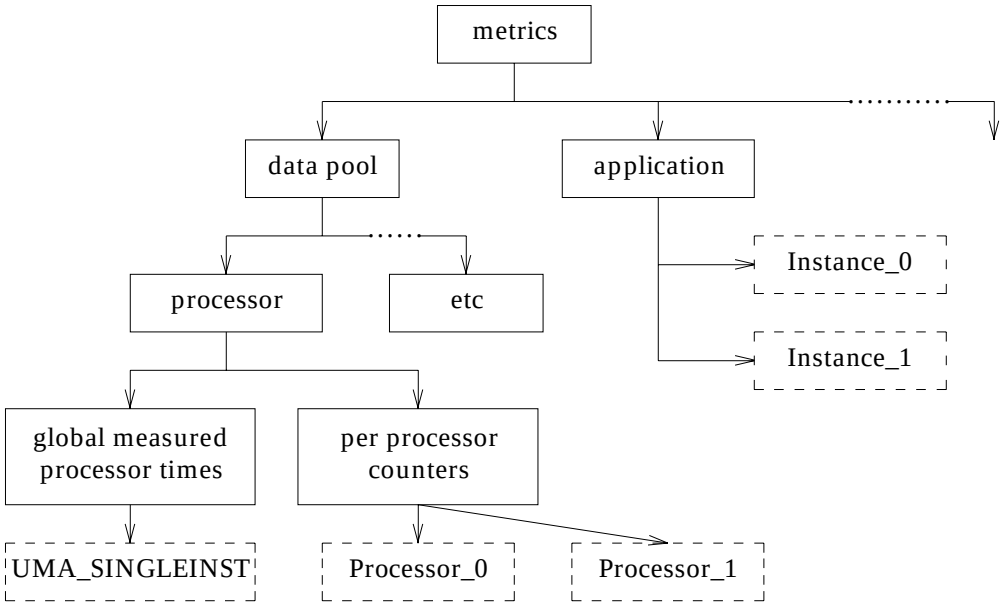


Figure 3-2 Name Space Example

The Data Pool is organised as a class/subclass hierarchy. In Figure 3-2 there is a processor class with two sample subclasses, global measured processor times and per processor counters. In

this example, there are two processor instances of the *per*processor*counters* class, but only a single instance of the *global*measured*processor*times class. In the DCI, a class with a single instance is specified with an instance type of UMA_SINGLEINST. The DCI name space class hierarchy is not limited to two class levels, as might be surmised from the accompanying diagram. This is merely a Data Pool convention.

Note that there are measurements available for two instances of the application metric class. These are probably differentiated by the process identifier of each running copy of the application. There is an important point here. The DCI name space class hierarchy is abstract and represents the organisation of metric classes. The DCI name space instance hierarchy represents system values and uses those values directly. In the example, the application metric class can be instantiated by process identifier and the Data Pool's processor metrics by process number. Instantiated classes are simply those classes which can support instances. Uninstantiated classes are simply positions in the name space and provide its hierarchical structure. An instantiated class cannot have subclasses.

Another point that can be drawn from this example is that the DCI is optimised to work with entire metric classes. One can specify individual metrics within an instantiated class by using an explicit value for the *datumId* in the **DCIMetricId** structure, but it is expected that this use will be the exception rather than the rule. Usually a single wildcarded datumId would be specified to efficiently obtain all the metrics available within an instantiated class.

The positioning of the datumId below the instance in the name space implies that metrics are consistently provided across all instances of a class. In general, this will be the case. However, there are situations in which it is not possible for certain instances of a class to provide all metrics within the class. As an example, if a subclass were used for disk metrics, one might provide part of the subclass for one disk instance and a different part for a second disk instance.

3.3.5 DCIDatumId Reservation

3.3.5.1 Special polled and event metrics

The DCIDatumId is a UMAUint4 (32 bit) quantity. Certain values of the DCIDatumId are reserved for special polled metrics and event metrics; further, several DCIDatumId values are reserved for vendor use:

```
Reserved Polled metric          DCIDatumId value
DCI_INVALIDDATUMID              0x000000ff

Reserved Event metric           DCIDatumId value
DCI_FINALDATA_EVENT             0x000000f8
DCI_INSTANCEADDED               0x000000f7
DCI_INSTANCEREMOVED             0x000000f6
DCI_DATACHANGED                 0x000000f5

Reserved for vendor use         0x000000e8 - 0x000000ef
```

3.3.5.2 Support for derivation of metrics

Measurement application developers frequently need to derive reported metrics from the raw metrics available on a system. While the key focus of the DCI is to provide access to the raw metrics, it also supports such derivation through an encoding of the DCIDatumId. This encoding allows metric providers to record relationships between metrics within a class. These relationships can then be interpreted by consumers.

A derived metric is defined by explicitly encoding the manner of its derivation in the derived metric's DatumId. Each derived metric holds a place in the class namespace.

The types of relationships that can be expressed include:

Sum The derived metric is obtained by adding two other metrics in the class.

Ratio The derived metric is obtained by taking the ratio of two other metrics in the class.

See *Data Types* and *Measurement Units* for a complete description of derived metric support.

Note that derived metric support is limited to the encoding of inter-metric relationships. Each derived metric holds a place in the namespace, but there is no derived data available through the DCI. No actual derivation occurs within the DCI; this remains the responsibility of the consumer.

3.3.6 DCIMetricId Code Sample

A completely named metric consists of the DCIClassId, the DCIInstanceId within that class, and the DCIDatumId. The DCIMetricId contains the size of the sum of the individual parts, so traversing a list of DCIMetricIds becomes a simple task (although complicated by casting requirements due to the variable size of the DCIMetricId structure). The following code performs a traversal of a list of DCIMetricIds having length metricIdListCount:

```
DCIMetricId     *aMetricIdPointer;
DCIMetricId     aMetricIdList[];
int             metricIdListCount;


aMetricIdPointer = aMetricIdList;
while (metricIdListCount--)
{
    ... do something with the current metricId

    aMetricIdPointer =
            (DCIMetricId *) ((char *)aMetricIdPointer
            + aMetricIdPointer->size);
}
```

3.3.7 DCIClassId Code Sample

Although defined as a fixed size structure, the actual metric class identifier is a variable size 4 byte integer array. The DCIMetricId contains a reference to this array by offset, element size and number of elements. The array follows the structure directly, as started by the variable length data declaration.

Thus, to walk the integers in a DCIClassId, the following code fragment could be used:

```
DCIClassId     *aClassId;
int             i, numlevels;

numlevels  = dciclassidlen(aClassId);
for (i = 0; i < numlevels; i++) {
{
    ... do something with dciclassidlevel(aClassId, i)
}
```

Note that the size of each array element is always four bytes.

3.3.8 DCIInstanceId

A single metric instance identifier is enclosed in the DCIInstanceId structure. A metric instance identifier is used to select one of otherwise identical groups of metrics. The metric instance identifier is used to distinguish among several instantiations of objects such as disks, processors, processes, etc. Thus a metric instance identifier is closely associated with the object being measured.

The DCI does not reserve any instance values to represent wildcards and therefore the DCIInstanceId contains a bitmask to indicate wildcarding of a particular level in a metric instance identifier. Since these flags are represented as a bitmap in a 4 byte integer (UMAUint4), the maximum number of levels for a metric instance identifier is 32. The bitmap indicates which instance levels are explicitly included in the DCIInstanceId structure; those not explicitly included (i.e., those for which their corresponding bit is off) are wildcarded. The bitmap starts at bit position 0 and the metric instance identifier levels start at level 0.

Figure 3-3 shows the layout of the **DCIInstanceId** structure.

The size of the entire DCIInstanceId structure and all its associated variable length data is stored in the *size* member of the DCIInstanceId structure. Instance levels within a single metric instance identifier can vary in size, but are always a multiple of 4 bytes. The actual information about the instance levels is encoded in the DCIClassAttr structure defined later in this document.

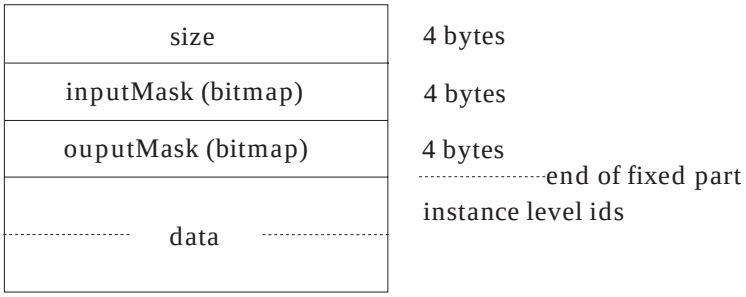


Figure 3-3 DCIInstanceId Diagram

3.3.9 DCIInstanceId Structure Examples

The DCIInstanceId structure allows applications to efficiently specify the following:

- single instance values with multiple levels
- single instance values with wildcarded levels. The number of levels and the size of each level is wildcarded. This wildcard is represented by inputMask = DCI_ALL_INSTANCES. It is essentially a requirement whenever a wildcarded class is presented in a DCIMetricId. In this case, one cannot know if the number of instance levels in all classes matching the wildcard will be identical.

As an example, consider a metric class which is instantiated on a per-processor, per-disk basis and that both these metric instance identifier level types are four byte values. Specifying the metric instance identifier for processor 1, disk 2 is as shown in Figure 3-4, assuming that the instance values for processor1 = 1, disk2 = 2. The first instance level value (level 0) specifies the processor and the second one (level 1) the disk. Note that the outputMask is set to 2^32-1 indicating that all instance levels are to be filled in when this DCIInstanceId is returned in any

DCIReturn structure.

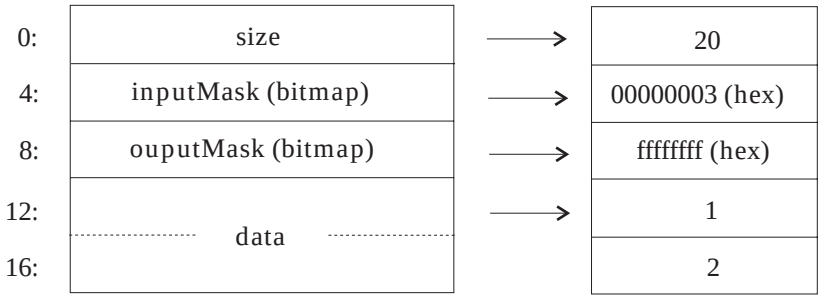


Figure 3-4 DCIInstanceID: Two Instance Levels

Note that the structure sizes would not change if both levels in the above instance were specified as one byte values. There is an implicit four byte padding since the single byte instance must be written to a four byte aligned instance field.

Using the above example, specifying all disks for processor 1 would be as shown in Figure 3-5.

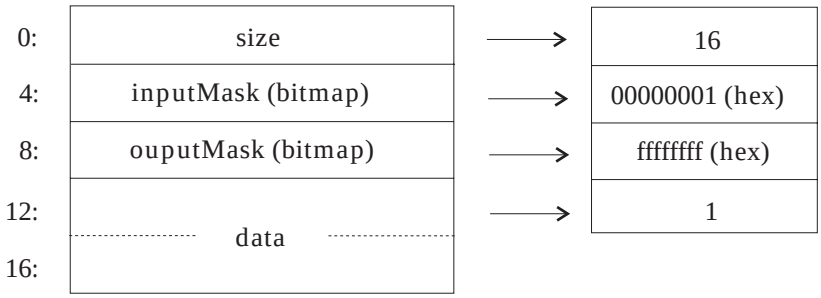


Figure 3-5 DCIInstanceId: Two Instance Levels, Wildcarding

Note that the second bit (bit 1) in the inputMask is set to 0, indicating that level 1 is wildcarded. The value for level 2 must not be specified. Note that the instance id structure is smaller than the previous example.

3.3.10 Wildcards

Wildcards are used with all three name space levels, classes, instances, and individual metrics. The DCI name space class hierarchy reserves the integer value of 2^32 - 1 (all bits in the integer are set) as the "DCI_ALL" wildcard value for any particular class level. This convention is used to specify groups of metric classes. Any integer in a metric class identifier can be wildcarded. In the case of instance ids, wildcards will always expand into valid instances without error, unless no instances can be found for the class.

Similarly, metric instance identifier levels can be wildcarded. As seen in the previous section's examples, this wildcarding is via a different mechanism than the class identifier. This is because no field of an instance can be reserved for a wildcard value. The wildcard must be specified outside of the instance range. Note that instance levels are wildcarded by setting bits in the inputMask to 0, and that there is a special "DCI_ALL_INSTANCES" inputMask value that wildcards all instance levels.

As noted in Section 3.3.9 on page 34, the metric datum identifier uses the same wildcard as the metric class identifier, the value 2^32 - 1 ("DCI_ALL"). When this wildcard is used it indicates all metrics in the instantiated class. This also signals that all data is presented as a single entity, rather than broken out by datum identifier separately, as is explained in Section 3.4.1.3 on page 45.

3.3.11 Access Control

One of the purposes of a hierarchical name space is to enable the implementation of secure access control mechanisms. This is done by storing access control information at the nodes of the hierarchy. This information can then be used by the DCI Server in cooperation with the underlying operating system to provide discretionary and mandatory access control. Access control can be performed at both the class and instance levels. The DCI access control data type is:

```
typedef struct DCIAccess {
    UMAUint4        size;
    UMAVarLenData   access;
} DCIAccess;
```

The usage of the structure elements is as follows:

size The total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

access Byte array containing the access information.

The purpose of defining a variable sized field structure is to allow for the wide variation in access control structures between operating systems and differing security levels. A DCIAccess structure for standard UNIX might have a size of twelve and the access field of DCIAccess would store the access mode and the owner's group and user id. A system with a more complicated access control system may store variable access control lists in a DCIAccess structure.

The degree of access control is entirely up to the implementation of the DCI Server. Some implementations may choose for performance reasons to support absolutely no access control. These implementations would not bother to store or check access control information. A higher level of access control may store and check access structures only at the class level. The most pervasive implementations would store a DCIAccess structure at both the class and instance levels. There is no access control at the metric datum identifier level.

Highly secure implementations may go further and layer the DCI name space on existing, secure file systems in order to conserve mechanism and provide complete mandatory and discretionary access control. Again, the choice of how much access control is entirely up to the implementation and should be documented locally.

3.4 DCI Name Space Attribute Structures

The purpose of name space attributes is to, as much as possible, allow the information stored in the name space to be self describing. These descriptions are registered by the metrics provider when adding metric classes and instances. The objects in the DCI namespace that must be described to a consumer are metric classes, instantiations of those classes (instances), and the characteristics of the data associated with each individual metric.

The attribute structures are kept separate from the data itself for performance reasons. It is expected that many DCI applications will be reading the same data multiple times so it makes sense to read the invariant information once, put it aside, and then read the changing data as much as the application requires.

There are two main attribute structures:

DCIClassAttr

Describes the attributes of an entire class. This includes some flags, a label, and implementation defined access control information.

The metric class attributes also describes the characteristics of the data associated with each metric datum contained in the class definition. Two substructures are employed for this purpose: the DCIDataAttr substructure describes the characteristics of metric datums that correspond to polled metrics; the DCIEventAttr substructure describes the characteristics of metric datums that correspond to events.

In particular, the DCIDataAttr substructure describes the datum id, the data type, the data units, and a label for each metric. Further, since most DCI routines return entire instantiated classes (which contain the data for all polled metrics in each instance), the DCIDataAttr substructure also indicates an offset within an entire return structure at which each individual polled metric's data can be directly located.

For events, the DCIEventAttr substructure describes the datum id, and the label. In addition, events can return additional data (for example, an event associated with a timer interrupt could pass the current value of the program counter; such information could be postprocessed to obtain a profile of application (or system) behaviour). Each event can have several pieces of additional data returned. The characteristics of such event data is described through the inclusion in the DCIEventAttr substructure of one DCIEventDataAttr substructure for each such piece of data to be returned when the event actually occurs. These event specific structures are described in Section 3.4.3 on page 47.

The number and type of instance levels with which this class is to be instantiated is also described in this attribute structure (for example, is this a per-thread class, a per-processor and per-disk class, etc.).

The DCIClassAttr structure can be extended with implementation specific information (termed, **local extensions**). One possible use for this optional section is a text formatting string for displaying class information.

DCIClassAttr structures are created by metric providers, and registered with the DCI Server at class registration time.

The DCIClassAttr structure for a particular metric class can be obtained with the DCI call *dciGetClassAttributes*( ).

Details of the DCIClassAttr structure are presented in Section 3.4.1 on page 39.

DCIInstAttr

Describes the attributes for a specific instantiation of a class (an instance). This descriptive data includes the instance label, and implementation defined access control information. As

with the metric class attribute structure, the instance attribute structure is extensible with an implementation specific (optional) extension.

DCIInstAttr structures are created by metric providers at instance registration time; they are included with the structure describing the instance method.

The DCIInstAttr structure for a particular instance can be obtained with the DCI call *dciGetInstAttributes*( ). Details of the DCIInstAttr structure are presented in Section 3.4.2 on page 47.

3.4.1 DCIClassAttr

The DCIClassAttr structure describes the attributes of a class. These class attributes are set by the provider of a class in the call to *dciRegister*( ) that registers the class and cannot be changed unless the class is unregistered with the *dciUnregister*( ) call. They may be retrieved by any consumer with the proper access rights using the *dciGetClassAttributes*( ) function.

The DCIClassAttr structure provides a label for the class, defines access control information for the class, describes every individual metric supported by the class, and describes the structure of the class's instance space. It also provides for local extensions to the class attributes structure. Several of the fields contained in the following definition have variable size.

```
typedef struct DCIClassAttr {
    UMAUint4            size;
    UMAUint4            flags;
    UMAVarLenDescr      access;
    UMAVarLenDescr      method;
    UMAVarLenDescr      label;
    UMAArrayDescr       instLevel;
    UMAVarArrayDescr    dataAttr;
    UMAVarArrayDescr    eventAttr;
    UMAElementDescr     extensions;
    UMAVarLenData       data;
} DCIClassAttr;
```

The usage of the structure elements is as follows:

size The total number of contiguous bytes of storage in the structure and all its associated variable length data, and must be a multiple of 4.

flags One or more bit mapped flags with the following possible values:

DCI_ENABLED
: This class is enabled and available for use. Absence of this flag means the class is disabled.

DCI_NOTIMPLEMENTED
: This class is unimplemented.

DCI_NOTAPPLICABLE
: This class is not applicable to the measured system and hence not available.

DCI_OBSOLETE
: This class is being phased out and is not available.

DCI_PROVIDER_INSTANCE
: The provider for this class does not register instances with the DCI Server. When a list of instance is requested by a consumer, the Server must request that list directly from the provider using the class method. If this flag is set, it is an error not to specify a class method also.

DCI_PERSISTENT_CLASS
: The class should not be removed from the namespace (that is, unregistered) when the process that registered this class terminates (for example, issues the exit( ) system call). (See Section 2.6 on page 16, "Operating System Interaction" for a discussion of persistence.)

DCI_POSSIBLEINVALIDDATA
: This may return invalid data for some metrics for some instances. The application must check the DCI_INVALIDDATUMID metric for a list of other class metrics which are not valid.

access Descriptor to the initial access control information for the class.

method Descriptor to a DCIMethod structure for a class method. If the size of this structure is zero, then there is no class method.

label Descriptor for a variable length ASCII and internationalised text label describing this class. If no label is desired, this must still contain a valid zero length string.

instLevel Descriptor for a variable length DCIInstLevel structure that describes each instance present in this class.

dataAttr Descriptor for an array of variable length DCIDataAttr structures, each representing an polled metric supported by this class.

eventAttr Descriptor for an array of variable length DCIEventAttr structures, each representing an event metric defined in this class.

extensions Descriptor for a variable length implementation specific block of class information.

data Data section for all variable length information in this structure.

The relationship between the various fields in this data structure is illustrated in the DCIClassAttr diagram in Figure 3-6.

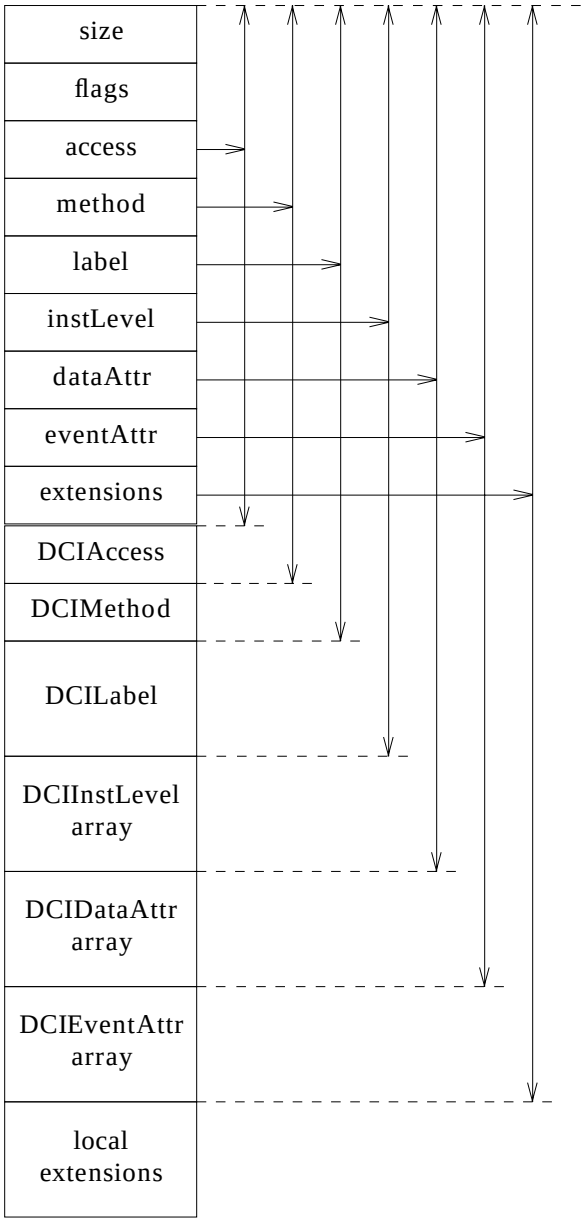


Figure 3-6 DCIClassAttr Diagram

3.4.1.1 DCILabel

The label attributes structure contains two forms of the label. The first is a null terminated ASCII string and the second is an internationalised description. Both variable sized fields must be padded out to a four byte boundary.

The DCILabel structure is:

```
typedef struct DCILabel{
    UMAUint4            size;
    UMAVarLenDescr      ascii;
    UMAElementDescr     i18n;
    UMAVarLenData       data;
} DCILabel;
```

The usage of the structure elements is as follows:

size Total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

ascii Descriptor for the variable length UMATextString for the ASCII label.

i18n Descriptor for the variable length data for the internationalised description.

data Data of DCIlabel for ascii and i18n.

The internationalised label indicated with the i18n descriptor is optional, and the internal structure (of the variable length contents) is unspecified because of the wide range of structures used to describe internationalised text. A typical example of the contents of the i18n label would be the name of a message catalogue and the index number of a message in that catalog.

There are some important points about DCILabel. One is that once a metric class, instance or datum has been labeled then the metric provider cannot relabel it unless the metric has been completely unregistered. Unregistering a metric class or instance can be detected by consumers. See the *dciOpen*( ) and *dciClose*( ) routine descriptions for how this is done.

In order for applications to search the name space using a semantically meaningful label (rather than the metric identifier), these labels should be unique, that is:

- metric class labels should be unique within each level of the DCI name space class hierarchy
- instance labels should be unique within each level of the DCI name space instance hierarchy
- individual metric (datum) labels should be unique within the class.

This ability to search the name space by label implies that providers must have registered a label with each name space entity (metric class, metric instance, metric datum), and that these labels should obey the uniqueness criteria outlined above. Maintaining unique labels is the responsibility of the entity that registers the class or adds a new instance. No DCI library function will return an error status if the label is not unique.

The following characters are reserved and should not be used in the ascii component of the DCILabel:

```
!       (exclamation)
@       (at sign)
#       (hash, pound sign)
$       (dollar sign)
%       (percent)
^       (caret)
&       (ampersand)
*       (asterisk)
(,)     (parentheses)
[,]     (square brackets)
{,}     (braces)
|       (vertical bar)
,       (comma)
.       (period)
'       (single quote)
"       (double quote)
`       (back tick)
:       (colon)
;       (semicolon)
...     (any non printing character, except " " (blank))
```

This reservation allows implementers to record meta-information in labels associated with datumIds, Class Attributes and Instance Attributes.

3.4.1.2 DCIInstLevel

The DCIInstLevel structure is used in an instantiated DCI class attributes structure to describe each instance level. This structure allows classes to have multiple, self described instance levels. For example, multilevel instances can be used to categorise metrics as "per-processor, per-disk I/O metrics" in a multiprocessor system with asymmetric I/O where disk drives are partitioned between processors.

The DCIInstLevel structure is shown in Figure 3-6.

```
typedef struct DCIInstLevel {
    UMADataType     type;
    UMAInstTagType  itype;
    UMAUint4        size;
} DCIInstLevel;
```

The usage of the structure elements is as follows:

type Type of the instance level value.

itype Instance type.

size Size of the instance level value in bytes.

Each instance level has an instance **type** (**itype**) and an instance value type (type) associated with it. Note that different classes can be instantiated by different types of instances. For example, consider a class having a 2 level instantiation: per-processor/per-device statistics. In this example, the class is instantiated by processor id, and by device id. There would be two DCIInstLevel structures associated with this class. The processor id instance level has a particular instance type (itype) associated with it (UMA_PROCESSOR); its instance value type (type) might be UMA_UINT4 (indicating that a four byte unsigned integer is used to record a specific processor id). The device id instance level has a different instance type (itype) associated with it (for example, UMA_DEVICE); its instance value type (type) might be

UMA_UINT8 (indicating that an eight byte unsigned integer is used to record a specific device number).

Instance types facilitate correct class-to-class navigation. In cases where several classes share a common instance representation (say, one instance level is the processor number, with **itype** UMA_PROCESSOR in each of the related classes), it is extremely useful to find the correlated metrics in each class. This can only be done if each instance representation in the system is identical for all relevant classes. For example, for two different classes which produce information per processor and have an **itype** of UMA_PROCESSOR, the numerical value for ''Processor #2'' must be identical.

UMAInstTagTypes are shown in Table 3-1, and enumerated in the **<uma.h>** file described in **Part 2** of this publication. UMADataTypes are shown in Table 3-2 on page 52, and enumerated explicitly in the **<uma.h>** file in **Part 2** of this publication.

Instance Type	Comment
UMA_SINGLEINST	a single instance of value '0' exists
UMA_WORKINFO	UMA_WORKINFO enumeration
UMA_WORKID	data associated with DCI_WORKINFO
UMA_MSG_QUEUE	
UMA_SEMAPHORE	semaphore handle
UMA_SHR_SEGMENT	shared segment handle
UMA_PROCESSOR	processor number
UMA_FSGROUP	
UMA_MOUNTPOINT	
UMA_INODE	inode number
UMA_DISKID	disk device number
UMA_BUCKET_NO	an index into a histogram
UMA_DISKPARTITION	
UMA_ACCESS_PORT	
UMA_DEVICE	generic device number
UMA_KERNEL_TABLES	
UMA_CHANNEL	channel number
UMA_IOP	I/O processor number
UMA_PATH	
UMA_SYSCALL	system call number
UMA_ENUMERATION	
UMA_STREAMS	
UMA_CONTROLLERID	controller number
UMA_SCHED_CLASS	scheduling class type
UMA_LOGICALVOL	logical volume identifier
UMA_REMOTE_FSTYPES	
UMA_IPADDR	IP address
UMA_FILESERVER_COMMAND	file server command
UMA_FILECLIENT_COMMAND	client command to a file server
UMA_SERVER_COMMAND	command to DCI server
UMA_CLIENT_COMMAND	command issued by DCI client
UMA_MEMOBJECT_ID	a memory object identifier

Table 3-1 The UMAInstTagType Enumeration

Note that there is a special kind of instance type called UMA_SINGLEINST. This is the instance level type used by a provider registering a class known to have only a single instance. Such classes typically include metrics of a global (or system-wide) nature: for example, refer to the global physical I/O counters class in the Data Pool Definitions (DPD) specification.

Also included is the specific size of the identifier for this particular instance level. The purpose of the DCIInstLevel size field is to indicate the specific size of each level, in a multiple level instance specification. In the above example, processor ids require 4 bytes and device numbers require 8. The DCIInstanceId structure instances.size field would be 12 (the sum of the size of each level in a multilevel instance). The DCIInstLevel size field would be 4 for the processor instance level, and 8 for the device instance level in this class's DCIClassAttr structure.

3.4.1.3 DCIDataAttr

The DCIDataAttr structure is used to describe an individual polled metric. It is used as an array field of the DCIClassAttr structure which is used as an argument to *dciRegister*( ) when the class is registered. The DCIDataAttr structure is defined as follows:

```
typedef struct DCIDataAttr {
    UMAUint4        size;
    DCIDatumId      datumId;
    UMADataType     type;
    UMAUnit         units;
    UMAUint4        flags;
    UMAUint4        offset;
    UMAVarLenDescr  label;
    UMAVarLenData   data;
} DCIDataAttr;
```

The usage of the structure elements is as follows:

size This is the total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

datumId The datum identifier for this metric which must be unique within the class that defines it. It must be in the range 1 to 2^32-2, since 0 is a delimiter value and 2^32-1 is the wildcard value. Each polled metric datumId must be distinct from all other datumIDs in the same class, including those for events.

type The datatype for this metric. This describes the representation of data as enumerated in Table 3-2 on page 52.

units For data that expresses a count or time, *units* expresses more information about the quality of the data; for example, the metric could be a count of pages, or a count of disks. For some types of metrics (such as text strings), *units* does not give additional information, and UMA_NOUNITS may be specified. In the case that *type* is UMA_DERIVED, the units field represents the actual derived data metric (see Table 3-7 on page 56). Table 3-3 on page 53, Table 3-4 on page 54, Table 3-5 on page 54 and Table 3-6 on page 55 represent various associated *units* values. This is the units used to describe this polled metric.

flags This describes other characteristics of the metric ID, in terms of what method operations the provider supports on the metric. See Section 3.4.7 on page 57 for more information on method operations and definitions of the DCI_OP_ flags. In particular, it should be set to the union of one or more of the following values:

DCI_QUERYABLE
The metric is queryable using the DCI_OP_GETDATA operation.

DCI_SETTABLE
The metric is settable using the DCI_OP_SETDATA operation.

DCI_RESERVABLE
The metric is reservable using the DCI_OP_RESERVEDATA operation, and releasable using the DCI_OP_RELEASEDATA operation.

offset When polled metric data is returned as a result of being specified using a wildcard as the datumId, all the data within a single class is returned in a single block. This field is the offset into that block where this polled metric starts.

label Descriptor for a variable length ASCII and internationalised text label describing this polled metric. If no label is desired, this must still contain a valid zero length string.

data Data section for all variable length information in this structure.

3.4.1.4 DCIEventAttr

The DCIEventAttr structure is used to describe an individual event metric. It is used as an array field of the DCIClassAttr structure which is used as an argument to *dciRegister*( ) when the class is registered. The DCIEventAttr structure is defined as follows:

```
typedef struct DCIEventAttr {
    UMAUint4            size;
    DCIDatumId          datumId;
    UMAVarLenDescr      label;
    UMAVarArrayDescr    eventDataAttr;
    UMAVarLenData       data;
} DCIEventAttr;
```

The usage of the structure elements is as follows:

size This is the total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

datumId The datum identifier for this metric which must be unique within the class that defines it. It must be in the range 1 to 2^32-2, since 0 is a delimiter value and 2^32-1 is the wildcard value. Each event datumId must be distinct from all other datumIDs in the same class, including those for polled metrics.

label Descriptor for a variable length ASCII and internationalised text label describing this event. If no label is desired, this must still contain a valid zero length string.

eventDataAttr
This is a descriptor for the event data that correspond to this event. The descriptor points to an array of DCIEventDataAttr structures, each of which specifies the format of one piece of data that is attached to an event metric by the provider when it is generated. See Section 3.4.3 on page 47 for information on how this relates to the event data itself.

data Data section for all variable length information in this structure.

3.4.2 DCIInstAttr

Instance attributes are considerably simpler than metric class attributes. The DCIInstAttr structure can be used to discover a particular instance's label and access control information. There is also a provision for local extensions to this structure. The DCIInstAttr structure definition is:

```
typedef struct DCIInstAttr {
    UMAUint4            size;
    UMAUint4            flags;
    UMAVarLenDescr      access;
    UMAElementDescr     extension;
    UMAVarLenDescr      label;
    UMAVarLenData       data;
} DCIInstAttr;
```

The usage of the structure elements is as follows:

size This is the total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

flags Special instance state flags.

access Descriptor for the DCIAccess access control structure for this instance.

extension Descriptor for variable length extension associated with this instance.

label Descriptor for variable length DCILabel structure associated with this instance.

data The actual variable length data (DCIAccess, DCILabel and extensions) associated with this instance.

The flags field contains a sequence of bit mapped flags. The values of these flags are given in the **<dci.h>** file. Their descriptions are below:

DCI_PERSISTENT_INSTANCE
The instance should not be removed from the namespace (that is, unregistered) when the process that registered this instance terminates (for example, issues the exit( ) system call). (See Section 2.6 on page 16, "Operating System Interaction" for a discussion of persistence.)

3.4.3 Events and Event Data Attributes

The only information required for some events is simply that they have occurred. However, there is a wide range of applications for events that have a need for associated data to describe additional characteristics of the event. The event related data structures in the DCI attempt to be flexible enough to match this requirements range without imposing undue burden upon the simple cases.

There are two specialised structures used only with events. The first structure, the DCIEventDataAttr, is used in the DCIEventAttr to describe event data to be delivered along the event when it occurs. The second, DCIEvent, is the structure used when an event occurs to describe the event itself, and pass along any associated event data.

Note that event metric attributes are separated from event data to improve event delivery performance by decreasing the size of the transmitted structures; the nonvariant attributes may be fetched only once and saved for future reference by the MAP.

3.4.3.1 DCIEventDataAttr

The DCIEventDataAttr structure is used in the DCIEventAttr structure. For example, if a provider needs to pass along three additional pieces of data whenever an event occurs, the *count* field of the eventDataAttr structure (in the DCIEventAttr structure, see Figure 3-7 on page 51) is set to three, and there will be three of the following structures in the variable length section of the DCIEventAttr structure.

```
typedef struct DCIEventDataAttr {
    UMAUint4        size
    UMADataType     type;
    UMAUnit         units;
    UMAUint4        offset;
    UMAVarLenDescr  label;
    UMAVarLenData   data;
} DCIEventDataAttr;
```

The usage of the structure elements is as follows:

size This is the total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

type This is the type of this event datum. This type implies either a fixed number or a variable number of bytes in the case of UMA_TEXTSTRING or UMA_OCTETSTRING.

units This is the units of this event datum.

offset This is the offset into the DCIEvent structure where this event is to be found.

label Descriptor for a variable length ASCII and internationalized text label describing this event datum. If no label is desired, this must still contain a valid zero length string.

data Data section for all variable length information in this structure.

There are several things to note about this structure. First is the fact that it does not describe a DCI metric. There is no mechanism to automatically retrieve the value of selected metrics whenever an event is reported. This structure describes raw data that will be delivered directly by a provider to the DCI Server when the provider makes a *dciPostEvent*( ) call.

Second, if the provider does not plan to transmit event data for an event, then the *count* field of the eventDataAttr structure (in the DCIEventAttr structure, see Figure 3-7 on page 51) is zero, and no DCIEventDataAttr structures will be present. A consumer can check the structure of the event data by using the *dciGetClassAttributes*( ) to retrieve this structure for any event metrics in the class.

3.4.3.2 DCIEvent

The DCIEvent structure describes the occurrence of an event itself. This is the structure that a consumer receives from *dciWaitEvent*( ). Part of the information in this structure is furnished when an event is reported using *dciPostEvent*( ), and part is furnished by the DCI Server itself, as follows.

Consumers have control over the format of output events; this control is expressed through the content of the datumId used at the time of the *dciOpen*( ). The datumId is logically divided into 3 pieces: 12 bits of user specified flags, 12 bits of user specified event id and 8 bits of datum id. [Recall that the datumId for polled metrics is partitioned to support derived metrics. Note that for both polled and event metrics, the low order 8 bits are reserved for the unique datum id.]

The flags determine the format and content of the returned data for this event:

DCI_EVENTHDR
An event header is to be included in the posted event structure

DCI_EVENTHDRCLASSID
A class id is to be included in the posted event structure

DCI_EVENTHDRINSTANCEID
An instance id is to be included in the posted event structure

DCI_EVENTHDRTIMESTAMP
A time stamp of the type UMATimeSpec is to be included in the posted event structure

DCI_EVENTHDRVENDORTIMESTAMP
A timestamp of an implementation defined format is included in the posted event structure

DCI_EVENTHDRCOMPTIMESTAMP
A compressed timestamp is to be included in the posted event structure

DCI_EVENTHDRSTREAMID
An event stream identifier is to be included in the posted event structure

DCI_EVENTHDRDATA
The event data is to be included in the event structure

Values for these flags are set by the provider at *dciRegister*( ) time to indicate the providers capability with respect to reporting data for the corresponding event. At *dciOpen*( ) time the DCI Server will use the provider's flags as a mask to determine the extent to which it can satisfy the consumer's request for event format and content. (For example, if the provider registered events indicating no DCI_EVENTHDR, then a consumer's request for event headers will be denied.)

These flags are also returned by the DCI Server as part of the header associated with all returned events (assuming the consumer has requested that an event header be included as part of the returned event's data).

The event header (DCI_EVENTHDR) is optional. If it is not present, this indicates that the structure of the event data is implementation defined. If it is present, then the event data is defined as follows:

- If the DCI_EVENTHDRCLASSID flag is enabled, then a complete class identifier is included in each event. The class identifier corresponds to the class in which this event is defined as an event metric.
- If the DCI_EVENTHDRINSTANCEID flag is enabled, then a complete instance identifier is included in each event. The instance identifier uniquely identifies the instance that generated this event.
- If the DCI_EVENTHDRTIMESTAMP flag is enabled, the DCI Server attaches a timestamp to every event when it occurs. This timestamp can be used by a consumer for event ordering. The timestamp field represents the value of the system time of day as close as possible to the actual posting of the event. Note that events will not typically have unique timestamps, so additional information may be required in an implementation if event uniqueness is required.
- If the DCI_EVENTHDRVENDORTIMESTAMP is enabled, the DCI Server attaches a timestamp of an implementation defined format to every event when it occurs.
- If the DCI_EVENTHDRCOMPTIMESTAMP flag is enabled, then an implementation defined compressed timestamp is included in the event data. Typically, this would consist of the

lower order 32 bits of the full 64 bit timestamp.

- If the DCI_EVENTHDRSTREAMID flag is enabled, then an implementation defined 32-bit identifier is included in the event data that can be used to discriminate between different event streams (for example, say from different processors on a symmetric multiprocessor system).
- If the DCI_EVENTHDRDATA flag is enabled, then the event data, as described in the DCIEventDataAttr structure, is included in the event.

For events containing an eventHeader, the event structure is defined as follows:

```
typedef struct DCIEvent {
    UMAUint4            eventHeader;
    UMAVarLenData       data;
} DCIEvent;
```

Note that the structure of events lacking an event header is implementation defined.

The usage of the structure elements is as follows:

eventHeader The event header is a variable length structure. The first word consists of 12 bits of flags, 12 bits of event id and 8 bits of size. If the size byte contains 0xff, then the event's size exceeds 256 bytes and a subsequent 32 bit word contains the actual size of the event.

data Data section for all variable length information in this structure. This includes any data mandated by the presence of the event header flags, followed by the variable length data prescribed for this event metric by its registered DCIEventDataAttr structure.

Figure 3-7 on page 51 shows how the information in the three structures, DCIEventAttr, DCIEventDataAttr, and DCIEvent are related.

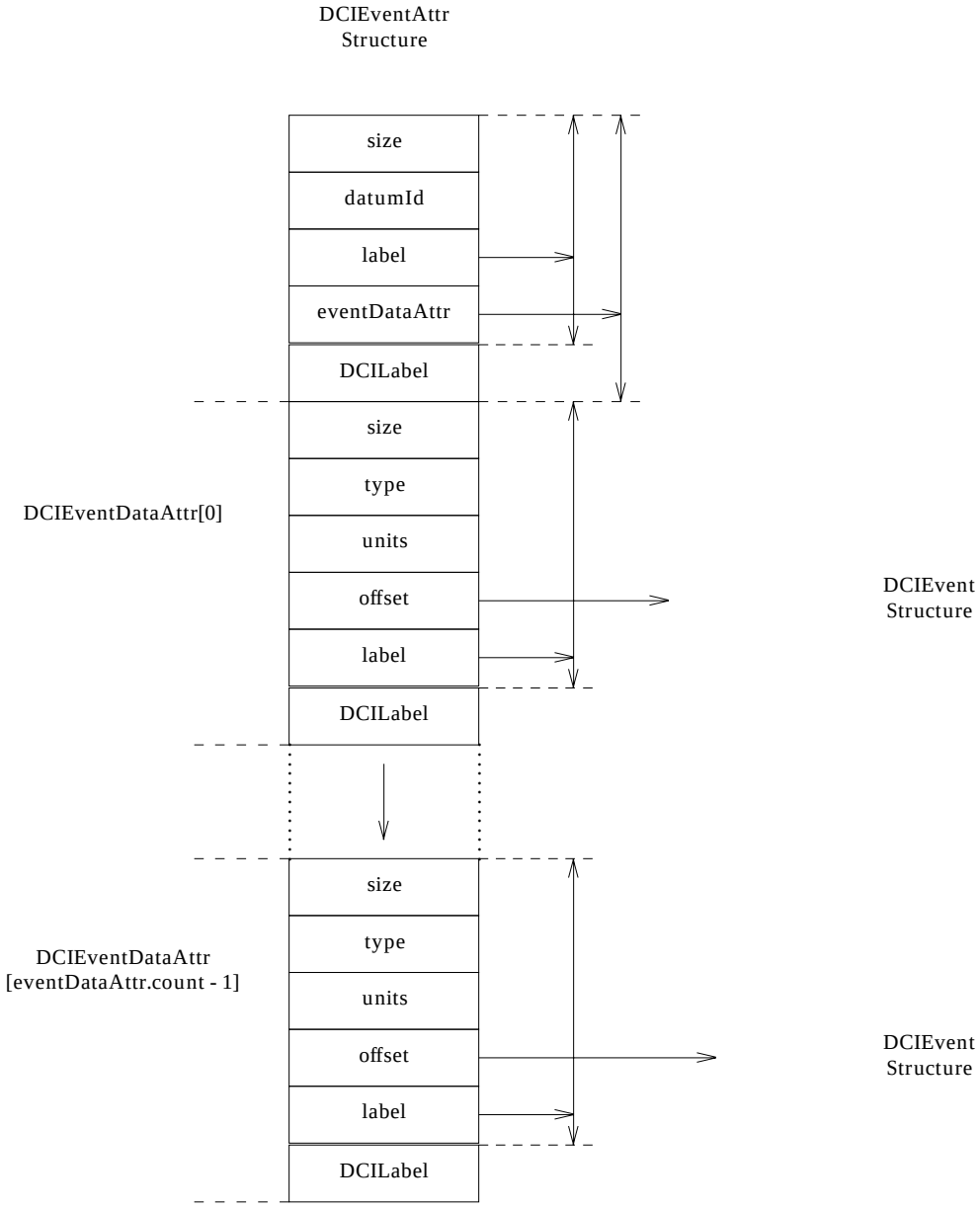


Figure 3-7 DCIEventAttr and DCIEventDataAttr Structures
(with reference to DCIEvent structure)

When a provider reports an event using the *dciPostEvent*( ) function, the DCI Server combines the information furnished by the provider with the information it provides itself to form a DCIEvent structure. The DCI Server then passes this structure to any consumers who are waiting for this event.

3.4.4 Data Types

The dataType field in the attributes structures describes the metric type. By examining the value of dataType one can determine the type and size of data object for the associated metric. These are necessary for correct interpretation of the data described by the attributes structure.

The following table lists the symbolic name for the UMA data type value, the related UMA data type, the size of the data type in bytes, and a brief description. It is important to note that the UMA data type values are simply a flag indicating the type and the UMA data type column are the types themselves. So a UMA_INT4 value in dataType states that a UMAInt4, or a four byte integer, is being used for a particular metric.

UMA Data Type Value	UMA Data Type	Size in Bytes	Description
UMA_INT4	UMAInt4	4	four byte integer
UMA_INT8	UMAInt8	8	eight byte integer
UMA_UINT4	UMAUint4	4	four byte unsigned integer
UMA_UINT8	UMAUint8	8	eight byte unsigned integer
UMA_BOOLEAN	UMABoolean	4	boolean value, with FALSE = 0, TRUE != 0
UMA_OCTETSTRING	UMAOctetString	variable	octet string
UMA_TEXTSTRING	UMATextString	variable	variable length text data
UMA_TIMEVAL	UMATimeVal	8	UNIX like time structure
UMA_TIMESPEC	UMATimeSpec	8	time interval in nanoseconds
UMA_DERIVED	—	0	data that must be derived from other metrics.
UMA_CLASSDATA	UMAClassData	variable	all polled metrics within the same class.

Table 3-2 UMADataType Values

The first five data types listed in Table 3-2 consist of signed and unsigned integers of different sizes. The next four entries describe different structures. In all cases, the size of a data object is padded to a four-byte boundary. (The reason for this restriction is so that structures built from a combination of these data types can always be four-byte aligned. When performing unaligned accesses, such as reading a four-byte integer at an address having the least significant bit set, some system architectures suffer serious performance degradation, or are simply unable to perform the access.)

Three variable length data types are supported: UMAOctetString, UMATextString and UMAClassData. The data structures for the first two types are identical: a variable length byte array called ‘‘string’’ and a ‘‘size’’ field includes the size of the entire structure along with the variable length data and enough padding ensure that the ‘‘size’’ field is always a multiple of 4 bytes in size. While the UMATextString data type is composed of a null-terminated array of bytes, the UMAOctetString may include embedded nulls or multibyte characters. This interpretation of the UMAOctetString data is beyond the scope of this specification.

```
typedef struct UMAString {
    UMAUint4        size;       /* size of entire structure */
    char            string[1]   /* variable length text string */
} UMATextString, UMAOctetString
```

The data structure for the UMAClassData depends on the context in which it appears. Its purpose is to compactly identify the contents of the *Final Data* event associated with a class. Providers for classes that support final data events must return final values for all polled metrics in that class prior to instance termination. The definition of UMAClassData simplifies the registration of such a *Final Data* event by requiring that the DCIEventDataAttr for that event declare the type of data as UMAClassData. [This avoids requiring repetition of all attributes for

all polled metrics explicitly in the DCIEventDataAttr structure.] The *Final Data* event for a particular class would contain a single Event Data Attribute having type UMAClassData, units UMA_NOUNITS, and offset 0.

The UMATimeVal and UMATimeSpec structures specify a time interval in terms of a number of seconds and a number of fractional seconds: in the case of UMATimeVal the fractional seconds are measured in microseconds and in the case of UMATimeSpec the fractional seconds are measured in nanoseconds. These time interval structures are used for timeout parameters as well as timestamps; in the latter case, the number of seconds is given relative to the date Jan 1, 1970 at 12 am (also known as the Epoch). These time intervals are malformed if the number of seconds is negative or if the fractional number of seconds is negative or if the total fractional component is greater than or equal to 1 second.

The UMA_DERIVED data type is used to encode relationships between datums within a single class. When used as the type of a particular DCIDatumId, the corresponding **UMAUnit** field should be examined to determine the specific interrelationship that is encoded. See *Measurement Units* for further discussion of the supported encodings.

Note that a data type of UMA_DERIVED indicates that the associated metric is not available directly, but must be derived from other metrics using the encoded interrelationship. Such a derived metric cannot be explicitly specified for data retrieval, but must be part of a whole class of retrieved data (i.e., the class has been fetched with the wildcarded datumId). The values of derived metrics must be obtained by the application; there is no DCI API support for doing so directly.

3.4.5 Measurement Units

The attributes structure *unit* field of typedef **UMAUnit** is used to indicate what the metric is measuring (or, in the case of derived metrics, how the metric is related to other metrics). For non-derived metrics, the purpose of the *unit* field is to provide additional descriptive information about the properties of the metric to the measurement application (the DCI consumer). For example, for metrics that report memory related size information, the *unit* field indicates whether the value returned is expressed in bytes, kilobytes, megabytes, etc. For metrics that indicate time values, the *unit* field indicates whether the value returned is in seconds, milliseconds, microseconds, nanoseconds, etc.

The DCI provides for five distinct unit types: size, time, count, info and derived. (There is an additional unit type: UMA_NOUNITS, used for metrics for which the *unit*s designation is not meaningful.) Within each unit type there are many values; these are enumerated below in a series of tables.

Size	
UMA_BYTES	bytes
UMA_KBYTES	kilobytes
UMA_MBYTES	megabytes
UMA_GBYTES	gigabytes
UMA_TBYTES	terabytes

Table 3-3 Size Units

Time	
UMA_SECS	seconds
UMA_MILLISECS	milliseconds
UMA_MICROSECS	microseconds
UMA_NANOSECS	nanoseconds
UMA_PICOSECS	picoseconds
UMA_TICKS	machine clock ticks

Table 3-4 Time Units

There are a large number of object types that can be counted and new operating systems can always invent new types of objects. The purpose here is to identify some of the commonly measured objects and establish some *unit* field values for these. The types of objects being counted can be divided naturally into two types: software objects or ‘‘system abstractions’’ and hardware objects.

Counts of System Abstractions	
UMA_COUNT	an unspecified count
UMA_EVENT	an unspecified event
UMA_PAGES	memory pages
UMA_BLOCKS	disk blocks
UMA_CHARACTERS	characters
UMA_QLENGTH	queue length
UMA_PROCESSES	processes
UMA_TASKS	tasks
UMA_THREADS	threads
UMA_JOBS	jobs
UMA_USERS	users
UMA_TRANSACTIONS	transactions
UMA_MESSAGES	messages
UMA_SESSIONS	sessions
UMA_STREAMSMODULES	streams modules
UMA_STREAMSHEADS	streams heads
UMA_STREAMSMSGS	message blocks
UMA_PACKETS	packets
UMA_INODES	inodes
UMA_FILES	files
UMA_FILESYSTEMS	file systems

Table 3-5 System Abstraction Count Units

Counts of Hardware Activity or Objects	
UMA_READS	reads
UMA_WRITES	writes
UMA_SEEKS	seeks
UMA_IOCTLS	ioctls
UMA_CONNECTIONS	connections
UMA_RETRIES	retries
UMA_MOUNTS	mounts
UMA_REWINDS	rewinds
UMA_POSITIONINGS	positionings
UMA_MARKS	tape marks
UMA_PORTS	ports
UMA_PROCESSORS	processors
UMA_DISKS	disks
UMA_NETS	networks
UMA_SLINES	serial lines
UMA_BUSSES	busses
UMA_CHANNELS	I/O channels

Table 3-6 Hardware Activity Count Units

Two of the above counts, UMA_COUNT and UMA_EVENT, can be used for undefined counts and events. There will, of course, be objects that are not listed here or vendor specific objects. A value of UMA_NOUNITS indicates that units are not applicable to this metric. To handle future and vendor expansion this specification reserves the numerical range 0-2^16 for predefined DCI unit values. The rest of *unit*'s numerical range is available for extension.

Refer to the **<uma.h>** file at the end of this specification for a mapping of DCI units to a specific enumerated type.

For derived metrics (i.e., those whose dataType attributes field is recorded as UMA_DERIVED), the units indicate precisely how this derived metric is related to other metrics in the class.

Inter-metric relationships are encoded explicitly in the DCIDatumId of each derived metric. For this purpose, the DCIDatumId is partitioned in the following manner:

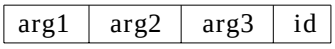


Figure 3-8 DCIDatumId for Derived Metric Support

Where arg1, arg2 and arg3 are the unique (within a class) last 8 bits of the DCIDatumIds of the other metrics in the class to which the derived metric is related.

The following types of relationships can be expressed:

Manifest Constant	Argument Definition	Formula
UMA_DERIVED_SUM2	arg1 = a value arg2 = a value	arg1 + arg2
UMA_DERIVED_SUM3	arg1 = a value arg2 = a value arg3 = a value	arg1 + arg2 + arg3
UMA_DERIVED_DIFFERENCE	arg1 = a value arg2 = a value	arg1 - arg2
UMA_DERIVED_AVERAGE	arg1 = a value arg2 = a value	arg1 / arg2
UMA_DERIVED_PERCENT	arg1 = a value arg2 = a value	arg1 / (arg1 + arg2)
UMA_DERIVED_PRODUCT	arg1 = a value arg2 = a value	arg1 * arg2
UMA_DERIVED_VARIANCE	arg1 = a count arg2 = a sum arg3 = a sum of squares	(arg3/arg1)-(arg2)^2/arg1

Table 3-7 Derived Data Units

(Note that a definition of standard deviation is not given since standard deviation is derived directly from variance, and does not depend on relationships between other metrics.)

Finally, for certain metrics, the units designation is not meaningful. In these cases, the UMA_NOUNITS value is selected.

For situations in which "units" do not apply	
UMA_NOUNITS	indicates units do not apply to this metric

Table 3-8 Metrics with no Units

3.4.6 Invalid Data

There are cases where a whole class of data cannot be retrieved for all instances, such as the "byteswritten" count for ReadOnly devices, or the size of unmounted disk partitions. In these cases, rather than splinter the class to contain only the valid metrics, an additional data structure can be returned with the fetched data and the application can check whether the data it seeks is valid or not. As a performance optimization, the application need only check each data metric for validity if a hint bit is set in the class attributes flags (DCI_POSSIBLEINVALIDDATA) and the informational return status DCI_INVALIDDATAPRESENT is received. In the case of *dciWaitEvent*( ), there is no way to return DCI_INVALIDDATAPRESENT and the measurement application (consumer) is obligated to examine each piece of data for whole classes that have the DCI_POSSIBLEINVALIDDATA attribute flag set.

A special datumId (DCI_INVALIDDATUMID) is reserved for use in classes which need to express partial class data for some instances. The *dciRegister*( ) will fail if a class is specified with DCI_POSSIBLEINVALIDDATA and a DCI_INVALIDDATUMID was not. This associated data for this datumId is a variable length DCIInvalidData structure (since this structure is variable length, the data offset for the datumId represents an indirect offset to the actual structure). This structure is an array of DCIDatumIds which are not valid in the presented class data for this instance. If an invalid metric is explicitly referenced (using *dciGetData*( ) or *dciSetData*( ) with a non-wildcarded DCIDatumId) then the error DCI_INVALIDDATA will be generated. If this class can produce invalid data, but all metrics are valid for this instance, the offset for the DCIInvalidData structure is 0 and should not be referenced.

If a particular metric is marked as invalid, then its value should not be referenced by the application. Such invalid metrics may not be settable with *dciSetData*( ), depending upon the provider.

For example, if a class of data is returned with the informational status DCI_INVALIDDATAPRESENT, then the invalid list must be checked before referencing a metric in the returned list. One must obtain the UMAUint4 offset for the DCI_INVALIDDATUMID; if the offset is zero, then no validity data needs to be consulted (although in this case, the implementation should have not generated the DCI_INVALIDDATAPRESENT status). If the offset is non-zero, then it is added to the base address of the returned data and cast to a DCIInvalidData structure pointer. Use the macro *dciinvaliddatacount* to determine how many invalid metrics exist and the macro *dciinvaliddatumid* to check each DCIDatumId in the list.

3.4.7 DCI Server/Provider Communication

The DCI Server needs to communicate with a provider for several purposes. When a *dciGetData*( ) or *dciSetData*( ) request is made, the DCI Server must forward the request to the appropriate provider and then return the appropriate data to the consumer that requested it. When a consumer calls *dciConfigure*( ) to configure a metric, the DCI Server must ask the provider to perform the configuration operation. Finally, if a provider has specified DCI_PROVIDER_INSTANCE in the flags field of the DCIClassAttr, then that provider is saying that it will never make any calls to *dciAddInstance*( ) or *dciRemoveInstance*( ). So whenever the DCI Server needs to find out what instances are available, it must explicitly ask the provider to find out what instances currently exist. Typically, a DCI Server will need this information in order to respond to a *dciListInstanceId*( ) or *dciGetInstAttributes*( ) request from a consumer. One reason a provider may decide to do this is that the instance space of a class is changing very rapidly and the overhead of making frequent calls to *dciAddInstance*( ) and *dciRemoveInstance*( ) is deemed unacceptable to system performance. Methods are the mechanism that allow a provider to instruct the DCI Server exactly how to perform this communication.

The DCI specifies four method types, as outlined in the following table, which are explained in detail in further sections.

Method	Explanation
DCI_WAIT	Provider blocks waiting for requests
DCI_STORE	DCI Server stores posted data
DCI_ADDRESS	DCI Server retrieves data from provider's address space
DCI_CALLBACK	DCI Server requests data by calling a routine in provider's address space

Table 3-9 Method Types

All DCI implementations must provide at least the DCI_WAIT method for portability, but may choose to either implement or not implement the other three methods since these may depend on features that may not be available on some operating systems. Also, some methods available to user space metric providers may not be available to operating system metric providers on the same machine. For example, it doesn't make sense to have a device driver block waiting for a metrics request. An implementation may also freely specify additional method types as well. The DCI Server returns an error if a provider attempts to specify an inappropriate or unavailable method.

Methods can be specified on a class basis using *dciRegister*( ) when the class is registered, or for a specific instance using *dciAddInstance*( ) when the instance is added. For a DCI_PROVIDER_INSTANCE class, a provider may only (and must) specify a class method, since instance data is managed by the provider. In all other cases, either an instance method, a

class method, or both may be specified.

A method, either a class method or an instance method, may support several of the different types of requests that a DCI Server may make to a provider. The type of request is embodied in one of seven bitmap operation codes.

Operation Type	Corresponding Consumer API Call
DCI_OP_GETDATA	dciGetData( )
DCI_OP_CONFIGURE	dciConfigure( )
DCI_OP_LISTINSTANCES	dciListInstanceId( ) for DCI_PROVIDER_INSTANCE class
DCI_OP_GETINSTATTR	dciGetInstAttributes( ) for DCI_PROVIDER_INSTANCE class
DCI_OP_SETDATA	dciSetData( )
DCI_OP_RESERVEDATA	dciSetData( )
DCI_OP_RELEASEDATA	dciSetData( )

Table 3-10 Types of Operations

When the DCI Server executes a method, it indicates which operation it is requesting. If the provider registered both a class method and an instance method, the DCI Server tries the instance method first. If that method fails it retries the operation using the class method. If any of the methods fail, the DCI Server returns an error to the requesting consumer.

The provider responds to the request with a *dciPostData*( ) call, which also indicates to which type of operation it is responding. All <specified> methods, except DCI_ADDRESS, use *dciPostData*( ) to deliver the metrics and other data to the DCI Server.

The DCIMethod structure is used to identify a method. It can describe a class method if set in the DCIClassAttr structure, or can describe an instance method if used as an argument to *dciAddInstance*( ). The DCI Server can then use the information in this structure to determine what method to use to satisfy requests from a consumer.

```
typedef struct DCIMethod {
    UMAUint4            size;
    DCIMethodType       type;
    UMAElementDescr     method;
    UMAVarLenData       data;
} DCIMethod;
```

The usage of the structure elements is as follows:

size This is the total number of contiguous bytes of storage in the structure and all its associated variable length data, which must be a multiple of 4.

type The method type which indicates how data will be retrieved, and is one of DCI_WAIT, DCI_STORE, DCI_ADDRESS, or DCI_CALLBACK.

method Descriptor for variable length method data.

data Data for the variable length attributes and method data.

Not every method type can support every operation. Although described in detail below, Table 3-11 on page 59 summarises which methods may support which operations. Attempting to specify an unsupported operation in a DCIMethod argument results in an error code of DCI_METHODOPUNAVAILABLE.

Method Operations	Method Types			
	DCI_WAIT	**DCI_STORE**	**DCI_ADDRESS**	**DCI_CALLBACK**
DCI_OP_GETDATA	yes	yes	yes	yes
DCI_OP_CONFIGURE	yes	no	no	yes
DCI_OP_LISTINSTANCES	yes	no	no	yes
DCI_OP_GETINSTATTR	yes	no	no	yes
DCI_OP_SETDATA	yes	yes	yes	yes
DCI_OP_RESERVEDATA	yes	yes	yes	yes
DCI_OP_RELEASEDATA	yes	yes	yes	yes

Table 3-11 Valid Operations for Each Method Type

The following sections describe the use of provider operations, *dciPostData*( ) and methods in detail.

3.4.7.1 Provider Operations for Polled Metrics

In most cases all data transferred from the DCI Server to the provider, or vice versa, are encoded in a DCIReturn structure. The exception is DCI_ADDRESS, in which case the DCI Server retrieves the requested data itself. To clarify the role of the metric identifiers and corresponding data for each operation, the DCIReturn structure in both directions are described in detail for each operation. First some behaviour identical to all methods is described.

Class identifiers can never be wildcarded at the provider level. So, wildcards are not permitted in the class identifiers in any of the DCIReturn structures passed between DCI Server and provider. Datum identifiers can always be wildcarded, although in some methods the datum identifiers are ignored. For DCI_PROVIDER_INSTANCE classes the instance identifier can be wildcarded in any of the operations. For all other classes the instance identifier cannot be wildcarded. Note that the DCI_OP_LISTINSTANCES and DCI_OP_GETINSTATTR operations are only used for DCI_PROVIDER_INSTANCE classes. A DCI_PROVIDER_INSTANCE class provider expands the wildcarded instance identifiers as necessary.

Since the data returned by the provider is located in a separate data buffer, all offsets in the return DCIReturn structure are relative to the start of this data buffer.

DCI_OP_GETDATA

The DCIReturn structure as it is received by the provider contains fully specified metric identifiers. Summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately with offsets to the actual data, which conforms to the previously registered attributes structure. In the case of a datum identifier wildcard, the corresponding data section contains all data of the specified class instance, each datum with an offset as specified in the previously registered attributes structure. Summary, status and return values are significant.

DCI_OP_CONFIGURE

The DCIReturn structure as it is received by the provider contains fully specified metric identifiers with offsets to the actual data. The instance identifier may be of zero length, which means "all instances, including future ones". Each data section corresponding to a metric identifier contains DCIConfig data structure(s). Summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately with offsets to the actual data. Each data section corresponding to a metric

identifier contains DCIConfig data structure(s). Summary, status and return values are significant.

The *dciConfigure*( ) manual page further defines the DCIConfig structure for both the request and reply directions of any configure operation.

DCI_OP_LISTINSTANCES
The DCIReturn structure as it is received by the provider contains metric identifiers of which the datum identifier is ignored. No data is provided and summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately. The datum identifiers are ignored. No data is provided. Summary, status and return values are significant.

DCI_OP_GETINSTATTR
The DCIReturn structure as it is received by the provider contains metric identifiers of which the datum identifier is ignored. No data is provided and summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately, and contains offsets to the actual data. The datum identifiers are ignored. Each data section corresponding to a metric identifier contains DCIInstAttr structure(s). Summary, status and return values are significant.

DCI_OP_SETDATA and DCI_OP_RESERVEDATA
The DCIReturn structure as it is received by the provider contains fully specified metric identifiers with offsets to the actual data, which conforms to the previously registered attributes structure. In the case of a datum identifier wildcard, the corresponding data section contains all data of the specified class instance(s), each datum with an offset as specified in the previously registered attributes structure. Summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately. No data is returned. Summary, status and return values are significant.

DCI_OP_RELEASEDATA
The DCIReturn structure as it is received by the provider contains fully specified metric identifiers. Summary and status values are unused.

The DCIReturn structure returned by the provider repeats or expands all metric identifiers appropriately. No data is returned. Summary, status and return values are significant.

3.4.7.2 Provider Methods for Polled Metrics

DCI_WAIT
The DCI_WAIT method indicates that the provider is planning to register its metrics and then use the *dciWaitRequest*( ) routine to wait for consumer requests for its registered metrics. For a provider to register it, the type field of the DCIMethod structure is set to DCI_WAIT. There is no method data required when registering a DCI_WAIT method. All operations are supported by this method.

For a provider to use the DCI_WAIT method, it first makes a call to *dciWaitRequest*( ) that will block until the DCI Server requests an operation on one or more of the indicated metrics. When the call returns the operation is filled in, and the DCIReturn structure contains the requested metric identifiers. If data is associated with the request in the case of a DCI_OP_SETDATA, DCI_OP_RESERVEDATA or DCI_OP_CONFIGURE, it is contained in the DCIReturn structure. The provider must answer to the request by submitting results

with *dciPostData*( ).

DCI_STORE

The DCI_STORE method indicates that the provider wants to spontaneously generate polled metric data and send it to the DCI Server using *dciPostData*( ) without waiting for the DCI Server to make an explicit request. The DCI Server should independently handle requests from consumers, without interaction with the provider. The DCI Server is thus responsible for allocating storage to retain the values of these metrics and for returning appropriate errors to consumers or providers if it is unable to allocate this memory (or disk space). This method may not be used to support the DCI_OP_LISTINSTANCES, DCI_OP_GETINSTATTR, or DCI_OP_CONFIGURE operations.

When this method is used, each *dciAddInstance*( ) call, for which this method is in effect, must specify the initial values of all metrics in the class instance, by filling out the DCIReturn buffer.

If the DCI_OP_GETDATA operation on a metric is supported by the DCI_STORE method, a consumer may retrieve the current value of the metric using *dciGetData*( ), and the provider may change it using *dciPostData*( ). If the DCI_OP_SETDATA operation on a metric is supported by the DCI_STORE method, a consumer with proper access rights may also set the value of the polled metric using *dciSetData*( ).

DCI_ADDRESS

The DCI_ADDRESS method may be used when the provider is able to specify the location of the values of metrics in terms of a single address per metric. It may be supported by implementations on operating systems that allow the DCI Server to reach into the address space of a provider and read memory. It is an error to specify this method type to support the DCI_OP_LISTINSTANCES, DCI_OP_GETINSTATTR or DCI_OP_CONFIGURE method type.

In order to use this method type, all polled metric values for the class must exist in a single block of address space that can be copied directly into the data area of a DCIReturn structure. Each datum in the class must exist at the offset into the block that is specified in the offset field of the corresponding DCIDataAttr structure.

When the method is registered for an instance method using *dciAddInstance*( ) or a class method using *dciRegister*( ), the data area of the DCIMethod struct contains the following struct.

```
struct DCIAddressMethodData {
    void        *address;
    UMAUint4    size;
    void        *sync;
};
```

The usage of the structure elements is as follows:

address Address of the block of memory in the provider's address space where values for the polled metrics in this class start.

size The size of the memory block in bytes.

sync Pointer to an implementation dependent synchronisation structure used to synchronise access to the entire block of memory pointed to by address. If this is zero, no synchronisation is performed.

When a consumer requests an operation supported by the DCI_ADDRESS method, the DCI Server goes out and directly reads or sets the current value of this metric using the sync

object to synchronise access to that location.

If the address space of a provider that has registered a DCI_ADDRESS instance method becomes inaccessible to the DCI Server, for example, if the provider terminates, all instances added by that provider are implicitly removed. If a DCI_CLASS method was registered, then the class is implicitly removed.

DCI_CALLBACK

The DCI_CALLBACK method allows a provider to specify a function in its address space that the DCI Server will call to perform an operation. It may be supported by implementations on operating systems that allow the DCI Server to reach into the address space of a provider and execute code. All operations are supported by this method.

When a provider specifies this method, it supplies the DCI Server with a function that it may call to perform an operation. When the DCI Server wants to execute an operation, it calls this function with the following prototype:

```
DCIStatus callback(
    UMAUint4    operation,      /* in */
    DCIReturn   *request        /* in */
);
```

The usage of the structure elements is as follows:

operation

The operation being requested. Must be one of the values possible in the operation field of the DCIMethod struct.

request

Pointer to the DCIReturn structure containing all metric identifier information, and data in the case of DCI_OP_SETDATA, DCI_OP_RESERVEDATA or DCI_OP_CONFIGURE. The metric identifiers may contain wildcards for the datumId, for the instanceId in the case of a class method, but not for the classId.

When a consumer requests an operation implemented by this method, the DCI Server makes the appropriate callback. The provider must use *dciPostData*( ) to post the data requested by the operation, as it cannot be returned using the callback.

If the address space of the provider that has registered a DCI_CALLBACK method becomes inaccessible to the DCI Server, for example if the provider terminates, all instances added by that provider are implicitly removed. If a class method was registered, the class is implicitly removed.

Note that the callback routine is not a part of the DCI specification. It is provided by the provider. The routine must follow certain conventions. These are outlined below:

Description of **callback** behaviour

The **callback** routine is a prototype routine for the callback routine used in the DCI_CALLBACK method. This routine is defined by the provider using a local name and registered together with the DCI_CALLBACK method with the DCI Server, using *dciRegister*( ) or *dciAddInstance*( ). The DCI Server will call this routine when a consumer requests the provider's metrics using *dciGetData*( ), alters metrics using *dciSetData*( ), lists instances with *dciListInstanceId*( ), requests instance attributes with *dciGetInstAttributes*( ), or configures metrics using *dciConfigure*( ).

When the DCI Server executes this callback function, it passes in the operation and the requested metrics. If data is passed it is encoded in the DCIReturn structure using offsets from the beginning of the DCIReturn buffer.

If the operation is DCI_OP_GETDATA or DCI_OP_RELEASEDATA, only metric identifiers are supplied in the DCIReturn structure. Only datum identifier wildcards are allowed, unless the class is a DCI_PROVIDER_INSTANCE class in which case instance identifier wildcards are allowed.

If the operation is DCI_OP_LISTINSTANCES, only metric identifiers are supplied in the DCIReturn structure, of which the datum identifier should be ignored. The instance identifiers can be wildcarded.

If the operation is DCI_OP_GETINSTATTR, only metric identifiers are supplied in the DCIReturn structure, of which the datum identifier should be ignored. The instance identifiers can be wildcarded.

If the operation is DCI_OP_CONFIGURE, data is supplied in the DCIReturn structure in addition to the metric identifiers. Each data section contains one or more DCIConfig structures, depending on the number of instance identifiers encoded in the corresponding metric identifier. The instance identifier can be zero length which indicated ''all current and future instances''. Only datum identifier wildcards are allowed, unless the class is a DCI_PROVIDER_INSTANCES class in which case instance identifier wildcards are allowed.

If the operation is DCI_OP_SETDATA or DCI_OP_RESERVEDATA, data is supplied in the DCIReturn structure in addition to the metric identifiers. This data conforms to the previously registered attributes structure. Only datum identifier wildcards are allowed, unless the class is a DCI_PROVIDER_INSTANCES class in which case instance identifier wildcards are allowed.

Return Values from **callback**

The **callback** prototype routine returns DCI_SUCCESS to the DCI Server if it successfully serviced the request. Otherwise, this routine returns one of the DCI error values to indicate the failure type.

[DCI_NOTPRESENT]
The DCI service is not available.

[DCI_NOIMPLEMENTATION]
In a DCI subset implementation, the specified routine has not been implemented.

[DCI_NOTINITIALIZED]
The DCI subsystem is not currently initialised.

[DCI_SYSERROR]
An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.

[DCI_ALLOCATIONFAILURE]
The provider could not allocate memory for the return buffer which would be submitted with *dciPostData*( ).

[DCI_INVALIDARG]
One of the input arguments is invalid: a negative value was used for numIds, bufferSize is smaller than the size of a DCIReturn structure, MetricIdList was malformed, or the MethodList was malformed.

[DCI_INTERRUPTED]
The **callback** prototype call was interrupted by a signal and did not complete.

3.5 DCI Routine Return Status and Structures

The DCI routines have a consistent interface. An application, either a metrics provider or consumer, calls a routine with a list of metric identifiers as an argument. The DCI routine performs the request and then writes status and data into a user provided buffer or, if requested, into a buffer allocated on behalf of the user by the DCI. If the DCI allocated the status buffer, then the user is responsible for subsequently freeing the allocated memory. The return structure used for this buffer, pictured in the accompanying diagram, is the same for the different DCI routines. The diagram shows a successful return structure for a request with a list, after wildcard expansion, of ''n'' metrics.

A DCI routine always returns the DCIStatus type. These routines use the DCIStatus type definition to return success when the request succeeds for every metric in the expanded input metric list. In the success case, the routine DCIStatus return value is DCI_SUCCESS. A DCIStatus error or warning value is returned if the request cannot be satisfied for any metric in the list. The error status value is a summary error and may not be the same for every metric in the expanded input list. The application must traverse the DCIReturn structure to discover the status value for each metric in the expanded input list.

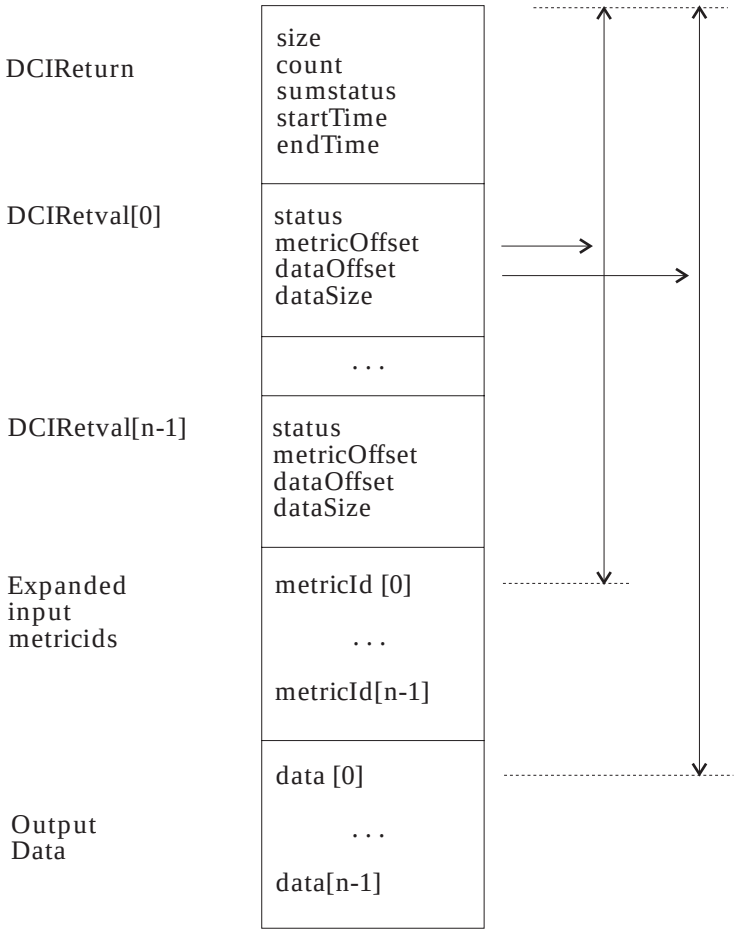


Figure 3-9 DCIReturn Structure Example

The return structure begins with a four byte integer that contains the total number of bytes written. The purpose of this size is to allow range checking and for quick traversal of multiple DCIReturn structures. The total size is followed by a count of the total number of expanded input list identifiers. The next field consists of a summary status word. Two timestamps follow; the first indicates the time of the start of the DCI operation; the second indicates the time of the completion. Both timestamps are (optionally) filled in by the DCI Server/server library, and only have significance for the dciGetData api call. The two timestamps may be used by a consumer to gauge the currency of the collected data; if the two timestamps are not "close enough", the DCI operation may have taken too long to complete. (If the timestamps are not "close enough", the consumer may choose to discard the data and retry the request.) Each timestamp is optional. A value of zero (0) for either timestamp indicates that the implementation does not support timestamping at the server. [Note that if the provider also chooses to provide a timestamp, it may do so by including a specific datumId in the class for that purpose.]

This header, the total size, count, status and timestamps, is followed by three arrays. The first is an array of count DCIRetval structures, one for each expanded input metric identifier. The DCIRetval structure gives the return status, an offset to a copy of the metric identifier, an offset to the output data for this request, and the size of the data returned by this request. All offsets are from the beginning of the DCIReturn structure. Note that the DCIRetval structure is a fixed size, allowing for quick traversal of the DCIRetval array.

Following the array of DCIRetval structures is a copy of the expanded input metric identifier list and the returned data buffer. The expanded metric identifier list may be either a list of DCIClassId's or DCIMetricID's. Some DCI routines, such as *dciListClassId*( ), take metric class identifier lists as input. Others, such as *dciListInstanceId*( ), take full metric identifiers, class plus instance, as input. Applications acquire offsets into these buffers by traversing the DCIRetval array.

The DCIReturn, DCIRetval, and DCIStatus structures and type definitions are defined as follows:

```
typedef struct DCIRetval {
    DCIStatus   status;         /* status for input argument      */
    UMAUint4    metricOffset;   /* offset to input id value       */
    UMAUint4    dataOffset;     /* offset to data value           */
    UMAUint4    dataSize;       /* size of data returned in bytes */
} DCIRetval;

typedef struct DCIReturn {
    UMAUint4    size;           /* total bytes in DCIReturn       */
    UMAUint4    count;          /* number of returned elements    */
    DCIStatus   sumstatus;      /* summary status                 */
    UMATimeSpec startTime;      /* Start time of operation        */
    UMATimeSpec endTime;        /* End time of operation          */
    DCIRetval   retval[1];      /* status,input id, and output    */
} DCIReturn;

typedef UMAUint4 DCIStatus;
typedef struct DCIRetval DCIRetval;
typedef struct DCIReturn DCIReturn;
```

Some routines may not return data. In this case there is no returned data area in the DCIReturn structure and the dataOffset and dataSize fields of the DCIRetval structure are unused and their values are set to zero. Clearing these fields prevents applications which inadvertently ignore

status values from accessing memory outside of the input buffer range.

The returned data which follows the array of status structures can be used directly as input to other DCI routines. For example, *dciListClassId*( ) can return a list of expanded metric class identifiers that can be directly used as input to *dciListInstanceId*( ). This improves application performance by avoiding additional data manipulation for successful requests.

There is no requirement that the DCIReturn structures are returned in the same order as the input metric list nor is there a requirement that any additional return data be in the same order as the DCIReturn structures. It is expected that many implementations will retain the same ordering but there may be an implementation specific performance advantage to writing portions of the return structure out of order.

An application can choose to supply its own buffer for the DCI return structure or have the DCI library allocate a buffer on its behalf. This is done by passing in the address of the return buffer address. If the buffer address is zero then the DCI routine allocates memory and writes the return buffer address. If the buffer address is not zero then the DCI routine writes the return buffer to the given address, limiting the write to the size of the supplied buffer.

In the event that the return buffer supplied is too small to contain the data, the following conditions apply:

- A buffer ≥ sizeof(DCIReturn) but smaller than the required size truncates the data on an even record boundary and alter the count appropriately. The DCI implementation is not under any obligation to provide the partial data (that is, the count could be set to 0), but it must fill in the ‘‘size’’ and ‘‘count’’ fields.
- A buffer ≤ sizeof(DCIReturn) causes the library call to return either DCI_INVALIDARG if the buffer was provided by the caller or DCI_SYSERROR if the library could not allocate the full buffer.

The reason for allowing the choice of a DCI allocated buffer or an application allocated buffer is that in most cases applications would want the DCI service to size and allocate the return structure, especially since the DCI service would know in advance the buffer size, but there are cases where the application needs to manage its own memory. An example would be a metrics archival application which wants to have the results of *dciGetData*( ) calls be written directly to a memory mapped file. If the If DCI routines did their own memory allocation for the return values then the application would have to copy the results to the proper address in the mapped region. By having the interface write the return structure into an application provided address only one write is performed. Thus allowing this choice provides convenience for the typical application while retaining the DCI interface’s flexibility.

Note that is implementation defined whether a DCI call using both wildcards and application allocated buffers performs partial work in the event the buffer is too small. For example, a *dciRemoveInstance*( ) call with wildcarded instance identifiers may require a very large output buffer to hold the dciRetval structures. If the buffer is allocated by the consuming application, it may be too small. It is implementation defined in this situation whether the DCI removes no instances, or only as many instances for which there is room in the buffer to indicate success.

The two data capture routines, *dciGetData*( ) and *dciWaitEvent*( ), allow the consumer to request separate return status and data buffers. This enables metric consumer applications in a data acquisition loop to discard successful, repetitive status returns while archiving the returned data to a sequential buffer. The DCI service allocates buffer space on behalf of the consumer if either the status or the data buffer address is set to zero. If the address of the data buffer address is zero, the DCI service defaults to writing both status and data to the status buffer address. This behaviour is described in the following tables.

If the address of the data buffer address is not zero, then the consumer application has selected split buffers. In this case, the values of the return status address, the data buffer address and the data buffer size are interpreted as follows:

	Input Value of Zero	**Input Value of Non-zero**
return status address	address of server-allocated return buffer caller must use *dciFreei*( ) to free	address of consumer-allocated return buffer
data buffer address	address of server-allocated data buffer caller must use *dciFree*( ) to free	address of consumer-allocated data buffer
data buffer size	size of returned data buffer	size of consumer-allocated data buffer

If the address of the data buffer address is zero, the consumer application has elected to have a single return buffer. In this case, the values of the return status address is interpreted as follows:

	Input Value of Zero	**Input Value of Non-zero**
return status address	address of server-allocated return buffer caller must use *dciFree*( ) to free	address of consumer-allocated return buffer (with data buffer in it).

If a consumer decides to split the status and data buffers then the size field in the DCIReturn structure in the status buffer refers to only the size of the status structure, it does not include the size of the data area. Also, the dataOffset fields in the DCIReturn structure reflect the offset from the beginning of the data buffer. The consumer can examine the returned *dataSize* argument provided with the *dciGetData*( ) and *dciWaitEvent*( ) routines to discover the data buffer size.

Chapter 4

DCI Routines Overview

4.1 Routine Summary and Subset Implementations

Two groupings of the DCI routines are next presented. The first divides the routines into four categories:

- polled metrics consumer routines
- polled metrics provider routines
- event handling routines.
- other routines.

Polled metric consumer routines are those used by metrics consumers to interrogate the name space, and obtain metric values. Polled metrics provider routines are those used by the metrics providers to populate and manage the name space, and make metric values available to the DCI Server. The event handling routines are used by both event providers and consumers to manage the registration and delivery of events. The remaining routines include utilities and administrative functions used by both providers and consumers to: manage memory, manage secure access, manage metric enablement and control event buffering policy. These categories are shown in Table 4-1. Subsequent chapters in this specification cover each category separately.

Consumer	Provider
dciInitialize	dciRegister
dciTerminate	dciUnregister
dciListClassId	dciAddInstance
dciListInstanceId	dciRemoveInstance
dciOpen	dciWaitRequest
dciClose	dciPostData
dciGetClassAttributes	
dciGetInstAttributes	
dciGetData	
dciAddHandleMetric	
dciRemoveHandleMetric	
Event Handling Routines	
dciWaitEvent	
dciPostEvent	
Other Routines	
dciSetClassAccess	
dciSetInstAccess	
dciAlloc	
dciFree	
dciConfigure	
dciSetData	
dciPerror	

Table 4-1 DCI Routines, Grouped by Use

The reason for partitioning the routines into provider and consumer groups is that this enables anyone writing provider or consumer code to focus quickly on the relevant routines. It also enables the design of a consistent set of interfaces within the grouping.

The second grouping of DCI routines, by function type, is presented in Table 4-2. This grouping provides guidelines for a staged implementation of the DCI routines.

With this grouping, the Data Capture Interface can be divided into five subsets:

- basic support
- multiple provider support
- access control support
- event support
- set capability support.

Again, it is intended that, as DCI implementations become widespread and more mature, there will be no subset implementations. If one chooses to do a subset implementation, then the unimplemented library routines should be stubbed and should return the error DCI_NOIMPLEMENTATION in the event they are accessed.

Grouping of the DCI routines by implementation subset is given in the following table:

Basic Support	**Multiple Providers**
dciOpen dciClose dciListClassId dciListInstanceId dciInitialize dciTerminate dciGetData dciGetClassAttributes dciGetInstAttributes dciAddHandleMetric dciRemoveHandleMetric dciConfigure dciFree dciAlloc dciPerror	dciRegister dciAddInstance dciUnregister dciWaitRequest dciPostData dciRemoveInstance
Access Control	**Event Support**
dciSetClassAccess dciSetInstAccess	dciWaitEvent dciPostEvent
Set Capability Support	
dciSetData	

Table 4-2 DCI Routines, Grouped by Implementation Subset

There are a number of reasons why the DCI specification currently describes subset implementations:

1. It lowers the initial implementation cost. One can ship a DCI compliant system quickly and then choose to add functions in the future.

2. It increases portability of the DCI to operating systems which cannot easily implement certain functions. For example, some systems have no notion of access control or could not support event delivery.
3. It allows for implementations which have no underlying operating system support. Tool vendors may want to implement a DCI interface in advance of their system vendors to increase the portability of their tool set.

The following sections provide additional information on how the routines are partitioned and the implications of each subset for the implementor.

4.1.1 Basic Support

Basic support would be the most limited level. It would support the interrogation of a fixed metrics name space and the acquisition of polled data. This level of service is similar to that available on many current systems. There is no provision for the run time modification of the metrics name space or user space providers. The routines that must be present in a basic implementation are:

dciOpen	If the indicated metrics are present in the namespace, a handle is returned. This handle can then be used in subsequent DCI calls (for example, *dciGetData*( ) to refer to the same metrics. The handle provides a guarantee to consumers that any DCI operation using this same handle produce informational return status if a wildcarded class or instance has been newly registered or instantiated.
dciClose	This routine invalidates the handle provided by the *dciOpen*( ) call.
dciListClassId	Allow consumers to look up a list of metric class identifiers in the metrics name space.
dciListInstanceId	Allow consumers to look up a list of instance identifiers in the metrics name space.
dciInitialize	Allow both consumers and providers to tell the DCI implementation that they are initiating a sequence of DCI operations.
dciTerminate	Allow both consumers and providers to tell the DCI implementation that they are terminating a sequence of DCI operations.
dciGetData	Allow consumers to get polled metric data.
dciGetClassAttributes	Allow consumers to acquire the metric class attributes.
dciGetInstAttributes	Allow consumers to acquire the metric instance attributes.
dciAddHandleMetric	Allow consumers to add metrics (polled or event) to an already open handle.
dciRemoveHandleMetric	Allow consumers to remove metrics (polled or event) from an already open handle.
dciConfigure	Allow consumers to transmit configuration requests to providers.
dciFree	Allow consumers to release return buffer and data buffer memory that had been allocated by the server.
dciAlloc	Allow consumers to allocate buffer space for use by the server. Space must be freed with the dciFree call.

dciPerror	Translate a dciStatus value into a corresponding text string.

Implementation of the Basic Support subset is required. The #define symbol _DCI_SUBSET_BASIC must be defined to have value:

```
DCI_SUBSET_BASIC (0x01)
```

(see the **<dci.h>** header file in Appendix A on page 153.

4.1.2 Multiple Providers

The first optional subset allows for multiple providers. For this level, one must implement those routines which allow the providers to modify the name space and register their ability to provide metrics:

dciRegister	Allow providers to register a list of metrics.
dciAddInstance	Allow providers to add an instance to a class.
dciUnregister	Allow providers to unregister a list of metrics.
dciWaitRequest	Allow providers to wait for service requests.
dciPostData	Allow providers to actively transmit polled metrics.
dciRemoveInstance	Allow providers to remove a list of instances from the name space.

Implementation of Multiple Providers is optional. If implemented, the #define symbol _DCI_SUBSET_MULTIPLE_PROVIDERS must be defined to have value:

```
DCI_SUBSET_MULTIPLE_PROVIDERS (0x02)
```

If not implemented, the #define symbol _DCI_SUBSET_MULTIPLE_PROVIDERS must be defined as 0 (see **<dci.h>** header file in Appendix A on page 153).

4.1.3 Access Control

Another optional API subset enables the access control mechanism. Even if the access control routines are in place the implementor can choose to specify how much access control is supported, just as implementations can choose the appropriate security levels. One can imagine implementations that want to streamline access checking in favour of performance. These implementations could check access control only at the group level and either return the [DCI_NOIMPLEMENTATION] reply for attempts to set or get access control information at the other levels or simply silently accept the request. To implement access control one must add the routines:

dciSetClassAccess	Allow the provider to set the default or explicit access control information for a set of metric classes.
dciSetInstAccess	Allow the provider to set the default or explicit access control information for a set of metric instances.

In addition, all consumer routines would be modified to perform access control checks (using the provider-set access control information).

Implementation of Access Control is optional. If implemented, the #define symbol _DCI_SUBSET_ACCESS_CONTROL must be defined to value:

```
DCI_SUBSET_ACCESS_CONTROL (0x04)
```

If not implemented, the #define symbol _DCI_SUBSET_ACCESS_CONTROL must be defined as 0 (see the **<dci.h>** header file in Appendix A on page 153).

4.1.4 Event Delivery Support

Event delivery support extends the DCI to include event metrics (Basic Support provides only for polled metric support). Event delivery requires buffering, as specified in the *dciOpen*( ) call. It facilitates asynchronous delivery of information, for example, the removal or addition of certain instances, as well as tracing: high bandwidth delivery of system performance related events. Event delivery adds the following two calls to the DCI service.

dciWaitEvent A consumer routine to retrieve one or multiple events.

dciPostEvent A provider routine to post an event.

Implementation of Event Support is optional. If implemented, the #define symbol _DCI_SUBSET_EVENT_SUPPORT must be defined to have value:

```
DCI_SUBSET_EVENT_SUPPORT (0x08)
```

If not implemented, the #define symbol _DCI_SUBSET_EVENT_SUPPORT must be defined as 0 (see the **<dci.h>** header file in Appendix A on page 153).

4.1.5 Set Capability

A separate DCI subset, enabling the implementation of other protocols, is the set capability. Although not required for pure measurement oriented systems as defined by UMA, it enables other protocols such as SNMP to make use of the DCI functions. The following call is added to the DCI service.

dciSetData A consumer routine to alter information in one or multiple providers.

Implementation of Set Capability is optional. If implemented, the #define symbol _DCI_SUBSET_SET_CAPABILITY must be defined to have value:

```
DCI_SUBSET_SET_CAPABILITY (0x10)
```

If not implemented, the #define symbol _DCI_SUBSET_SET_CAPABILITY must be defined as 0 (see the **<dci.h>** header file in Appendix A on page 153).

4.2 Routine Status Values

The following is a list of the return values for the Data Capture Interface routines. A return value, of the type DCIStatus, is presented in 3 separate locations for each DCI routine that is issued: in each element of the DCIRetval array (if the particular DCI routine is specified to return such an array), in the DCIReturn structure and as the direct return value of the DCI routine itself. The ‘‘status’’ member of each DCIRetval structure returned indicates the status of the result for the particular MetricId or ClassId referenced by the MetricOffset member. The status DCIReturn is a ‘‘summary’’ status which reflects the status for all DCIRetval structures returned. The return value of the DCI routine represents whether that call succeeded or the specific reason for its failure.

In the latter case, the DCIStatus value returned must be carefully examined before the DCIReturn and DCIRetval structures are referenced. The DCI routine may return ‘‘fatal’’ errors, indicating that the call could not be made and that the DCIReturn structure is not valid and should not be dereferenced. Such ‘‘fatal’’ errors may arise because:

- the call failed because it was not implemented by the host
- the DCI is not currently activated
- the call failed due to some internal system error
- the call failed because the input arguments were malformed
- the call was interrupt or prematurely timed out with no data returned.

In such a case, the caller can expect no DCIReturn data and is obliged to deallocate any memory the DCI Server had allocated for the caller. For example, many of the DCI routines accept a bufferAddress which may be set to zero if the DCI Server should allocate a properly sized memory object. If the contents of bufferAddress were set to NULL before the call and achieved a non-NULL value after the DCI routine which returned the ‘‘fatal’’ DCIStatus, the caller must then deallocate this memory with *dciFree*( ).

In the case of [DCI_SYSERROR], the underlying system had some failure that caused the routine to fail. Such errors might be due to lack of system resources, address faults or other conditions which might be uncontrollable by the application. In this case the DCI routine may succeed if issued a second time (if the error was a transient one), or may continue to return [DCI_SYSERROR]. The underlying error is placed in *errno*; it is implementation-defined how the application may attempt recovery from the error.

There is one exception to the description of ‘‘fatal’’ errors above. If the caller had preallocated a buffer for the return value and it proved too small for the data requested, the DCIStatus returned by the routine is [DCI_NOSPACE]. In this case, the first word of the buffer contains the buffer size that would have held this request. This size could be used to preallocate a larger buffer and retry the call.

It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

Although the DCI routine may have executed properly, the DCIStatus value stored in the *status* member of the DCIReturn structure represents the *summary* of any individual DCIRetval status. If all individual DCIRetval status values are DCI_SUCCESS, then the summary value is [DCI_SUCCESS]. However, if 1 or more of the DCIRetval structures show an error, warning or

informational *status* member, then the summary status will not be [DCI_SUCCESS], but rather represent the most serious severity status among the DCIRetval structures. The summary status then, may contain [DCI_SUCCESS], [DCI_INFORMATIONAL], [DCI_WARNING] or [DCI_FAILURE]. To determine the severity class of any returned status value, the application need only bitwise logically ‘‘and’’ the status with [DCI_SUCCESS], [DCI_INFORMATIONAL], [DCI_WARNING], [DCI_FAILURE] or [DCI_FATAL].

Each DCIRetval structure presents its own status. That status may be classified as success, failure, warning or informational. A failure means that the operation requested of the associated MetricId or ClassId could not be completed. A warning indicates that the operation was well-formed, but could not be completed for some reason. An informational status is one for which the data requested is returned successfully, but with some possible change in state since the last request, or possibly with additional information useful to the application.

Every call that has been passed a valid handle can return an informational status of [DCI_CLASSADDED] or [DCI_INSTANCEADDED]. The [DCI_CLASSADDED] status is generated each time that a wildcarded class id expands into a *new* class. A *new* class is one that has been registered after the metric id had been added to the handle, by *dciOpen*( ) or *dciAddHandleMetric*( ), and has not since returned a [DCI_CLASSADDED] status for this handle. The [DCI_CLASSADDED] status message is returned, in the DCIRetval structure, with and only with the first piece of returned information associated with the new class. Similarly, [DCI_INSTANCEADDED] status is generated each time a wildcarded instance id expands into a *new* instance. A *new* instance is one that was added to a class after the metric id had been added to the handle, using *dciOpen*( ) or *dciAddHandleMetric*( ).

The following fatal errors are returned from the DCI routine itself.

[DCI_ALLOCATIONFAILURE]
[DCI_BADFLAGS]
[DCI_BADHANDLE]
[DCI_INITIALIZED]
[DCI_INTERRUPTED]
[DCI_INVALIDARG]
[DCI_NOIMPLEMENTATION]
[DCI_NOSPACE]
[DCI_NOTINITIALIZED]
[DCI_NOTPRESENT]
[DCI_SYSERROR]

The following status success value is returned from the DCI routine itself:

[DCI_SUCCESS]

The following summary status values are stored in the DCIReturn structure:

[DCI_FAILURE]
[DCI_WARNING]
[DCI_INFORMATIONAL]
[DCI_SUCCESS]

The following errors are returned as individual status:

[DCI_BADCONFIRM]
[DCI_CLASSEXISTS]
[DCI_CLASSNOTEMPTY]
[DCI_CLASSNOTPERSISTENT]
[DCI_DCIMAJORUNSUPPORTED]

[DCI_DCIMINORUNSUPPORTED]
[DCI_DERIVEDDATA]
[DCI_EVENTSUPPORT]
[DCI_INSTANCEEXISTS]
[DCI_INSTANCENOTPERSISTENT]
[DCI_INVALIDDATA]
[DCI_INVALIDFIELD]
[DCI_INVALIDMETHODOP]
[DCI_METHODERROR]
[DCI_METHODOPNOTSUPPORTED]
[DCI_METHODTYPEUNAVAILABLE]
[DCI_NOACCESS]
[DCI_NOCLASS]
[DCI_NODATUMID]
[DCI_NOINSTANCE]
[DCI_NOMETRIC]
[DCI_NOTENABLED]
[DCI_NOTEVENTMETRIC]
[DCI_NOTEXT]
[DCI_NOTPOLLEDMETRIC]
[DCI_NOTQUERYABLE]
[DCI_NOTRESERVABLE]
[DCI_NOTRESERVED]
[DCI_NOTSETTABLE]
[DCI_NOWILDCARD]
[DCI_RESERVED]
[DCI_SUBSETUNSUPPORTED]
[DCI_TIMEOUT]

The following warnings are returned as individual status:

<none defined at this time>

The following informational values are returned as individual status:

[DCI_CLASSADDED]
[DCI_INSTANCEADDED]
[DCI_INVALIDDATAPRESENT]
[DCI_NOSUCHTRANSACTION]

The following success values are returned as individual status:

[DCI_SUCCESS]

The DCIStatus values are defined as follows:

[DCI_ALLOCATIONFAILURE]
The DCI library was requested to provide memory for the return values and could not. The application could attempt to allocate its own memory and try the request again.

[DCI_BADCONFIRM]
The reservation confirmation is either invalid or has expired.

[DCI_BADFLAGS]
The flags argument was malformed, perhaps with conflicting flags specified.

[DCI_BADHANDLE]
The handle provided is not currently valid. The handle must be that returned from a *dciOpen*( ) call.

[DCI_CLASSADDED]
The specified class has been added to the wildcarded set of classes previously opened since the last time this set of classes was expanded. This status is transient and is not returned the next time the expanded set is expanded into this class.

[DCI_CLASSEXISTS]
This class could not be registered because it already exists.

[DCI_CLASSNOTEMPTY]
Class could not be removed because either there are subclasses still registered, or instances still defined.

[DCI_CLASSNOTPERSISTENT]
The parent class is not persistent and the associated DCIClassAttr specifies persistence.

[DCI_DCIMAJORUNSUPPORTED]
The specified DCI Version major number is not supported by the DCI Server.

[DCI_DCIMINORUNSUPPORTED]
The specified DCI Version major number is supported by the DCI Server, but the specified minor version number is not.

[DCI_DERIVEDDATA]
An attempt was made to retrieve (or set) a metric that has type DCI_DERIVED. Derived data cannot be retrieved; instead, the metric's DCIDatumId should be examined to determine which related metrics should be obtained, and what kind of computation should be performed to actually obtain the desired value.

[DCI_EVENTSUPPORT]
The consumer has attempted to open an event metric with format and content requirements beyond those supported by the registering provider. The provider will form the conjunction ("and") of the provided bit map with the registered bit map to determine the format and content of event data for this event metric.

[DCI_FAILURE]
Summary status for the case where at least 1 returned individual status was an error.

[DCI_FATAL]
The called DCI routine could not complete due to input argument or system errors and any associated DCIReturn may be incomplete and should not be referenced.

[DCI_INFORMATIONAL]
Summary status for the case where at least 1 returned individual status was informational and there were no individual error or warning status values returned.

[DCI_INITIALIZED]
The DCI subsystem is already initialised.

[DCI_INSTANCEEXISTS]
This instance could not be added because it already exists.

[DCI_INSTANCEADDED]
The specified instance has been added to the wildcarded set of instances previously opened since the last time this set of instances was expanded. This status is transient and is not returned the next time the expanded set is expanded into this instance.

[DCI_INSTANCENOTPERSISTENT]
The parent class is not persistent and the associated DCIInstAttr specifies persistence.

[DCI_INTERRUPTED]
This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

[DCI_INVALIDARG]
One or more of the input arguments to the DCI routine were malformed.

[DCI_INVALIDDATA]
An attempt was made to retrieve (or set) a metric that is invalid for this particular class instance.

[DCI_INVALIDDATAPRESENT]
The associated class of data referenced contains at least one invalid metric. Each metric must be examined before use.

[DCI_INVALIDFIELD]
One or more of the associated argument structures contained an invalid field specification.

[DCI_INVALIDMETHODOP]
The method operation specified is not a valid operation.

[DCI_METHODERROR]
An error has occurred in the method invoked to retrieve or set the requested metric.

[DCI_METHODOPNOTSUPPORTED]
The method operation may not be specified in conjunction with this method type.

[DCI_METHODTYPEUNAVAILABLE]
The specified method ‘‘type’’ member is one of those documented in this specification, but which is not available on this platform. Only DCI_WAIT is guaranteed to be available on all implementations.

[DCI_NOACCESS]
The calling process does not have permission to retrieve information about the requested metric or to initialise a connection to the DCI server.

[DCI_NOCLASS]
The requested metric class identifier is not present in the name space.

[DCI_NODATUMID]
The specified *datumId* does not exist.

[DCI_NOIMPLEMENTATION]
In a DCI subset implementation, the specified routine has not been implemented.

[DCI_NOINSTANCE]
There is no such instance identifier in the name space.

[DCI_NOMETRIC]
There is no such metric identifier in the name space.

[DCI_NOSPACE]
The provided buffer is too small for the return structure.

[DCI_NOSUCHTRANSACTION]
The specified transactionId is either invalid or the associated transaction was cancelled.

[DCI_NOTENABLED]
The requested metric is currently not enabled by its provider.

[DCI_NOTEXT]
No label has been specified with metric.

[DCI_NOTEVENTMETRIC]
An event metric was required and the requested metric identifier was not for such a metric type.

[DCI_NOTINITIALIZED]
The DCI subsystem is not currently initialised.

[DCI_NOTPOLLEDMETRIC]
A polled metric was required and the requested metric identifier was not for such a metric type.

[DCI_NOTPRESENT]
The DCI service is not available.

[DCI_NOTQUERYABLE]
The associated metric does not support being queried.

[DCI_NOTRESERVABLE]
The associated metric does not support being reserved.
[DCI_NOTRESERVED]
The associated metric could not be released because it was not already reserved.
[DCI_NOTSETTABLE]
The associated metric does not support being set.
[DCI_NOWILDCARD]
A wildcard cannot be used in this context.
[DCI_RESERVED]
The associated metric is already reserved by another consumer.
[DCI_SUBSETUNSUPPORTED]
One or more of the specified DCI API subsets are not supported by the DCI Server.
[DCI_SUCCESS]
The specified request was free of errors, warnings or informational return status.
[DCI_SYSERROR]
An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_TIMEOUT]
The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.
[DCI_WARNING]
Summary status for the case where at least 1 returned individual status was a warning and there were no individual error status values returned.

Chapter 5

Metrics Consumer Routines

This Chapter describes the interfaces used by metrics consumers.

NAME

dciAddHandleMetric - adds metrics to an open handle

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciAddHandleMetric(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	A handle returned from *dciOpen*( ) that has not been subsequently closed.
metricIdList	Address of a list of metric identifiers.
numIds	The number of input metric identifiers.
bufferAddress	Points to the address of a return status buffer.
bufferSize	The size of the return status buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciAddHandleMetric*( ) blocks indefinitely.

DESCRIPTION

The *dciAddHandleMetric*( ) adds additional metrics to an open handle. Added metrics will be subject to the same buffering policies that are currently in use in the open handle.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciAddHandleMetric*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero when *dciAddHandleMetric*( ) is called, then *dciAddHandleMetric*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciAddHandleMetric*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciAddHandleMetric*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.

[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for numIds, bufferSize is smaller than the size of a DCIReturn structure, or metricIdList was malformed.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NODATUMID]	The associated metricId specified a nonexistent datumId for the specified class.
[DCI_NOINSTANCE]	One or more requested instance identifiers are not present in the name space.

[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOTQUERYABLE]	The specified metric identifier could not be accessed.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

NAME

dciAlloc - allocate memory which can be destroyed with dciFree

SYNOPSIS

```
#include <sys/dci.h>

void *dciAlloc(
    UMAUint4 size                          /* in */
);
```

ARGUMENTS

size size in bytes of memory to be allocated.

DESCRIPTION

dciAlloc( ) will create a memory object of at least the size specified. The memory returned by this call must be deallocated using *dciFree*( ), and is indistinguishable from memory allocated by the DCI server on behalf of the caller.

If *dciAlloc*( ) is issued with a size of 0 bytes, of if the data could not be returned, a (void *)0 is returned.

RETURN VALUES

dciAlloc( ) will either return the address of a memory object of the proper size, or a (void *)0 to indicate failure. No other error returns are specified.

NAME

dciClose - close a metrics list

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciClose(
    DCIHandle       handle                  /* in */
);
```

ARGUMENTS

handle Handle that was returned from a prior *dciOpen*( ) call.

DESCRIPTION

The *dciClose*( ) routine is the counterpart of *dciOpen*( ). It closes the handle that is associated with a list of metrics. Subsequent attempts to use the closed handle in any DCI call will fail.

dciClose( ) will flush any events pending on the handle. If the consumer wishes to capture these events, then a *dciWaitEvent*( ) should be issued prior to the call to *dciClose*( ).

RETURN VALUES

The *dciClose*( ) routine returns [DCI_SUCCESS] if the handle was open. Otherwise, this routine returns one of the following error values:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_BADHANDLE]	The handle provided is not currently open.

NAME

dciConfigure - send configuration information to provider

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciConfigure(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    DCIConfig       *configList,            /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	Handle that was returned from a prior *dciOpen*( ) of metricIdList or a superset of metricIdlist.
metricIdList	Address of a list of DCIMetricId structures.
configList	List of DCIConfig structures, one per input DCIMetricId.
numIds	The number of DCIMetricId structures in *metricIdList*. It is also the number of DCIConfig structures in *configList*.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciConfigure*( ) blocks indefinitely.

DESCRIPTION

The *dciConfigure*( ) routine provides a communication channel between the consumer and its metrics providers. A metrics consumer can send a DCIConfig structure to the providers that deliver the metrics in *metricIdList*. Each DCIConfig structure in the configList corresponds to a DCIMetricId structure in the metricIdList. A handle for the *metricIdList* must be provided.

The *metricIdList* supports two special identifiers in the *dciConfigure*( ) call. A zero length instance identifier means ‘‘all current and future instances’’ of the given class(es). A datum identifier with the value DCI_ALL means "all current data" of the given class(es). By combining these two special identifiers, the requested operation is effectively performed on the given class(es).

The DCIConfig structure allows consumers to ask providers to enable or disable metrics, to set buffer policies in the DCI Server, and to transmit provider specific information in an opaque wrapper provided by the DCIConfig structure. This wrapper is given by the following structure:

```
typedef struct DCIConfig {
    UMAUint4         size;       /* total structure size, in bytes   */
    UMAUint4         flags;      /* configuration request            */
    UMAElementDescr  configData; /* descriptor for the               */
                                 /* auxiliary config info            */
    UMAVarLenData    data;       /* auxiliary config data start here */
} DCIConfig;
```

The bits in the lower half of the flags field are reserved for the DCI interface and those in the upper half can be locally defined. The following flags are defined:

DCI_ENABLE
DCI_DISABLE
DCI_BUFFER_EVENTS_DISCARD
DCI_BUFFER_EVENTS_OVERWRITE
DCI_BUFFER_EVENTS_SETSIZE
DCI_BUFFER_EVENTS_GETSIZE
DCI_BUFFER_EVENTS_GETPOLICY
DCI_CONFIGURATION

DCI_ENABLE and *DCI_DISABLE* are administrative commands that instruct the DCI Server to force an entire class (and its corresponding metrics) to be turned on or off. No reference counts are implied; a *DCI_ENABLE*, or *DCI_DISABLE* request is performed without regard to any other ongoing consumer activity relating to the specified metrics. If a class is disabled, the DCI Server refuses requests for any metric in that class. This operation is typically restricted to administrators (however that may be enforced). These two flags are mutually exclusive.

DCI_BUFFER_EVENTS_DISCARD and *DCI_BUFFER_EVENTS_OVERWRITE* instruct the DCI Server of the type of buffering policy to use for its internal event buffer. *DCI_BUFFER_EVENTS_DISCARD* instructs the DCI Server to drop new events when the internal event buffer is full. *DCI_BUFFER_EVENTS_OVERWRITE* instructs the DCI Server to overwrite the oldest event(s) in the buffer when a new event arrives and the buffer is full. The internal buffer is only of importance when a consumer does not have any outstanding *dciWaitEvent*( ) requests for a particular handle. The corresponding metric identifier is ignored for this request. These two flags are mutually exclusive.

DCI_BUFFER_EVENTS_SETSIZE and *DCI_BUFFER_EVENTS_GETSIZE* set and retrieve the size of the DCI Server internal event buffer, respectively. The size is encoded as a UMAUint4 in the data field of the DCIConfig structure, with the UMAElementDescr filled out accordingly. The corresponding metric identifier is ignored for this request. These two flags are mutually exclusive.

DCI_BUFFER_EVENTS_GETPOLICY retrieves the current buffering policy from the DCI Server for this handle. The corresponding metric identifier is ignored for this request. The actual policy is returned as the corresponding bitflag in the flags field of the DCIConfig structure.

DCI_CONFIGURATION requests the current configuration of the class or class instances indicated in the metricIdList. Datum identifiers are ignored by this request. The actual configuration information is returned in the DCIConfig structure. Currently, the following flags are supported:

DCI_ENABLE DCI Server allows requests

DCI_DISABLE DCI Server disallows requests

Any other configuration information can be encoded in the upper two words of the flags field and the opaque data section of the DCIConfig structure. This information is provider specific and is not part of the DCI specification.

For all the above requests the *dciConfigure*( ) routine returns a per-metric status, if appropriate, in the return buffer using the standard DCIReturn structure.

If the return buffer address, *bufferAddress*, is zero when *dciConfigure*( ) is called, then *dciConfigure*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciConfigure*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

RETURN VALUES

The *dciConfigure*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciConfigure*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.

[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for numIds, bufferSize is smaller than the size of a DCIReturn structure, the metricIdList was malformed, metricIdList and classIdList are NULL, metricIdList and classIdlist are both non-NULL.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
[DCI_CLASSES_CHANGED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCES_CHANGED]	This new instance has been added within the scope of a wildcarded instance request.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.
[DCI_INTERRUPTED]	The *dciConfigure*( ) call was interrupted by a signal and did not complete.
[DCI_NOTENABLED]	The associated metric is currently not enabled by its provider.

NAME

dciFree - release memory allocated by the DCI

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciFree(
    void            *ptr                    /* in */
);
```

ARGUMENTS

ptr	The address of memory allocated by the DCI on behalf of the caller.

DESCRIPTION

Several DCI routines have a calling sequence which allows the DCI application to provide a memory buffer or ask that the DCI itself allocated a suitably sized memory buffer. In either case, it is the obligation of the program to eventually return the memory to the appropriate allocation pool. If the DCI has successfully allocated the memory for the caller, then the caller must call *dciFree*( ) in order for that memory to be released. Note that even if a DCI call has failed, any memory allocated by the DCI must be freed by the caller using *dciFree*( ).

If the argument to *dciFree*( ) is NULL, then *dciFree*( ) immediately returns.

dciFree( ) should not be called following a *dciTerminate*( ) call.

RETURN VALUES

dciFree( ) may not check its input arguments for consistency and may return no error based on its argument.

The following errors may be returned:

[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.

NAME

dciGetClassAttributes - acquire metric class attributes

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciGetClassAttributes(
    DCIHandle       handle,                 /* in */
    DCIClassId      *classIdList,           /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize              /* in */
);
```

ARGUMENTS

handle	Handle that was returned from a prior *dciOpen*( ) of *classIdList* or a superset of *classIdlist*.
classIdList	Address of a list of DCIClassId structures.
numIds	The number of DCIClassId structures in *classIdList*.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciGetClassAttributes*( ) routine is used to retrieve the attribute structures for a list of metric classes. The metric class identifiers can be optionally wildcarded. An optional handle may be provided for the *classIdList*: if the *handle* is valid and if *classIdList* contains wildcarded classes, then all metric classes of *classIdList* are confirmed to be a subset of the opened metric classes represented by *handle* and any newly registered classes is marked with a special informational status, *DCI_CLASSES_CHANGED*, in the associated DCIRetval structure. The *classIdList* is a subset of the opened metrics associated with handle. If the *handle* provided is 0 then no such confirmation or informational status is provided.

The *dciGetClassAttributes*( ) routine expands any metric class wildcards appearing in *classIdList* and determines if each expanded metric class is currently registered in the metrics name space. For each expanded metric class, a DCIRetval reply is created in the DCIReturn structure. The *metricOffset* member of each DCIRetval references the expanded DCIClassId structure and, if available, the *dataOffset* member references a DCIClassAttribute for the associated, registered metric class.

If the return buffer address, *bufferAddress*, is zero when *dciGetClassAttributes*( ) is called, then *dciGetClassAttributes*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciGetClassAttributes*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure could be written. Otherwise, *dciGetClassAttributes*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application.

A vendor-specific error code is placed in the variable *errno*.

[DCI_NOSPACE] The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer.

It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.

[DCI_BADHANDLE] The handle provided is not currently open.

[DCI_INVALIDARG] One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *classIdList* was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOACCESS] The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_CLASSES_CHANGED] This new class has been added within the scope of a wildcarded class request.

NAME

dciGetData - get polled metric data

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciGetData(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    void            **dataAddress,          /* in/out */
    UMAUint4        *dataSize,              /* in/out */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	Handle that was returned from a prior *dciOpen*( ) of *metricIdList* or a superset of *metricIdlist*.
metricIdList	Address of a list of DCIMetricId structures.
numIds	The number of DCIMetricId structures in metricIdList.
bufferAddress	Points to the address of a return status buffer.
bufferSize	The size of the return status buffer.
dataAddress	Points to the address of an optional return data buffer.
dataSize	Address of the size of the optional return data buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciGetData*( ) blocks indefinitely.

DESCRIPTION

The *dciGetData*( ) routine is used to acquire polled metrics data. *dciGetData*( ) expands any metric class and metric instance wildcards appearing in *metricIdList* and determines if each expanded metric class is currently registered in the metrics name space (and represents a subset of the metrics represented by the handle) and if each expanded instance is also registered. For each expanded metric identifier, a DCIRetval reply is created in the DCIReturn structure stored in the return buffer pointed to by **bufferAddress*. The *metricOffset* member of each DCIRetval specifies a location relative to **bufferAddress* and references the expanded DCIMetricId structure. The *dataOffset* member of each DCIRetval specifies a location relative the *dataAddress* argument (if the contents of *dataAddress* is 0, then the library allocates a buffer on behalf of the application) if split data and return information is specified. If split data and return information is not specified, then *dataOffset* is considered relative to *bufferAddress*. In either case, *dataOffset* refers to the requested data returned.

Note that the *dciGetData*( ) routine can optionally separate the data and status return buffers. This allows applications in a polled data acquisition loop to archive successfully acquired data and discard the status structure. To indicate that this separation is desired, the application must provide a non-NULL *dataAddress* argument. If applications separate the buffers, then the size of the data buffer is returned at the *dataSize* address and the *size* field of the DCIReturn structure is the size of the structure written to *bufferAddress*.

If the return buffer address, *bufferAddress*, is zero when *dciGetData*( ) is called, then *dciGetData*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

dciGetData( ) accepts both class and instance level wildcards. If wildcards are provided, then informational status indicates whether a new class or instance has been added since the last such DCI routine was issued. The *datumId* may specify a single data item to be retrieved or, if the value of *datumId* is *DCI_ALL* then an entire class worth of data is returned. The application references the appropriate DCIClassAttr structures to determine that size and type of each piece of data, and in the case of the wildcarded *datumId*, the offset of each piece of data within the whole class of data returned. When the whole class of data is returned, variable length data values (such as those with UMADataType of *UMA_TEXTSTRING*) are handled specially; in this case, the data retrieved from the specified offset is itself an offset to the actual data. This is necessary to ensure that all data can be obtained from fixed offsets when the whole class is returned. The indirection for variable length data is not needed when *datumId* in not wildcarded.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciGetData*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

RETURN VALUES

The *dciGetData*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer and the data was written to the data buffer. Otherwise, *dciGetData*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.

[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, *metricIdList* was malformed, *timeout* was malformed or the address of *dataSize* was not provided and *dataAddress* was specified and set to a NULL.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
[DCI_NODATUMID]	The associated metricId specified a nonexistent datumId for the specified class.
[DCI_CLASSES_CHANGED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCES_CHANGED]	This new instance has been added within the scope of a wildcarded instance request.
[DCI_NOT_POLLEDMETRIC]	The associated metricId specified a datumId which does not correspond with a polled metric.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.
[DCI_NOTENABLED]	The associated metric is currently not enabled by its provider.
[DCI_INVALIDDATA]	The specified metric for the associated instance could not be returned because it is not valid.

[DCI_INVALIDDATAPRESENT]	The associated class of data referenced contains at least one invalid metric. Each metric must be examined before use.
[DCI_DERIVEDDATA]	An attempt was made to retrieve the value for a metric whose type is DCI_DERIVED.
[DCI_NOTQUERYABLE]	The specified metric identifier could not be accessed.
[DCI_METHODERROR]	The DCI Server has encountered an error in the method invoked to satisfy the request for the selected metric.

NAME

dciGetInstAttributes - acquire metric instance attributes

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciGetInstAttributes(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	Handle that was returned from a prior *dciOpen*( ) of *metricIdList* or a superset of *metricIdlist*.
metricIdList	Address of a list of DCIMetricId structures.
numIds	The number of DCIMetricId structures in metricIdList.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciGetInstAttributes*( ) blocks indefinitely.

DESCRIPTION

The *dciGetInstAttributes*( ) routine is used to retrieve the attribute structures for a list of instances. The metric identifiers can be optionally wildcarded for both class or instance. The *datumId* field of the DCIMetricId is ignored. An optional *handle* may be provided for the *metricIdList*: if the *handle* is valid and if *metricIdList* contains wildcarded classes, then all metric classes of *metricIdList* are confirmed to be a subset of the opened metric classes represented by handle and any newly registered classes is marked with a special informational status, *DCI_CLASSES_CHANGED*, in the associated DCIRetval structure. Similarly, if there are instance wildcards and a valid *handle*, then each expanded instance is confirmed to be within a subset of metric identifiers represented by the *handle* and any newly added instances is marked with a special informational status, *DCI_INSTANCES_CHANGES*. If the handle provided is 0 then no such confirmation or informational status is provided.

dciGetInstAttributes( ) expands any metric class and metric instance wildcards appearing in *metricIdList* and determines if each expanded metric class is currently registered in the metrics name space and if each expanded instance is also registered. For each expanded metric identifier a DCIRetval reply is created in the DCIReturn structure stored in the return buffer. The *metricOffset* member of each DCIRetval references the fully expanded DCIMetricId structure. The *dataOffset member* of each DCIRetval is set to an array of DCIInstAttr structures. The number of elements in this returned DCIInstAttr array is stored in the *count* member of the DCIRetval structure.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciGetInstAttributes*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not

subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero when *dciGetInstAttributes*( ) is called, then *dciGetInstAttributes*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciGetInstAttributes*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciGetInstAttributes*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *metricIdList* was malformed.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
[DCI_CLASSES_CHANGED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCES_CHANGED]	This new instance has been added within the scope of a wildcarded instance request.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

NAME

dciInitialize - establish a connection to the Data Capture Interface

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciInitialize(
    DCIVersion    *request,     /* in */
    DCIVersion    *response     /* out */
);
```

ARGUMENTS

request The structure holding the requested version.

response The structure holding the actual version structure returned by the DCI Server.

DESCRIPTION

dciInitialize( ) is required for all DCI applications; it establishes a connection with the DCI Server, performing any implementation specific initialization needed. No other DCI API call may be successfully issued unless *dciInitialize*( ) returns DCI_SUCCESS. Use *dciTerminate*( ) to later disconnect from the DCI Server.

The application can request that the DCI Server provide a connection to the appropriate DCI API Version number, thus ensuring that all data structures and API semantics for the expected DCI Version are observed. Unless a fatal error occurs, the current version of the DCI Server and other information is returned in the *response* structure. The *request* argument is optional and may be replaced with a NULL pointer; in this case, the DCI library will provide a description of the current API version. The *response* argument is optional and may be replaced with a NULL pointer.

The *DCIVersion* structure is defined as follows:

```
/*
 * DCI version structure.
 * This is passed into the dciInitialize() call as a request
 * structure. As such, it specifies a request to connect to
 * a specific DCI API version. A DCI version structure is
 * also passed as an output parameter indicating the level
 * of support that this particular DCI implementation
 * is actually making available.
 */
typedef struct DCIVersion {
    UMAUint4     DCIMajorVersion;
    UMAUint4     DCIMinorVersion;
    UMAUint4     DCISubsetMask;
    UMAUint4     DCIVendorExtensions;
} DCIVersion;
```

The *DCIMajorVersion* and *DCIMinorVersion* fields indicate the specification level of the DCI to which this implementation corresponds. The *DCISubsetMask* field is a bitmask indicating which DCI subsets are implemented; it is formed by "OR"ing the appropriate DCI_SUBSET_* constants together (see below). The *DCIVendorExtensions* field is implementation defined, and may be used to indicate refinements on the implementation level (for example, if a vendor produces multiple implementations of the DCI, this field can be used to distinguish those implementations).

The *DCISubsets* structure is a bitmask describing which of the DCI API subsets are present in a particular implementation. Chapter 4 describes the possible subsets. The corresponding values for the bitmask are defined as follows:

```
DCI_SUBSET_BASIC                = 0x01
DCI_SUBSET_MULTIPLE_PROVIDERS   = 0x02
DCI_SUBSET_ACCESS_CONTROL       = 0x04
DCI_SUBSET_EVENT_SUPPORT        = 0x08
DCI_SUBSET_SET_CAPABILITY       = 0x10
```

For example, if a consumer MAP is created using DCI API Version 1.0, it can call *dciInitialize*( ) as follows:

```
DCIVersion_1 wantedDCIversion, receivedDCIversion;
DCIStatus status;

bzero(&wantedDCIversion, sizeof(wantedDCIversion));
wantedDCIversion.DCIMajorVersion = DCI_MAJORVERSION;
wantedDCIversion.DCIMinorVersion = DCI_MINORVERSION;
wantedDCIversion.DCISubsetMask = DCI_SUBSET_BASIC;

status = dciInitialize(&wantedDCIversion, &receivedDCIversion);

if (status & DCI_FATAL) {
                              /* could not make the call */
}
if (status & DCI_FAILURE) {
                     /* these could be due to version # */
}
if (status == DCI_SUBSETUNSUPPORTED) {
        /* the specified subset was not fully supported */
}
if (status & DCI_SUCCESS) {
    /*
     * check which subsets are enabled or examine the
     * vendor specific flags. Cast the response structure
     * to the appropriate data structure based on the
     * major number returned.
     */
}
```

RETURN VALUES

The *dciInitialize*( ) routine returns [DCI_SUCCESS] if it was able to establish a connection to the metric provider or consumer. Otherwise, this routine returns one of the following error values:

[DCI_INITIALIZED]	The DCI subsystem is already initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOACCESS]	The calling process does not have permission to initialise a connection to the DCI Server.
[DCI_MAJORUNSUPPORTED]	The specified DCI Version major number cannot be provided by the DCI Server. The DCIMajorVersion field of *response*

	represents the supported DCI version.
[DCI_MINORUNSUPPORTED]	The specified DCI Version major number can be provided, but the specified minor number could not. The DCIMinorVersion field of *response* represents the supported DCI version.
[DCI_SUBSETUNSUPPORTED]	One or more of the specified DCI subsets are not available. The DCISubsetMask in the *response* represents the fully supported DCI subsets. A subset can be partially supported, but *dciInitialize*( ) only reports complete support.
[DCI_NOTPRESENT]	The DCI service is not available.

NAME

dciListClassId - look up a list of metric class identifiers in the metrics name space

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciListClassId(
    DCIHandle       handle,                 /* in */
    DCIClassId      *classIdList,           /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize              /* in */
);
```

ARGUMENTS

handle	The handle returned from *dciOpen*( ). It is optional and if 0, this argument is ignored.
classIdList	Address of a list of DCIClassId structures.
numIds	The number of DCIClassId structures in *classIdList*.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciListClassId*( ) routine provides a list of registered classes contained in *classIdList*. The metric class identifiers can be optionally wildcarded. *dciListClassId*( ) expands any metric class wildcards appearing in *classIdList* and determines if each expanded metric class is currently registered in the metrics name space. For each expanded metric class, a DCIRetval reply is created in the DCIReturn structure. The *metricOffset* member of each DCIRetval references the expanded DCIClassId structure. The *dataOffset* member of each DCIRetval is set to 0.

If the return buffer address, *bufferAddress*, is zero when *dciListClassId*( ) is called, then *dciListClassId*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciListClassId*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciListClassId*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error

(for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.

[DCI_BADHANDLE] The handle provided is not currently open.

[DCI_INVALIDARG] One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *classIdList* was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOACCESS] The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_CLASSES_CHANGED] This new class has been added within the scope of a wildcarded class request.

[DCI_INSTANCES_CHANGED] This new instance has been added within the scope of a wildcarded instance request.

NAME

dciListInstanceId - look up a list of instance identifiers in the metrics name space

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciListInstanceId(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	The handle returned from *dciOpen*( ). It is optional and if 0 this argument is ignored.
metricIdList	Address of a list of DCIMetricId structures.
numIds	The number of DCIMetricId structures in *metricIdList*.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciListInstanceId*( ) blocks indefinitely.

DESCRIPTION

The *dciListInstanceId*( ) routine provides a list of valid instances contained in *metricIdList*. The metric class identifiers can be optionally wildcarded for both class or instance. The *datumId* field of the DCIMetricId is ignored. *dciListInstanceId*( ) expands any metric class and metric instance wildcards appearing in *metricIdList* and determines if each expanded metric class is currently registered in the metrics name space and if each expanded instance is also registered. For each expanded metric identifier, a DCIRetval reply is created in the DCIReturn structure stored in the return buffer. The *metricOffset* member of each DCIRetval references the fully expanded DCIMetricId structure. The *dataOffset* member of each DCIRetval is set to an array of DCIMetricId structures for each explicit instance. The number of elements in this returned DCIMetricId array is stored in the *count* member of the DCIRetval structure.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciListInstanceId*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero when *dciListInstanceId*( ) is called, then *dciListInstanceId*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciListInstanceId*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciListInstanceId*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *metricIdList* was malformed.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.
[DCI_CLASSES_CHANGED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCES_CHANGED]	This new instance has been added within the scope of a wildcarded instance request.

NAME

dciOpen - open a list of metrics and obtain a handle

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciOpen(
    DCIHandle       *handle,                /* out */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMAUint4        handleFlags,            /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	A pointer to a location to return the handle.
metricIdList	Address of a list of metric class identifiers.
numIds	The number of input metric identifiers.
bufferAddress	Points to the address of a return status buffer.
bufferSize	The size of the return status buffer.
handleFlags	Bitmapped flags as described below.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciOpen*( ) blocks indefinitely.

DESCRIPTION

The *dciOpen*( ) routine instructs the Data Capture service to perform an access check for every metric in *metricIdList*. If successful, a *handle* is returned which can be used in subsequent operations to access all or some of the metrics in *metricIdList*.

Metrics can be dynamically added to or deleted from a handle using *dciAddHandleMetric*( ) and *dciRemoveHandleMetric*( ).

The *dciOpen*( ) routine also allows the specification of a handle specific buffering scheme for provider generated events. Event buffering may be desired when a consumer wants to minimise loss of events that may be generated while it is off doing other work and not blocked in a *dciWaitEvent*( ) call.

handleFlags contains one or more bitmapped flags which may be set to specify a buffering scheme for event metrics. If no flags are specified, no buffering is performed. The buffering scheme is on a per handle basis. The initial buffer size is system dependent but is queryable and settable using *dciConfigure*( ).

The following values may be set in *handleFlags*:

DCI_BUFFER_EVENTS_DISCARD
DCI_BUFFER_EVENTS_OVERWRITE

DCI_BUFFER_EVENTS_DISCARD and DCI_BUFFER_EVENTS_OVERWRITE instruct the DCI Server of the type of buffering policy to use for its internal event buffer. DCI_BUFFER_EVENTS_DISCARD instructs the DCI Server to drop new events when the

internal event buffer is full. DCI_BUFFER_EVENT_OVERWRITE instructs the DCI Server to overwrite the oldest event(s) in the buffer when a new event arrives and the buffer is full. The internal buffer is only of importance when a consumer does not have any outstanding *dciWaitEvent*( ) requests for a particular handle. The corresponding metric identifier is ignored for this request. These two flags are mutually exclusive.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciOpen*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero when *dciOpen*( ) is called, then *dciOpen*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciOpen*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciOpen*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *metricIdList* was malformed.
[DCI_BADFLAGS]	Two or more mutually exclusive flags were used together.

[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_EVENTSUPPORT]	The consumer has attempted to open an event metric with format and content requirements beyond those supported by the registering provider. The provider will form the conjunction ("and") of the provided bit map with the registered bit map to determine the format and content of event data for this event metric.
[DCI_NOCLASS]	No requested metric class identifier is present in the name space.
[DCI_NODATUMID]	The associated metricId specified a nonexistent datumId for the specified class.
[DCI_NOINSTANCE]	No requested instance identifier is present in the name space.
[DCI_NOACCESS]	There was an access permission failure for at least one request.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.
[DCI_NOTQUERYABLE]	The specified metric identifier could not be accessed.

NAME

dciPerror - produce an error message based on DCIStatus or errno

SYNOPSIS

```
#include <sys/dci.h>

void dciPerror(
    DCIStatus status,       /* in */
    int theerrno,           /* in (optional) */
    char *membuf,           /* in (optional) */
    int bufsize,            /* in */
    char *fmt, ...          /* in */
);
```

ARGUMENTS

status	A valid DCIStatus return value.
theerrno	The standard "_errno" (optional).
membuf	If present, the returned string is copied to this buffer.
bufsize	Maximum size in bytes of the membuf buffer.
fmt,	*A printf format string with a variable length parameter list.*

DESCRIPTION

dciPerror( ) will accept a DCIStatus value and lookup the corresponding text string describing the status. If the status is DCI_SYSERROR, then the argument "theerrno" is examined and the corresponding text string for that error is produced.

If membuf is supplied, then the output produced is copied to the buffer as a null terminated string up to bufsize bytes in length. If membuf is not present, the produced string is sent to stderr output.

fmt is a character string such as that used in *printf*( ). It may include format specifiers as a variable number of arguments (up to some fixed limit). The output produced includes the formatted output for *fmt* and then a ‘‘:’’ (colon) and the text string representing the error in question.

For example, if *myclassname*( ) existed and would produce a class expressed as the symbolic character string, "datapool.mem.bufcache":

```
dciPerror(DCI_CLASSEXISTS, 0, (char *)0, 0, "Opening class %s",
   myclassname(theclass))
```

would produce the following on stderr:

```
Opening class datapool.mem.bufcache:
    The DCI class is already registered.
```

LIMITATIONS

If membuf is not present, the string produced may be truncated to a fixed size, no smaller than 1024 bytes.

The number of arguments available to "fmt" may be limited, but that limit must be no less than 8.

RETURN VALUES

There are no return values for this routine.

NAME

dciRemoveHandleMetric - removes metrics from an open handle

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciRemoveHandleMetric(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle	A handle returned from *dciOpen*( ) that has not been subsequently closed.
metricIdList	Address of a list of metric identifiers.
numIds	The number of input metric identifiers.
bufferAddress	Points to the address of a return status buffer.
bufferSize	The size of the return status buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciRemoveHandleMetric*( ) blocks indefinitely.

DESCRIPTION

The *dciRemoveHandleMetric*( ) routine performs the inverse of *dciAddHandleMetric*( ). It disassociates and closes a list of metrics from an open handle. Note that a metric must be deleted from a handle using the exact same metricId with which it was opened or added. For example, it is not possible to open or add a group of metrics using a wildcarded datumId and then use *dciRemoveHandleMetric*( ) to delete only a single datumId from that group. The individual datum ids should be added first, then the wildcarded datum id should be removed. Doing otherwise may result in the metric being reset when it is closed.

If the return buffer address, *bufferAddress*, is zero when *dciRemoveHandleMetric*( ) is called, then *dciRemoveHandleMetric*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciRemoveHandleMetric*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

RETURN VALUES

The *dciRemoveHandleMetric*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer and the data was written to the data buffer. Otherwise, *dciRemoveHandleMetric*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *metricIdList* was malformed.
[DCI_BADHANDLE]	The given handle is not valid.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOCLASS]	The requested metric class identifier is not present in the handle.

[DCI_NODATUMID]	The associated metricId specified a nonexistent datumId for the specified class.
[DCI_NOINSTANCE]	One or more requested instance identifiers are not present in the handle.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_CLASSESADDED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCESADDED]	This new instance has been added within the scope of a wildcarded instance request.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

NAME

dciSetData - request a provider to set a metric.

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus  dciSetData(
    DCIHandle       handle,                 /* in */
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    UMAUint4        operation,              /* in */
    UMAUint4        *pConfirm,              /* in/out */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        *bufferSize,            /* in */
    void            *dataAddress,           /* in */
    UMAUint4        dataSize,               /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

handle An open handle containing the metrics to be set.

metricIdList Address of a list of polled metric identifiers, each of which must have been opened in the handle argument. This list specifies the subset of the handle metrics on which the set operation is to be performed. An empty list (numIds equal to zero) means that *dciSetData*( ) returns without setting any metrics.

numIds The number of input metric identifiers in the metricIdList.

operation This tells which operation to perform and must be one of *DCI_OP_RESERVEDATA*, *DCI_OP_SETDATA*, or *DCI_OP_RELEASEDATA*.

pConfirm If operation is *DCI_OP_RESERVEDATA*, *pConfirm was zero, and the reservation is successful for any metric in the metricIdList, *pConfirm is set to a reservation confirmation that can be used on a successive call to *dciSetData*( ). If *pConfirm is a valid reservation confirmation and the current reservation request is successful for any metric in the metricIdList, those metrics are added to the list of metrics covered by this confirmation. If the operation is *DCI_OP_RELEASEDATA*, this must contain a valid confirmation from a previous *DCI_OP_RESERVEDATA* operation. If the operation is *DCI_OP_SETDATA* and any of the metrics in the list have been reserved, *pConfirm must contain the corresponding confirmation. If none were reserved, *pConfirm must be zero.

This reservation confirmation may only be used by the process that requested it.

bufferAddress Points to the address of a return status buffer.

bufferSize The size of the return status buffer.

dataAddress If operation is *DCI_OP_SETDATA* or *DCI_OP_RESERVEDATA*, this points to a buffer containing data to set.

dataSize Size of the dataAddress buffer.

timeout If the operation is *DCI_OP_RESERVEDATA*, *timeout specifies the maximum time to allow a reservation to remain valid. If the operation is

DCI_OP_SETDATA, it specifies the maximum amount of time to try to set the metric. This is ignored when the operation is *DCI_OP_RELEASEDATA*. If metrics are added to an existing reservation confirmation, the timeout on that confirmation is reset to the timeout value from the latest call that adds metrics.

DESCRIPTION

The *dciSetData*( ) routine is used by an consumer to request that a provider set a metric to a certain value. This may be used to set configuration values for a component being managed by a specific provider.

The *dciSetdata*( ) routine facilitates setting multiple metrics in the same call and use by multiple consumers by supporting a two phase set operation using two calls to *dciSetData*( ). The first call, with the operation set to *DCI_OP_RESERVEDATA*, is made to see if a set operation on a given component could occur. This asks the DCI Server to validate that setting the indicated metrics could occur with the data provided and to reserve any resources that would be required if the set command were issued. For each metric specified in this call, if the specified settings could have been made, resources are reserved, and *pConfirm is set to a special value. This special value can be used on a second call in order to validate a *DCI_OP_SETDATA* or *DCI_OP_RELEASEDATA* operation on the metrics that were just reserved. A metric that has been reserved in this manner can not be set, reserved, or released without using the respective reservation confirmation until that metric is released (using the reservation confirmation), the reservation confirmation times out, or the respective handle is closed. Also, a *DCI_OP_SETDATA* operation on reserved data is not guaranteed to succeed if the data is different from what was used on the previous call with *DCI_OP_RESERVEDATA*.

If the *DCI_OP_RESERVEDATA* operation is not used and multiple consumers try to set a metric, the result is indeterminate. Also, if multiple metrics are specified in a single metricIdList, some may be successfully set and others may not be. Using *DCI_OP_RESERVEDATA* to validate arguments and allocate resources for all metrics in a single call, before actually using *DCI_OP_SETDATA*, minimises the chances that when *dciSetData*( ) returns, some metrics will have been set while others will not have been set.

Resources reserved with *DCI_OP_RELEASEDATA* are held until a *DCI_OP_SETDATA* or *DCI_OP_RELEASEDATA* operation completes or until the timeout specified in the *DCI_OP_RESERVEDATA* call is exceeded. In order minimise chances for permanent deadlock, there is also an implementation specific maximum value for the reservation timeout. If a consumer makes a reservation and decides it no longer wants to set the metric, the consumer should make a *dciSetData*( ) call using the *DCI_OP_RELEASEDATA* operation rather than waiting for the reservation to timeout.

If *dciSetData*( ) is used to modify data for a class which supports invalid data (the DCIClassAttr flag has DCI_POSSIBLEINVALIDDATA set), the provider may reject the request with a return status of DCI_NOTSETTABLE.

If the return buffer address, *bufferAddress*, is zero when *dciSetData*( ) is called or if the return data buffer address, *dataAddress*, is zero when *dciSetData*( ) is called, then *dciSetData*( ) allocates a buffer of the correct size and places the address in *bufferAddress* or *dataAddress* respectively. If *dataAddress* is allocated by the library, then the size is also stored into the address provided for *dataSize*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciSetData*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciSetData*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_BADHANDLE]	The handle provided is not currently open.
[DCI_BADCONFIRM]	The reservation confirmation is either invalid or has expired.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, *MetricIdList* was malformed, or the *operation* was not recognised.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOTPOLLEDMETRIC]	A polled metric was required and the requested metric identifier was not for such a metric type.

[DCI_NOACCESS]	The calling process does not have permission to retrieve information about the requested metric or to initialise a connection to the DCI server.
[DCI_NOTSETTABLE]	One or more of the associated metrics is not able to be set. This may be because the provider does not support modification of these metrics, or that an "invalid" metric was specified. The provider may choose to disallow modification of "invalid" metrics for the associated instance.
[DCI_NOTRESERVEABLE]	This metric does not support being reserved.
[DCI_RESERVED]	This metric is currently reserved by another consumer.
[DCI_NOTRESERVED]	This metric is not currently reserved so cannot be released.
[DCI_INVALIDDATA]	The associated metric is invalid for this particular class instance.
[DCI_DERIVEDDATA]	An attempt was made to set the value of a metric of type: DCI_DERIVED.
[DCI_METHODERROR]	The DCI Server has encountered an error in the method invoked to satisfy the request for the selected metric.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

NAME

dciTerminate - terminate a connection with the Data Capture Interface

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciTerminate(void);
```

ARGUMENTS

"none"

DESCRIPTION

This interface is used by both metrics providers and consumers. The *dciTerminate*( ) routine allows DCI implementations to know when an application is ending a series of DCI transactions. This interface allows a DCI implementation to perform implementation specific termination. Conversely, *dciInitialize*( ) is used to perform implementation specific initialisation.

RETURN VALUES

The *dciTerminate*( ) routine returns [DCI_SUCCESS] if it was able to terminate the connection to the metric provider or consumer. Otherwise, this routine returns one of the following error values:

[DCI_NOTINITIALIZED]	The DCI subsystem is currently not initialised.
[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.

Chapter 6

Metrics Provider Routines

This Chapter describes the interfaces used by metrics providers.

NAME

dciAddInstance - add an instance to a metric class

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciAddInstance(
    DCIClassId       *classId,              /* in */
    DCIInstanceId    *instanceId,           /* in */
    DCIInstAttr      *instAttr,             /* in */
    DCIMethod        *method,               /* in */
    DCIReturn        **bufferAddress,       /* in/out */
    UMAUint4         bufferSize             /* in */
);
```

ARGUMENTS

classId	Address of a single classId to add instance to. This may not contain any wildcards.
instanceId	Address of an instance identifier to be added to the class. This may not contain any wildcards.
instAttr	Address of an instance attribute corresponding to the instance identifier being added. If this is zero, an attribute with a zero length label and zero sized extended attributes is created for the instance.
method	Address of an instance method corresponding to the instance identifier being added. If this is zero, no instance method is added and there must be a class method already registered for the given class.
bufferAddress	Points to the address of a return value buffer. If the address pointed to is zero, the system allocates a buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciAddInstance*( ) routine adds an instance to a class. This routine takes as input an instance identifier (*instanceId*), and optionally, an attribute structure (*instAttr*) or method structure (*method*) that correspond with the instance identifiers.

The instance attribute specified in *instanceAttr* can be retrieved by a consumer using *dciGetInstAttributes*( ). The *method* allows a provider to specify a method to be used instead of a class method registered with the class. (Instances with and without methods can be mixed in a class.)

The DCI_ADDRESS and DCI_CALLBACK instance method types are dependent on the address space of the provider who issues the *dciAddInstance*( ) being available to the DCI Server. If the address space of such a provider with instances that have not been removed becomes unavailable, such as when the provider terminates, any instances registered with *dciAddInstance*( ) is implicitly removed. On a related note, if the address space of a class which has a class method of type DCI_ADDRESS or DCI_CALLBACK becomes unavailable to the DCI Server, the entire class is implicitly unregistered.

Class instances for which the DCI_STORE instance method is in effect must submit initial data in the DCIReturn structure *bufferAddress*. The data in the structure must conform to the previously registered attributes structure. All datum identifiers of the metric identifiers in the DCIReturn structure should be wildcarded and the corresponding data section should contain

all class data of the specified class instance(s), with an offset as specified in the previously registered attributes structure. None of the other identifiers can be wildcarded.

If the return buffer address, *bufferAddress*, is zero when *dciAddInstance*( ) is called, then *dciAddInstance*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciAddInstance*( ) routine returns [DCI_SUCCESS] if all requested metric instances can be registered in the name space and the standard DCI return structure can be written into the output buffer. Otherwise, this routine returns one of the DCI error values to summarise the failure type.

A failure causes one of the following to be returned. These are summary errors. If the request consisted of multiple metrics then the return buffer status field should be examined to determine which metrics failed.

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for numIds, bufferSize is smaller than the size of a DCIReturn structure, MetricIdList was malformed, or the MethodList was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_INSTANCEEXISTS] This instance could not be added because it already exists.

[DCI_NOWILDCARD] A class or instance wildcard cannot be used in this context.

[DCI_METHODTYPEUNAVAILABLE]
The specified method *type* member is one of those documented in this specification, but which is not available on this platform. Only DCI_WAIT is guaranteed to be available on all implementations.

[DCI_METHOPOPNOTSUPPORTED]
The requested operation may not be specified in conjunction with this method type.

[DCI_INVALIDMETHODOP] The operation specified is not a valid operation.

[DCI_INVALIDFIELD] The associated DCIMethod attributes structure was malformed.

[DCI_NOTEXT] No label has been specified with DCIInstanceAttr field of the DCIMethod.

[DCI_INSTANCENOTPERSISTENT]
If the parent class is not persistent and the new DCIInstAttr specifies persistence.

[DCI_BADFLAGS] If persistence is defined for the DCIInstAttr, but the method type is DCI_ADDRESS or DCI_CALLBACK.

[DCI_NOCLASS] The specified metric class identifier is not present in the name space.

[DCI_NOACCESS] The caller does not have permission to add an instance to this class.

NAME

dciPostData - post polled data, instance data, or configuration data

SYNOPSIS

```
#include <sys/dci.h>
DCIStatus dciPostData(
    UMAUint4     operation,              /* in */
    UMAUint4     transactionId,          /* in */
    DCIReturn    *status,                /* in */
    void         *data,                  /* in */
    UMAUint4     dataSize,               /* in */
    DCIReturn    **bufferAddress,        /* in/out */
    UMAUint4     bufferSize              /* in */
);
```

ARGUMENTS

operation	The type of operation. Must be one of the values possible in the operation field of the DCIMethod struct.
transactionId	The identifier of the transaction associated with this data post.
status	Address of a DCIReturn structure listing the metric identifiers and data offsets.
data	Address of the data buffer.
dataSize	Size of the data buffer in bytes.
bufferAddress	Points to the address of a return value buffer. If the address pointed to is zero, the system allocates a buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciPostData*( ) routine is used by a provider for transmitting a list of polled data, instance data, configuration data or status values in response to a consumer request. The provider supplies the type of operation, which determines the type of data returned in the DCIReturn structure *status* and data buffer. The provider writes a DCIReturn structure to the *status* parameter which lists the metric identifiers and related data offsets. The data buffer may be filled with data corresponding to the operation. The *status* DCIReturn structure combined with the data buffer is used to fulfill a consumer's request.

If the operation is *DCI_OP_GETDATA* the supplied data must conform to the previously registered attributes structure. Although the DCI Server cannot verify the content of the data, it verifies that the size matches the registered attributes. Only datum identifier wildcards are allowed, in which case all instantiated class data is returned within a single data section.

If the operation is *DCI_OP_CONFIGURE* the supplied data must contain DCIConfig structures, one or more per metric identifier depending on the number of encoded instance identifiers.

If the operation is *DCI_OP_LISTINSTANCES* the supplied metric identifiers must have expanded instance identifiers. The data buffer is unused. The datum identifiers are ignored.

If the operation is *DCI_OP_GETINSTATTR* the supplied metric identifiers may have expanded instance identifiers and the supplied data must contain DCIInstAttr structures, one or more per metric identifier, identifying each instantiated metric class. The datum identifiers are ignored.

If the operation is *DCI_OP_SETDATA*, *DCI_OP_RESERVEDATA* or *DCI_OP_RELEASEDATA* only the status information is used; the data buffer is unused.

The *transactionId* is used to alert the DCI Server as to the ultimate DCI consumer for this data. For DCI_WAIT method classes, the *transactionId* was returned from the associated *dciWaitRequest*( ). For classes not using DCI_WAIT, the *transactionId* is not relevant and must be set to 0.

When the *dciPostData*( ) returns, the DCIReturn structure *bufferAddress* contains return status' for all submitted metric identifiers.

If the return buffer address, *bufferAddress*, is zero when *dciPostData*( ) is called, then *dciPostData*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciPostData*( ) routine returns [DCI_SUCCESS] if all requested metric instances can be submitted to the DCI Server. Otherwise, this routine returns one of the DCI error values to summarise the failure type.

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_ALLOCATIONFAILURE]	The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, *metricIdList* was malformed, or the *methodList* was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE]	There was at least one failure status.
[DCI_WARNING]	There was at least one warning status and no failure status.
[DCI_INFORMATIONAL]	There was at least one information status and no failure or warning status.
[DCI_SUCCESS]	All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS]	The request succeeded and there may be associated data.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
[DCI_NODATUMID]	The associated metricId specified a nonexistent datumId for the specified class.
[DCI_NOMETRIC]	There is no such metric identifier in the name space.
[DCI_NOSUCHTRANSACTION]	The *transactionId* was either invalid or the associated transaction was no longer active.
[DCI_NOWILDCARD]	A class or instance wildcard cannot be used in this context.

NAME

dciRegister - register a metric class

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciRegister(
    DCIClassId      *classId,               /* in */
    DCIClassAttr    *classAttr,             /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize              /* in */
);
```

ARGUMENTS

classId	Address of a DCIClassId structure.
classAttr	Address of a DCIClassAttr structure.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciRegister*( ) routine allows providers to create a new class in the name space. For the specified DCIClassId element in *classIdList*, there must be a corresponding DCIClassAttr structure which establishes the definition of the new class; information such as the number of instance levels and their types and well as the label and type of all datum metrics which are provided. The DCIClassId must be fully specified without the use of class wildcards.

If the return buffer address, *bufferAddress*, is zero when *dciRegister*( ) is called, then *dciRegister*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

Upon return *dciRegister*( ) fills the return buffer specified by bufferAddress with a DCIReturn structure. For the input DCIClassId, there is a corresponding DCIRetval structure produced in the return buffer. The *metricOffset* field of the DCIRetval specifies an offset from the buffer specified by *bufferAddress* and contains the DCIClassId associated with the *status* field of the DCIRetval structure. The *dataOffset* field is set to 0.

RETURN VALUES

The *dciRegister*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure could be written. Otherwise, *dciRegister*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer.

It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library could not allocate the memory for the return buffer. The application could attempt to allocate its own memory and try the request again.

[DCI_INVALIDARG] One of the input arguments is invalid: *numIds* was 0, *bufferSize* is smaller than the size of a DCIReturn structure, *classIdList* was malformed or *classAttrList* was malformed.

[DCI_INVALIDFIELD] One of the argument structure fields was not acceptable. This result will occur if a UMA_DERIVED DCIDataAttr is presented with a nonzero method *flags* argument.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS The request succeeded and there may be associated data.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space. This is only returned if any class other than the deepest specified class is not in the namespace.

[DCI_NOWILDCARD] A wildcard cannot be used in this context.

[DCI_INVALIDFIELD] The associated DCIClassAttr structure has an invalid field.

[DCI_CLASSEXISTS] This class could not be registered because it already exists.

[DCI_CLASSNOTPERSISTENT] If the parent class is not persistent, and the new DCIClassAttr specifies persistence.

[DCI_BADFLAGS] If persistence is defined for a DCIClassAttr, but the method type is DCI_ADDRESS or DCI_CALLBACK.

[DCI_NOACCESS] The caller does not have permission to register this class.

[DCI_NOTEXT] No label has been specified in the DCIClassAttr field of this class.

NAME

dciRemoveInstance - removes instances from a metric class

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciRemoveInstance(
    DCIMetricId     *metricId,              /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize              /* in */
);
```

ARGUMENTS

metricId	Address of a DCIMetricId structure.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciRemoveInstance*( ) routine removes the specified instances from the metrics name space. This routine takes as input an address of a DCIMetricId structure. The associated instances are removed from the name space. If so configured, this leads to a "final data" notification to all interested DCI consumers which have any removed instance open. The *datumId* field of the DCIMetricId is ignored. Upon return, *dciRemoveInstance*( ) produces DCIReturn structure at the address pointed to by *bufferAddress*. Each DCIRetval structure within the DCIReturn structure reflects the status of each instance removed. The *metricOffset* member of each DCIRetval specifies a location relative to *bufferAddress* and references the DCIMetricId structure. The *dataOffset* member is zero.

If the return buffer address, *bufferAddress*, is zero when *dciRemoveInstance*( ) is called, then *dciRemoveInstance*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciRemoveInstance*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure was written into the output buffer. Otherwise, *dciRemoveInstance*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error

(for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library was requested to provide memory for the specified buffer and could not. The application could attempt to allocate its own memory and try the request again.

[DCI_INVALIDARG] One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, or *metricIdList* was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_NOINSTANCE] The requested instance identifier is not in the name space.

[DCI_NOACCESS] The caller does not have permission to remove an instance from this class.

NAME

dciSetClassAccess - sets access control for metric classes

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciSetClassAccess(
    DCIClassId      *classIdList,           /* in */
    DCIAccess       *accessList,            /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize              /* in */
);
```

ARGUMENTS

classIdList	Address of a list of metric class identifiers.
accessList	A list of access control structures, one for each input metric class identifier.
numIds	The number of input metric class identifiers.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciSetClassAccess*( ) routine allows a provider to set the access control for a list of class identifiers. The access control information is initially set when classes are registered. This routine allows subsequent modification of the access control structure. There is no requirement that restricting access disables those consumers which have previously opened the metrics class but some secure implementations may enforce this behaviour.

If the return buffer address, *bufferAddress*, is zero then *dciSetClassAccess*( ) allocates the return buffer on behalf of the caller and return its address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory. Upon return *dciSetClassAccess*( ) writes the standard DCIReturn structure into the status buffer.

RETURN VALUES

The *dciSetClassAccess*( ) routine returns [DCI_SUCCESS] if all requested metric identifiers are in the name space, the caller has permission to set their access, and the standard DCI return structure can be written into the output buffer. Otherwise, this routine returns one of the DCI error values to summarise the failure type.

A failure causes one of the following to be returned. These are summary errors. If the request consisted of multiple classes then the return buffer status field should be examined to determine which classes failed.

[DCI_NOTINITIALIZED]	The DCI subsystem is currently not initialised.
[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_INVALIDARG]	One or more of the input arguments to the DCI routine were malformed.

[DCI_FAILURE]	There were multiple but different failure types for the requested metric identifiers.
[DCI_NOCLASS]	Any requested metric class identifier is not present in the name space.
[DCI_NOACCESS]	There was an access permission failure for at least one request.
[DCI_NOSPACE]	The size of the given return buffer is smaller than needed. A partial write of the results may have been performed. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_NOSUPPORT]	The implementation does not support this interface.

The possible status values written into the DCIReturn status field are:

[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOACCESS]	The caller does not have permission to modify access control.
[DCI_NOSPACE]	There was not enough room in the return buffer to write the expanded metric values for the requested instance identifier. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

NAME

dciSetInstAccess - sets access control for metric instances

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciSetInstAccess(
    DCIMetricId     *metricIdList,          /* in */
    DCIAccess       *accessList,            /* in */
    UMAUint4        numIds,                 /* in */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

metricIdList	Address of a list of metric class and instance identifiers.
accessList	A list of access control structures, one for each input instance identifier.
numIds	The number of input instance identifiers.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciSetInstAccess*( ) blocks indefinitely.

DESCRIPTION

The *dciSetInstAccess*( ) routine allows a provider to set the access control for a list of instance identifiers. The access control information is initially set when instances are added to the name space. This routine allows subsequent modification of the access control structure. There is no requirement that restricting access disables those consumers which have previously opened an instance but some secure implementations may enforce this behaviour.

The access control value of intermediate instance levels in a multiple level instance structure can be set by limiting the input instance value to the desired level. For example, if one wanted to set the access control field on the first instance level in a two level structure then only use the first level as the instance argument in *metricIdList*.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciSetInstAccess*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero then *dciSetInstAccess*( ) allocates the return buffer on behalf of the caller and return its address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory. Upon return *dciSetInstAccess*( ) writes the standard DCIReturn structure into the status buffer.

RETURN VALUES

The *dciSetInstAccess*( ) routine returns [DCI_SUCCESS] if all requested metric identifiers are in the name space, the caller has permission to set their access, and the standard DCI return

structure can be written into the output buffer. Otherwise, this routine returns one of the DCI error values to summarise the failure type.

A failure causes one of the following to be returned. These are summary errors. If the request consisted of multiple classes then the return buffer status field should be examined to determine which classes failed.

[DCI_NOTINITIALIZED]	The DCI subsystem is is currently not initialised.
[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_INVALIDARG]	One of the input arguments is invalid.
[DCI_FAILURE]	There were multiple but different failure types for the requested metric identifiers.
[DCI_NOCLASS]	Any requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	Any requested instance identifier is not present in the name space.
[DCI_NOACCESS]	There was an access permission failure for at least one request.
[DCI_NOSPACE]	The size of the given return buffer is smaller than needed. A partial write of the results may have been performed. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_NOSUPPORT]	The implementation does not support this interface.
[DCI_INTERRUPTED]	This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The possible status values written into the DCIReturn status field are:

[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	Any requested instance identifier is not present in the name space.
[DCI_NOACCESS]	The caller does not have permission to modify access control.

[DCI_NOSPACE]	There was not enough room in the return buffer to write the expanded metric values for the requested instance identifier. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

NAME

dciUnregister - unregister a metric class.

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciUnregister(
    DCIClassId    *classId,              /* in */
    DCIReturn     **bufferAddress,       /* in/out */
    UMAUint4      bufferSize             /* in */
);
```

ARGUMENTS

classId	Address of a DCIClassId structure.
bufferAddress	Points to the address of a return value buffer.
bufferSize	The size of the return buffer.

DESCRIPTION

The *dciUnregister*( ) routine allows providers to remove one of their registered classes from the namespace. This routine takes as input the address of a DCIClassId structure. The DCIClassId may include the class identifier wildcard value, DCI_ALL. For each DCIClassId in the array, the class is unregistered and removed from the namespace.

If the return buffer address, *bufferAddress*, is zero when *dciUnregister*( ) is called, then *dciUnregister*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciUnregister*( ) routine returns [DCI_SUCCESS] if the DCIReturn structure could be written. Otherwise, *dciUnregister*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined

to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library was requested to provide memory for the specified buffer and could not. The application could attempt to allocate its own memory and try the request again.

[DCI_INVALIDARG] One of the input arguments is invalid: *numIds* was 0, *bufferSize* is smaller than the size of a DCIReturn structure, or *classIdList* was malformed.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_CLASSNOTEMPTY] The class could not be removed because either there are subclasses still registered, or instances still defined for this class.

[DCI_NOACCESS] The caller does not have permission to unregister this class.

NAME

dciWaitRequest - wait for service request

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciWaitRequest(
    DCIMetricId     *metricIdList,          /* in */
    UMAUint4        numIds,                 /* in */
    UMAUint4        *operation,             /* out */
    UMAUint4        *transactionId,         /* out */
    DCIReturn       **bufferAddress,        /* in/out */
    UMAUint4        bufferSize,             /* in */
    UMATimeVal      *timeout                /* in */
);
```

ARGUMENTS

metricIdList	Address of a list of DCIMetricId structures.
numIds	The number of DCIMetricId structures in metricIdList.
operation	The type of operation. Must be one of the values possible in the operation field of the DCIMethod struct.
transactionId	Address of a memory location into which the associated *transactionId* is stored on a successful return from *dciWaitRequest*( ).
bufferAddress	Points to the address of a return value buffer. If the address pointed to is zero, the system allocates a buffer.
bufferSize	The size of the return buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciWaitRequest*( ) blocks indefinitely.

DESCRIPTION

The *dciWaitRequest*( ) routine allows a metrics provider to synchronously wait for a metrics service request. The *dciWaitRequest*( ) returns when a consumer requests the provider's metrics using *dciGetData*( ), alters metrics using *dciSetData*( ), lists instances with *dciListInstanceId*( ), requests instance attributes with *dciGetInstAttributes*( ), or configures metrics using *dciConfigure*( ).

The *dciWaitRequest*( ) announces the metrics it can serve by providing the *metricIdList*. The metric identifiers in this list should correspond to the way the provider registered itself. If the provider registered DCI_WAIT as a class method, it can use a wildcard at the instance level, otherwise it can only use wildcards at the datum level

When *dciWaitRequest*( ) returns, the *operation* contains the operation requested by the DCI Server, and the *bufferAddress* contains a DCIReturn structure with the requested *metricIds*. The *transactionId* returned is to be used as an input argument for all associated *dciPostData*( ) responses to ensure that the data sent will be properly directed to a waiting consumer. If data is supplied it is encoded in the DCIReturn structure with data offsets from the beginning of the DCIReturn structure.

If the operation is *DCI_OP_GETDATA* or *DCI_OP_RELEASEDATA*, only metric identifiers are supplied in the DCIReturn structure. Datum wildcarding may be used. If the class is not *DCI_PROVIDER_INSTANCE*, instance identifier can not be wildcarded.

If the operation is *DCI_OP_LISTINSTANCES*, only metric identifiers are supplied in the DCIReturn structure, of which the datum identifier should be ignored. The instance identifiers can be wildcarded.

If the operation is *DCI_OP_GETINSTATTR*, only metric identifiers are supplied in the DCIReturn structure, of which the datum identifier should be ignored. The instance identifiers can be wildcarded.

If the operation is *DCI_OP_CONFIGURE*, data is supplied in the DCIReturn structure in addition to the metric identifiers. Each data section contains one or more DCIConfig structures, depending on the number of instance identifiers encoded in the corresponding metric identifier. The instance identifier can be zero length which indicates "all current and future instances". Only datum identifier wildcards are allowed, unless the class is a DCI_PROVIDER_INSTANCES class in which case instance identifier wildcards are allowed.

If the operation is *DCI_OP_SETDATA* or *DCI_OP_RESERVEDATA*, data is supplied in the DCIReturn structure in addition to the metric identifiers. This data conforms to the previously registered attributes structure. Only datum identifier wildcards are allowed, unless the class is a DCI_PROVIDER_INSTANCES class in which case instance identifier wildcards are allowed.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciWaitRequest*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

If the return buffer address, *bufferAddress*, is zero when *dciWaitRequest*( ) is called, then *dciWaitRequest*( ) allocates the return buffer on behalf of the caller and returns the buffer address in *bufferAddress*. The caller is then responsible for subsequently freeing the allocated memory using *dciFree*( ).

RETURN VALUES

The *dciWaitRequest*( ) routine returns [DCI_SUCCESS] if the DCI Server successfully placed a request. Otherwise, this routine returns one of the DCI error values to summarise the failure type.

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_NOSPACE]	The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation

defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library was requested to provide memory for the specified buffer and could not. The application could attempt to allocate its own memory and try the request again.

[DCI_INVALIDARG] One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, *metricIdList* was malformed, or the *methodList* was malformed.

[DCI_INTERRUPTED] This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOACCESS] The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_NOINSTANCE] The requested instance identifier is not in the name space.

[DCI_NODATUMID] The associated metricId specified a nonexistent datumId for the specified class.

[DCI_NOMETRIC] There is no such metric identifier in the name space.

[DCI_NOWILDCARD] A class or instance wildcard cannot be used in this context.

[DCI_TIMEOUT] The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

Chapter 7

Event Routines

This Chapter describes the interfaces which handle events.

NAME

dciPostEvent - provider routine to post an event

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciPostEvent(
    DCIMetricId      *metricId,           /* in */
    UMAUint4         eventDataSize,       /* in */
    UMAUint4         eventDataCount,      /* in */
    UMAVarLenData    *eventData           /* in */
);
```

ARGUMENTS

metricId	Address of an event metric identifier for the event to be posted.
eventDataSize	Size of the event data in bytes. This must agree with the total of all size fields in the DCIEventDataAttr structures from the DCIEventAttr structure for this event registered in the DCIClassAttr structure.
eventDataCount	Number of entries in the eventData variable length array. Note that the array contains variable length entries; the size is passed by the provider to the server to enable the server to place it directly in the DCIEvent (see Figure 3-6 on page 50) structure it creates for this event.)
eventData	The data that should be passed on in the eventData field of the DCIEvent structure for this event.

DESCRIPTION

The *dciPostEvent*( ) routine is used by a provider for event notification. The inputs to this routine are the metric identifier for the event and the associated event data. If there is no associated event data, meaning that the only information of interest for this event is simply that the event occurred, then eventDataSize must be zero. Otherwise the data given by the event pointer must conform to the event attributes structure this provider previously registered for this event. Although the DCI Server cannot verify the content of the event data it does verify that the size matches the registered attributes. Once the input arguments have been verified, then the DCI Server packages it into a DCIEvent structure along with a timestamp and forward the event to the consumer.

RETURN VALUES

The *dciPostEvent*( ) routine returns [DCI_SUCCESS] if the event was posted successfully. Otherwise, *dciPostEvent*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT]	The DCI service is not available.
[DCI_NOIMPLEMENTATION]	In a DCI subset implementation, the specified routine has not been implemented.
[DCI_NOTINITIALIZED]	The DCI subsystem is not currently initialised.
[DCI_SYSERROR]	An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.
[DCI_INVALIDARG]	One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIReturn structure, *metricIdList* was malformed, *timeout* was malformed or the address of *dataSize* was not provided and

	dataAddress was specified and set to a NULL.
[DCI_NOCLASS]	The requested metric class identifier is not present in the name space.
[DCI_NOINSTANCE]	The requested instance identifier is not in the name space.
DCI_NODATUMID]	The [associated metricId specified a nonexistent datumId for the specified class.
[DCI_NOMETRIC]	There is no such metric identifier in the name space.
[DCI_NOTEVENTMETRIC]	The associated metricId specified a datumId which is not an event metric.
[DCI_NOACCESS]	The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.

NAME

dciWaitEvent - wait for one or more events

SYNOPSIS

```
#include <sys/dci.h>

DCIStatus dciWaitEvent(
    DCIHandle          handle,              /* in */
    DCIMetricId        *metricIdList,       /* in */
    UMAUint4           numIds,              /* in */
    DCIEventReturn     **bufferAddress,     /* in/out */
    UMAUint4           bufferSize,          /* in */
    void               **dataAddress,       /* in/out */
    UMAUint4           *dataSize,           /* in/out */
    UMATimeval         *timeout,            /* in */
    UMAUint4           eventFlags           /* in */
);
```

ARGUMENTS

handle	A handle opened on events to be waited for. This must contain at least one valid event.
metricIdList	Address of a list of metric identifiers, each of which must be an open event in the handle argument. These events specify a subset of the handle on which to base this function call. If metricIdList is zero, then the function call is based on all event metrics in the handle.
numIds	The number of input metric identifiers.
bufferAddress	Points to the address of a return status buffer.
bufferSize	The size of the return status buffer.
dataAddress	Points to an address of a buffer to use to store event data. If dataAddress is NULL, then the system allocates a buffer, setting the value of dataSize to the size of the allocated buffer.
dataSize	Address of the size of the return data buffer.
timeout	Pointer to a UMATimeVal structure that specifies the maximum time to wait for this request to complete. When *timeout* is NULL, *dciWaitEvent*( ) blocks indefinitely.
eventFlags	Bitmapped flags as described below.

DESCRIPTION

The *dciWaitEvent*( ) routine waits for any event in both the *handle* and the *metricIdList* to occur. If this *metricIdList* subset of the handle specifies a wildcard that when expanded would contain any polled metrics, that is, non events, they are ignored without producing any errors. An error results if any individual, that is, non-wildcarded, polled metric is specified by the *handle* and *metricIdList* combination.

dciWaitEvent( ) always separates the event data it is returning and the status return buffers in order to allow consumers in an event data acquisition loop to archive successfully acquired data and discard the status structure. *dciWaitEvent*( ) uses *bufferAddress* to write summary information about all events that have occurred and uses *dataAddress* to write the associated event data. If the return status buffer address is zero, then *dciWaitEvent*( ) allocates the return status on behalf of the caller and return its address in *bufferAddress*. If the return data buffer

address is zero, then *dciWaitEvent*( ) allocates a data buffer of size **dataSize* on behalf of the caller and return its address in *dataAddress*. The caller is responsible for subsequently freeing the allocated memory using *dciFree*( ).

The following flags may be specified in *eventFlags*:

DCI_EVENT_NOBLOCK
Do not block. Return immediately. This flag is mutually exclusive with the *DCI_EVENT_FILLBUFFER* flag.

DCI_EVENT_FILLBUFFER
This call should block until the buffer specified by *dataAddress* or the system allocated buffer if *dataAddress* was zero, is full. Any event that could not be fully written to *dataAddress* without overflowing the *dataAddress* buffer is buffered (or lost) using the policy for the respective handle and is not be reported in the return structure for this call. This flag is mutually exclusive with the *DCI_EVENT_NOBLOCK* flag.

This routine can be used with the class identifier wildcard value, DCI_ALL, and instance level wildcarding. Any attribute structure changes to the subsetted name space since the handle was created, or the addition of classes and instances to wildcarded collections are indicated by an appropriate status return value. The *timeout* parameter allows consumers to set a maximum event wait period.

If event buffering was not specified when the handle was opened, it is highly likely that events that a consumer is interested in can be lost. If an event occurs when no consumer is blocked in *dciWaitEvent*( ) and buffering of events has not been enabled, then the event may be lost. Lost events are reported in the DCIEventReturn struct of the next *dciWaitEvent*( ) call using that handle. If buffering is enabled and the buffer is not full, the event is saved in the per handle buffer until the next *dciWaitEvent*( ) call on that handle. If the buffer space is full when the event occurs, either one or more previous events are discarded, or the current event is discarded depending on the buffering policy set in *dciOpen*( ). If a consumer is using the *metricIdList* to specify only a subset of the events in the handle, then the behaviour of the system event buffering mechanisms is implementation dependent.

With no flags set, the default behaviour of *dciWaitEvent*( ) is to return after collecting only the events that are currently buffered or that occur during the call, or if none, block until at least one event has occurred. This behaviour can be modified by *DCI_EVENT_NOBLOCK* to allow it to return without any events occurring or by *DCI_EVENT_FILLBUFFER* to allow it to continue to wait for further events. Generation of status information of informational severity alone does not cause *dciWaitEvent*( ) to return prematurely.

If event data is returned from a class which could contain invalid data (the associated DCIClassAttr attribute flag has DCI_POSSIBLEINVALIDDATA set), and a the entire class of data is returned, the application is required to check each metric for validity before extracting its associated data.

The *timeout* parameter points to a type UMATimeVal structure that specifies the maximum time to wait for the completion of the *dciWaitEvent*( ) call. If the *timeout* has expired before the call completes, then one or more of the DCIRetval structures associated with the expanded metrics will show a DCI_TIMEOUT status. If the *timeout* parameter is NULL, then this call is not subject to a timeout. This call can be interrupted by a delivered signal; in this case, the DCIStatus returned for the call is DCI_INTERRUPTED and it is implementation defined whether any partial results are delivered.

RETURN VALUES

The *dciWaitEvent*( ) routine returns [DCI_SUCCESS] if the following DCIEventReturn structure was written into the return status buffer, *bufferAddress*.

```
typedef struct DCIEventReturn {
    UMAUint4        size;
    UMAUint4        numEvents;
    UMAUint4        numLostEvents;
    UMAVarLenDescr  eventStatus;
    UMAUint4        buffer_offset;
    UMAVarLenData   data;
} DCIEventReturn;
```

where the structure elements are as follows:

size Total number of bytes being returned. Must be a multiple of 4.

numEvents Number of events being reported.

numLostEvents Number of events in this handle lost or unrecorded for any reason since the last time a DCIEventReturn structure was returned for this handle.

eventStatus Descriptor for the variable sized data comprising the event status information. The event status information is a sequence of numEvents and consequent DCIStatus values, one for each event recorded in the dataAddress buffer, and in the same order.

buffer_offset This contains the offset from the beginning of the buffer to the event data.

data This is a place holder that conceptually contains all the data pointed to by the *eventStatus.offset* fields.

On return, *dataAddress* contains numEvents and consequent DCIEvent structures, one for each event that occurred.

If *DCI_EVENT_NOBLOCK* was set and no events were returned, the numEvents field of the DCIEventReturn is zero and the data area contains no DCIStatus. The numLostEvents field may be non zero in this case if events were lost since the last *dciWaitEvent*( ) call using this handle.

Note that [DCI_SUCCESS] could be returned and no events collected if the handle contained no events or if all events in the handle were disabled.

If [DCI_SUCCESS] was not returned, *dciWaitEvent*( ) returns one of the following fatal errors:

[DCI_NOTPRESENT] The DCI service is not available.

[DCI_NOIMPLEMENTATION] In a DCI subset implementation, the specified routine has not been implemented.

[DCI_NOTINITIALIZED] The DCI subsystem is not currently initialised.

[DCI_SYSERROR] An internal error has occurred (such as a shortage of resources) that may be beyond the control of the application. A vendor-specific error code is placed in the variable *errno*.

[DCI_NOSPACE] The provided buffer is too small for the return structure. The *size* field of the DCIReturn structure indicates the buffer size which would have held all the associated return values. If the *count* field of the DCIReturn structure is nonzero, then partial data was written to the buffer.

It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.

[DCI_ALLOCATIONFAILURE] The DCI library could not allocate the memory for the return buffer.The application could attempt to allocate its own memory and try the request again.

[DCI_BADHANDLE] The handle provided is not currently open.

[DCI_INVALIDARG] One of the input arguments is invalid: a negative value was used for *numIds*, *bufferSize* is smaller than the size of a DCIEventReturn structure, or *metricIdList* was malformed.

[DCI_BADFLAGS] One or more mutually exclusive flags were used together.

[DCI_TIMEOUT] A timeout occurred during data collection. Some events may have been reported.

[DCI_INTERRUPTED] This call was interrupted by a signal and did not complete. It is implementation defined whether partial results are provided. If partial results are provided, the application may need to amend the request list to avoid duplicating completed requests.

The summary status of all individual DCIRetval structure *status* members is stored in the DCIReturn structure *status* member. This summary status represents the highest severity of status returned among all DCIRetval structures.

[DCI_FAILURE] There was at least one failure status.

[DCI_WARNING] There was at least one warning status and no failure status.

[DCI_INFORMATIONAL] There was at least one information status and no failure or warning status.

[DCI_SUCCESS] All status returned was successful.

For each DCIRetval structure returned, the *status* member may contain the following:

[DCI_SUCCESS] The request succeeded and there may be associated data.

[DCI_NOCLASS] The requested metric class identifier is not present in the name space.

[DCI_NOINSTANCE] There is no such instance identifier in the handle.

[DCI_NOMETRIC] There is no such metric identifier in the name space.

[DCI_NOTEVENTMETRIC] The associated metricId specified a datumId which is not an event metric.

[DCI_NOACCESS] The caller does not have permission to find out if a requested instance identifier exists or does not have access to a metric identifier.

[DCI_NOSPACE]	There was not enough room in the return buffer to write the event data for the fired event. It is implementation defined whether or not partial data is available in the buffer in the case of a DCI_NOSPACE error (for example, on a *dciGetData*( ) call). It is also implementation defined whether or not the state of the DCI changes given that a DCI_NOSPACE error has occurred (for example, on a wildcarded *dciRemoveInstances*( ) call). In each of the above cases, individual DCIRetval status values must be examined to determine whether or not the data is valid, and whether or not the requested change actually occurred.
[DCI_CLASSESADDED]	This new class has been added within the scope of a wildcarded class request.
[DCI_INSTANCESADDED]	This new instance has been added within the scope of a wildcarded instance request.
[DCI_NOTENABLED]	The requested metric is currently not enabled by its provider.
[DCI_TIMEOUT]	The associated metric could not be expanded or referenced during the specified timeout period. This may be because the affiliated provider could not be contacted, or because the reference was never attempted due to an existing timeout condition in the input request list.

Appendix A

C Language Header Files

The include files **<dci.h>** and **<uma.h>** are used with the C language to define the DCI data types and structures. They also include the function prototypes for all exported DCI interfaces.

A.1 <uma.h>

The **<uma.h>** header file is given in the MLI Specification, which is in **Part 2 Appendix A** of this publication.

A.2 <dci.h>

The **<dci.h>** header file is given below.

```
#ifndef _SYS_DCI_H_
#define _SYS_DCI_H_

/* dci.h - exported interfaces and structures for the Data Capture Interface
 *
 * Description:
 *
 * ****************** NOTICE *********************************
 *  The <dci.h>, <mli.h> and <uma.h> header files
 *  introduce UMA symbols which may conflict with other
 *  symbols defined in an application.  Symbols with the
 *  following prefixes are therefore reserved to UMA:
 *      DCI
 *      dci
 *      UMA
 *      UMR
 *      UMS
 *
 *   Note that the header files are provided as advisory
 *   reference examples.
 * ****************** END OF NOTICE **************************
 */

#include <sys/uma.h>

#define DCI_ALL           0xffffffff    /* Wildcard value classes and datums */
#define DCI_ALL_INSTANCES 0x00000000    /* Wildcard value for instances */


/* The return status values for DCI API calls */

/* the summary values are bitmasks used to determine severity of result */
#define DCI_FAILURE             0x80000000
#define DCI_WARNING             0x40000000
#define DCI_INFORMATIONAL       0x20000000
#define DCI_SUCCESS             0x10000000
```

```
#define DCI_FATAL               0x08000000

/* fatal errors returned from the dci routine itself */
#define DCI_INITIALIZED         (DCI_FATAL | 0x01)
#define DCI_NOTINITIALIZED      (DCI_FATAL | 0x02)
#define DCI_NOTPRESENT          (DCI_FATAL | 0x03)
#define DCI_SYSERROR            (DCI_FATAL | 0x04)
#define DCI_INVALIDARG          (DCI_FATAL | 0x05)
#define DCI_NOSPACE             (DCI_FATAL | 0x06)
#define DCI_INTERRUPTED         (DCI_FATAL | 0x07)
#define DCI_BADHANDLE           (DCI_FATAL | 0x08)
#define DCI_ALLOCATIONFAILURE   (DCI_FATAL | 0x09)
#define DCI_BADFLAGS            (DCI_FATAL | 0x0a)
#define DCI_NOIMPLEMENTATION    (DCI_FATAL | 0x0b)

/* errors returned as individual or summary status */
#define DCI_NOTEVENTMETRIC        (DCI_FAILURE | 0x01)
#define DCI_NOCLASS               (DCI_FAILURE | 0x02)
#define DCI_NOINSTANCE            (DCI_FAILURE | 0x03)
#define DCI_NOMETRIC              (DCI_FAILURE | 0x04)
#define DCI_NOTEXT                (DCI_FAILURE | 0x06)
#define DCI_NOWILDCARD            (DCI_FAILURE | 0x07)
#define DCI_CLASSEXISTS           (DCI_FAILURE | 0x08)
#define DCI_INSTANCEEXISTS        (DCI_FAILURE | 0x09)
#define DCI_NODATUMID             (DCI_FAILURE | 0x0a)
#define DCI_METHODTYPEUNAVAILABLE (DCI_FAILURE | 0x0b)
#define DCI_NOTPOLLEDMETRIC       (DCI_FAILURE | 0x0c)
#define DCI_BADCONFIRM            (DCI_FAILURE | 0x0d)
#define DCI_NOTSETTABLE           (DCI_FAILURE | 0x0e)
#define DCI_NOTRESERVABLE         (DCI_FAILURE | 0x0f)
#define DCI_RESERVED              (DCI_FAILURE | 0x10)
#define DCI_NOTRESERVED           (DCI_FAILURE | 0x11)
#define DCI_NOTQUERYABLE          (DCI_FAILURE | 0x12)
#define DCI_METHODOPNOTSUPPORTED  (DCI_FAILURE| 0x13)
#define DCI_INVALIDMETHODOP       (DCI_FAILURE | 0x14)
#define DCI_NOTENABLED            (DCI_FAILURE | 0x15)
#define DCI_CLASSNOTPERSISTENT    (DCI_FAILURE | 0x16)
#define DCI_INSTANCENOTPERSISTENT (DCI_FAILURE | 0x17)
#define DCI_CLASSNOTEMPTY         (DCI_FAILURE | 0x18)
#define DCI_INVALIDDATA           (DCI_FAILURE | 0x19)
#define DCI_DERIVEDDATA           (DCI_FAILURE | 0x1a)
#define DCI_METHODERROR           (DCI_FAILURE | 0x1b)
#define DCI_SUBSETUNSUPPORTED     (DCI_FAILURE | 0x1c)
#define DCI_DCIMAJORUNSUPPORTED   (DCI_FAILURE | 0x1d)
#define DCI_DCIMINORUNSUPPORTED   (DCI_FAILURE | 0x1e)
#define DCI_INVALIDFIELD          (DCI_FAILURE | 0x1f)
#define DCI_TIMEOUT               (DCI_FAILURE | 0x20)
#define DCI_NOACCESS              (DCI_FAILURE | 0x21)
#define DCI_EVENTSUPPORT          (DCI_FAILURE | 0x22)

/* informational individual or summary status */
#define DCI_CLASSADDED          (DCI_INFORMATIONAL | 0x01)
#define DCI_INSTANCEADDED       (DCI_INFORMATIONAL | 0x02)
#define DCI_NOSUCHTRANSACTION   (DCI_INFORMATIONAL | 0x03)
#define DCI_INVALIDDATAPRESENT  (DCI_INFORMATIONAL | 0x04)

typedef UMAUint4        DCIStatus;      /* returned by all DCI routines */
typedef UMAUint4        DCIHandle;      /* returned by dciOpen()        */
typedef UMAUint4        DCIDatumId;     /* the datumId                  */
```

```
/* macros to access the datumId byte by byte.  The address of
 * the DCIDatumId is presented, and the address of the specified
 * byte is produced.
*/
#define dcidatumidarg1(datumidp)  ((char *)((*datumidp>>24)&0xff))
#define dcidatumidarg2(datumidp)  ((char *)((*datumidp>>16)&0xff))
#define dcidatumidarg3(datumidp)  ((char *)((*datumidp>>8)&0xff))
#define dcidatumidself(datumidp)  ((char *)((*datumidp>>0)&0xff))

/* Reserved DCIDatumId values */
#define DCI_INVALIDDATUMID      0x000000ff
#define DCI_FINALDATA_EVENT     0x000000f8
#define DCI_INSTANCEADDED       0x000000f7
#define DCI_INSTANCEREMOVED     0x000000f6
#define DCI_DATACHANGED         0x000000f5

/*=================================================================
 * DCI Versioning structure
 */

/* bitmapped flags to indicate the level of implementation support.
 * These are used as part of the structure passed from dciInitialize
 * to indicate the specific level of support available in an
 * implementation.
 */
#define DCI_SUBSET_BASIC                0x01
#define DCI_SUBSET_MULTIPLE_PROVIDERS   0x02
#define DCI_SUBSET_ACCESS_CONTROL       0x04
#define DCI_SUBSET_EVENT_SUPPORT        0x08
#define DCI_SUBSET_SET_CAPABILITY       0x10

/* DCI version structure.
 * This is passed into the dciInitialize() call as a request structure.
 * As such, it specifies a request to connect to a specific DCI API
 * version. A DCI version structure is also passed as an output parameter
 * indicating the level of support that this particular DCI implementation
 * is actually making available.
 */
typedef struct DCIVersion {
    UMAUint4     DCIMajorVersion;
    UMAUint4     DCIMinorVersion;
    UMAUint4     DCISubsetMask;
    UMAUint4     DCIVendorExtensions;
} DCIVersion;

typedef DCIVersion DCIVersion_1;

/* the major and minor version numbers of this dci.h */
#define DCI_MAJORVERSION       1
#define DCI_MINORVERSION       0

/* Compile time constants that indicate which DCI subsets are implemented
 * Implementers should change the value of the appropriate constant
 * if the corresponding subset is present in an implementation.
 * If a subset is not implemented the constant should have value 0.
 * For illustrative purposes below we show the assignments assuming only
 * Basic support is implemented.
 */
#define _DCI_SUBSET_BASIC              DCI_SUBSET_BASIC
```

```
#define _DCI_SUBSET_MULTIPLE_PROVIDERS  0x00 /*DCI_SUBSET_MULTIPLE_PROVIDERS*/
#define _DCI_SUBSET_ACCESS_CONTROL      0x00 /*DCI_SUBSET_ACCESS_CONTROL     */
#define _DCI_SUBSET_EVENT_SUPPORT       0x00 /*DCI_SUBSET_EVENT_SUPPORT      */
#define _DCI_SUBSET_SET_CAPABILITY      0x00 /*DCI_SUBSET_SET_CAPABILITY     */
#define _DCI_SUBSET_MASK                _DCI_SUBSET_BASIC                 | \
                                        _DCI_SUBSET_MULTIPLE_PROVIDERS    | \
                                        _DCI_SUBSET_ACCESS_CONTROL        | \
                                        _DCI_SUBSET_EVENT_SUPPORT         | \
                                        _DCI_SUBSET_SET_CAPABILITY

/*====================================================================
 * DCI Namespace identifier structures
 */

/*
 * The DCIClassId structure.
 *
 * The class identifier is a variable length array of 4 byte
 * unsigned integers, where each integer is referred to as a
 * "level".  Use the following macros to access this structure:
 *     dciclassidlen   - returns the number of levels specified
 *     dciclassidlevel - returns the address of the integer
 *                       for that level of the class identifier.
 */
typedef struct DCIClassId {
    UMAUint4       size;        /* size of the fixed and variable */
                                /* length portions                */
    UMAVarLenData  data;        /* variable length data           */
                                /* (the identifier)               */
} DCIClassId;


/* The DCIInstanceId structure
 *     An instance can have multiple levels, where each level name
 *     is stored in a variable length data structure.
 *     The 'size' field represents the combined size of the
 *     complete structure (i.e., all level names).
 *     The 'inputMask' field is a bitmask and indicates which
 *     instance levels have entries in the variable length
 *     'data' structure.  If an instance level's inputMask
 *     bit is not set, the level is wildcarded.
 *     The 'outputMask' is used to reduce the size of the
 *     returned instance id from a dci call.  It indicates
 *     to the provider/DCI Server which levels should be
 *     included in the returned instance ids.
 */
typedef struct DCIInstanceId {
    UMAUint4        size;           /* size of the fixed and    */
                                    /* variable length portions */
    UMAUint4        inputMask;      /* input mask, 32 levels    */
    UMAUint4        outputMask;     /* output mask, 32 levels   */
    UMAVarLenData   data;           /* variable length data     */
                                    /* (the instance level      */
                                    /*  identifiers)            */
} DCIInstanceId;

/* Definitions for the instance header size, everything but the
 * instLevels, and the maximum number of levels.  The latter is
 * dictated by the number of bits in the inputMask field. A
```

```
 * value of 0x00000000 for inputMask is reserved for a
 * completely wildcarded instance (see DCI_ALL_INSTANCES).
 */
#define DCI_MAXINSTLEVELS        32
#define DCI_MAXCLASSLEVELS      256

typedef struct DCIInstLevel {
    UMADataType         type;       /* type of the instance   */
                                    /* level value            */
    UMAInstTagType      itype;      /* instance type          */
    UMAUint4            size;       /* size of instance level */
                                    /* value in bytes         */
} DCIInstLevel;

/* The DCIMetricId structure
 *     A metric identifier consists of a Class & Instance identifier
 *     Both of these are variable length records with a size as their
 *     first field.
 */
typedef struct DCIMetricId {
    UMAUint4        size;       /* size of whole structure       */
    UMAVarLenDescr  classId;    /* descriptor for the            */
                                /* variable length DCIClassId    */
    UMAVarLenDescr  instanceId; /* descriptor for the            */
                                /* variable length DCIInstanceId */
    DCIDatumId      datumId;    /* the datum identifier          */
    UMAVarLenData   data;       /* data of DCIClassId            */
                                /* and DCIInstanceId             */
} DCIMetricId;

/* The DCILabel structure
 *     A DCI label can be used to create a human
 *     readable name of a metric class, instance or datum.  A label
 *     has two parts, a part to support internationalised text and a
 *     default ascii label should the I18N mechanism fail.  The
 *     former might not be the I18N text itself but a system
 *     dependent structure used to generate text.  The ascii
 *     data is a null terminated character string padded to a four
 *     byte boundary. The i18n data is an octet string (which is
 *     not necessarily null-terminated).
 */


typedef struct DCILabel {
    UMAUint4        size;   /* size of entire structure            */
    UMAVarLenDescr  ascii;  /* descriptor for the variable         */
                            /* length DCITextString for ascii      */
    UMAElementDescr i18n;   /* descriptor for the variable         */
                            /* length data for I18N                */
    UMAVarLenData   data;   /* data of DCILabel for ascii and i18n */
} DCILabel;

/* Data Attribute Structure */
typedef struct DCIDataAttr {
    UMAUint4        size;      /* size of whole structure          */
    DCIDatumId      datumId;   /* datum identifier for this metric */
    UMADataType     type;      /* datum type                       */
    UMAUnit         units;     /* units of this statistic          */
    UMAUint4        flags;     /* method operations flags          */
```

```
    UMAUint4        offset;    /* offset of this data when a       */
                               /* whole class is returned          */
    UMAVarLenDescr  label;     /* descriptor for the variable      */
                               /* length DCILabel                  */
    UMAVarLenData   data;      /* the DCILabel                     */
} DCIDataAttr;

typedef struct DCIClassAttr {
    UMAUint4         size;       /* size of whole structure           */
    UMAUint4         flags;      /* special class state               */
    UMAVarLenDescr   access;     /* descriptor for DCIAccess          */
                                 /* structure                         */
    UMAVarLenDescr   method;     /* descriptor for optional           */
                                 /* DCIMethod structure               */
    UMAVarLenDescr   label;      /* label data                        */
    UMAArrayDescr    instLevel;  /* descriptor for DCIInstLevel array */
    UMAVarArrayDescr dataAttr;   /* descriptor for DCIDataAttr array  */
    UMAVarArrayDescr eventAttr;  /* descriptor for DCIEventAttr array */
    UMAElementDescr  extensions; /* extension data                    */
    UMAVarLenData    data;       /* dataattr, eventattr, label        */
                                 /* DCIMethod,instance level, ext,    */
                                 /* and DCIAccess                     */
} DCIClassAttr;

/* bitmapped flags for DCIClassAttr */
#define DCI_ENABLED             0x01  /* (!enabled = disabled)        */
#define DCI_NOTIMPLEMENTED      0x02  /* unimplemented class          */
#define DCI_NOTAPPLICABLE       0x04  /* not applicable to the system */
#define DCI_OBSOLETE            0x08  /* class is being phased out    */
#define DCI_PROVIDER_INSTANCE   0x10  /* provider                     */
#define DCI_PERSISTENT_CLASS    0x20  /* keep class on exit/exec      */
#define DCI_POSSIBLEINVALIDDATA 0x40  /* class may contain metrics    */
                                      /* that are invalid for certain */
                                      /* instances of the class       */

typedef struct DCIEventDataAttr {
    UMAUint4        size;       /* size of the whole structure   */
    UMADataType     type;       /* format of data returned       */
    UMAUnit         units;      /* units of data returned        */
    UMAUint4        offset;     /* offset of data in return area */
    UMAVarLenDescr  label;      /* descriptor for the variable   */
                                /* length DCILabel               */
    UMAVarLenData   data;       /* the DCILabel                  */
} DCIEventDataAttr;

typedef struct DCIEventAttr {
    UMAUint4        size;           /* size of whole structure          */
    DCIDatumId      datumId;        /* datum identifier for this metric */
    UMAVarLenDescr  label;          /* descriptor for the variable      */
                                    /* length DCILabel                  */
    UMAVarArrayDescr eventDataAttr; /* descriptor for the array         */
                                    /* of DCIEventDataAttrs             */
    UMAVarLenData   data;           /* the DCILabel and                 */
                                    /* DCIEventDataAttr array           */
} DCIEventAttr;

typedef struct DCIEvent {
    UMAUint4          eventHeader;    /* the event header      */
    UMAVarLenData     data;           /* data for the event    */
```

```
} DCIEvent;

/* Bitmapped flags used in the first 12 bits of the event header      */
/* The flags indicate which items are include in the posted event     */
/* If included, items appear in the same order as their corresponding */
/* bit in the bitmap (e.g., header first, classid next ...)           */
#define DCI_EVENTHDR                0x001  /*include event header          */
#define DCI_EVENTHDRCLASSID         0x002  /*include a class id            */
#define DCI_EVENTHDRINSTANCEID      0x004  /*include an instance id        */
#define DCI_EVENTHDRTIMESTAMP       0x008  /*include a 64-bit time stamp   */
#define DCI_EVENTHDRVENDORTIMESTAMP 0x010  /*include an implementation     */
                                           /*defined timestamp            */
#define DCI_EVENTHDRCOMPTIMESTAMP   0x020  /*include a compressed timestamp*/
#define DCI_EVENTHDRSTREAMID        0x040  /*include an event stream id    */
#define DCI_EVENTHDRDATA            0x080  /*include the event data        */

/* DCIEventRetval is returned upon a dciWaitEvent() call */
typedef struct DCIEventRetval {
    UMAUint4        size;           /* size of the whole structure */
    UMAUint4        numEvents;      /* number of events reported   */
    UMAUint4        numLostEvents;  /* number of events lost       */
                                    /* and unrecorded              */
    UMAVarLenDescr  eventStatus;    /* descriptor for array of     */
                                    /* DCIStatus values            */
    UMAUint4        bufferhead;     /* may contain the offset      */
                                    /* for oldest event            */
    UMAVarLenData   data;           /* variable length data        */
                                    /* (DCIStatus,implstatus)      */
} DCIEventRetval;

/*=============================================================================
 * The DCIReturn structure
 *     All DCI routines return the same type of structure.  The
 *     structure consists of size and count fields followed by
 *     a variable sized array of DCIRetval structures.  There
 *     is one array element for every input metric.  Each array
 *     element contains status values, and offsets to the
 *     input argument and returned data.

typedef struct DCIRetval {
    DCIStatus       status;        /* status for input argument      */
    UMAUint4        metricOffset;  /* offset to input id value       */
    UMAUint4        dataOffset;    /* offset to data value           */
    UMAUint4        dataSize;      /* size of data returned in bytes */
} DCIRetval;

typedef struct DCIReturn {
    UMAUint4        size;           /* total bytes in DCIReturn      */
    UMAUint4        count;          /* number of returned elements   */
    DCIStatus       sumstatus;      /* summary status                */
    UMATimeSpec     startTime;      /* Start time of operation       */
    UMATimeSpec     endTime;        /* End time of operation         */
    DCIRetval       retval[1];      /* status, input id,i and output */
} DCIReturn;


typedef struct DCIAccess {
    UMAUint4        size;   /* size of the whole DCIAccess structure */
    UMAVarLenData   access; /* byte array containing the access      */
```

```
                                /* information                          */
} DCIAccess;

/*
 * DCIInstAttr -
 *     variable length array holding the attributes of an instance.
 */
typedef struct DCIInstAttr {
    UMAUint4         size;      /* size of the whole structure           */
    UMAUint4         flags;     /* special instance state                */
    UMAVarLenDescr   access;    /* descriptor for the DCIAccess structure */
    UMAElementDescr  extension; /* descriptor for variable length extns   */
    UMAVarLenDescr   label;     /* descriptor for variable length DCILabel*/
    UMAVarLenData    data;      /* data for DCIAccess, DCILabel          */
                                /* and extensions                        */
} DCIInstAttr;

/* bitmapped flags for DCIInstAttr */
#define DCI_PERSISTENT_INSTANCE  0x01    /* keep instance on exit/exec */

/* structure to indicate invalid data in a class.
 * This structure is returned for dciGetData requests on the
 * DCI_INVALIDDATUMID metric in the class (if the class is enabled
 * for invalid data, as indicated with the DCI_POSSIBLEINVALIDDATA
 * flag).
 * The structure consists of a size indicating total size of the
 * structure, and an array of datumids.  Each entry in the ids[]
 * array indicates a datum that is invalid for this particular
 * instance of the class.
 */
struct DCIInvalidData {
    UMAUint4        size;
    DCIDatumId      ids[1]; /* variable length component */
}
#define dciinvaliddatacount(invdatap)     ((invdatap->size)/sizeof(DCIDatumId) )
#define dciinvaliddatumid(invdatap,index)  (invdatap[index+1])


/* The DCIMethod structure
 *     This is a variable sized structure that describes the method
 *     metric providers use to deliver metrics.  This structure is
 *     only used by the provider routines and is not exported to
 *     consumers.  The method includes the method type, a variable
 *     sized method field which is padded to a four byte boundary,
 *     and an instance attribute structure.
 */

enum DCIMethodType {
    DCI_WAIT        = 0,    /* provider will block in dciWaitRequest   */
    DCI_ADDRESS     = 1,    /* data retrieved from provider address    */
    DCI_CALLBACK    = 2,    /* a provider function call is issued      */
    DCI_STORE       = 3     /* data is periodically stored by provider */
};
typedef enum DCIMethodType DCIMethodType;

typedef struct DCIMethod{
    UMAUint4        size;     /* size of this structure     */
    DCIMethodType   type;     /* how data will be retrieved */
    UMAElementDescr method;   /* descriptor for variable    */
```

```
                                /* length method data         */
    UMAVarLenData   data;       /* data for method            */
} DCIMethod;

/* methods operations */
#define DCI_OP_GETDATA            0x01
#define DCI_OP_SETDATA            0x02
#define DCI_OP_RESERVEDATA        0x04
#define DCI_OP_RELEASEDATA        0x08
#define DCI_OP_LISTINSTANCES      0x10
#define DCI_OP_GETINSTATTR        0x20
#define DCI_OP_CONFIGURE          0x40


typedef struct DCIInstanceData {
    UMAUint4       size;           /* size of the whole structure         */
    UMAUint4       count;          /* number of elements in each array    */
    UMAVarLenDescr instanceIdList; /* descriptor for DCIInstanceId array  */
    UMAVarLenDescr instAttrList;   /* descriptor for DCIInstAttr array    */
    UMAVarLenData  data;           /* DCIInstanceId and DCIInstAttr arrays*/
} DCIInstanceData;

typedef struct DCIAddressMethodData {
    void *address;                 /* the base address of a memory region */
    UMAUint4 size;                 /* the size of the region in bytes     */
    void *sync;                    /* opaque memory synchronisation token */
} DCIAddressMethodData;

typedef struct DCIConfig {
    UMAUint4        size;          /* total structure size, in bytes    */
    UMAUint4        flags;         /* configuration request             */
    UMAElementDescr configdata;    /* descriptor for the auxiliary      */
                                   /* config info                       */
    UMAVarLenData   data;          /* auxiliary config data starts here */
} DCIConfig;

/* These flags must exist in the lower half of the word */
#define DCI_ENABLE        0x01  /* enable the metrics   */
#define DCI_DISABLE       0x02  /* disable the metrics  */
#define DCI_CONFIGURATION 0x04  /* configuration data is passed to provider */
#define DCI_EVENT_ENABLE               0x0008
#define DCI_EVENT_DISABLE              0x0010
#define DCI_BUFFER_EVENTS_DISCARD      0x0020
#define DCI_BUFFER_EVENTS_OVERWRITE    0x0040
#define DCI_BUFFER_EVENTS_GETSIZE      0x0080
#define DCI_BUFFER_EVENTS_SETSIZE      0x0100
#define DCI_BUFFER_EVENTS_POLICY       0x0200

#define DCI_QUERYABLE   0x10000   /* polled metric is gettable   */
#define DCI_SETTABLE    0x20000   /* polled metric is settable   */
#define DCI_RESERVABLE  0x40000   /* polled metric is reservable */

/* dciWaitEvent() eventflags */
#define DCI_EVENT_NOBLOCK    0x01 /* return with any pending events */
                                  /* immediately                    */
#define DCI_EVENT_FILLBUFFER 0x02 /* return when the fillbuffer is  */
                                  /*  almost full                   */

/* DCI function prototypes. The function prototypes are always enabled
```

```
 * except _NO_PROTO is #define'd.
 *
 */
#ifndef _NO_PROTO
/* Basic support */
DCIStatus dciInitialize(DCIVersion *request, DCIVersion *response);
DCIStatus dciRegister(DCIClassId *classId, DCIClassAttr *classAttr,
                DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciTerminate(void);
DCIStatus dciListClassId(DCIHandle handle,
    DCIClassId *classIdList,UMAUint4 numIds,
    DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciListInstanceId(DCIHandle handle,
    DCIMetricId *metricIdList,UMAUint4 numIds,
    DCIReturn **bufferAddress, UMAUint4 bufferSize,
    UMATimeVal *timeout);
DCIStatus dciOpen(DCIHandle *handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress, UMAUint4 bufferSize,
    UMAUint4 handleflags, UMATimeVal *timeout);
DCIStatus dciClose(DCIHandle handle);
DCIStatus dciGetClassAttributes(DCIHandle handle, DCIClassId *classIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciGetInstAttributes(DCIHandle handle,DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress, UMAUint4 bufferSize,
    UMATimeVal *timeout);
DCIStatus dciConfigure(DCIHandle handle, DCIMetricId *metricIdList,
    DCIConfig *configlist, UMAUint4 numIds, DCIReturn **bufferAddress,
    UMAUint4 bufferSize, UMATimeVal *timeout);
DCIStatus dciGetData(DCIHandle handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress, UMAUint4 bufferSize,
    void **dataAddress, UMAUint4 *datasize, UMATimeVal *timeout);
DCIStatus dciSetData(DCIHandle handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, UMAUint4 operation, UMAUint4 *pConfirm,
    DCIReturn **bufferAddress, UMAUint4 *bufferSize,
    void *dataAddress, UMAUint4 dataSize, UMATimeVal *timeout);
DCIStatus dciFree(void *ptr);
void      *dciAlloc(UMAUint4 size);
void      dciPerror(DCIStatus status, int theerrno, char *membuf,
                int bufsize, char *fmt, ...);
DCIStatus dciAddHandleMetric(DCIHandle handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress, UMAUint4 bufferSize,
    UMATimeVal *timeout);
DCIStatus dciRemoveHandleMetric(DCIHandle handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIReturn **bufferAddress,
    UMAUint4 bufferSize, UMATimeVal *timeout);

/* Multiple Providers */
DCIStatus dciAddInstance(DCIClassId *classId,
    DCIInstanceId *instanceId, DCIInstAttr *instAttr,
    DCIMethod *method,
    DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciRemoveInstance(DCIMetricId *metricId,
    DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciWaitRequest(DCIMetricId *metricIdList, UMAUint4 numIds,
    UMAUint4 *operation, UMAUint4 *transactionID,
    DCIReturn **bufferAddress,
    UMAUint4 bufferSize, UMATimeVal *timeout);
DCIStatus dciPostData(UMAUint4 operation, UMAUint4 transactionID,
    DCIReturn *status, void *data, UMAUint4 dataSize,
```

```
    DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciUnregister(DCIClassId *classId,
    DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciSetClassAccess(DCIClassId *classIdList,DCIAccess *accessList,
    UMAUint4 numIds,DCIReturn **bufferAddress, UMAUint4 bufferSize);
DCIStatus dciSetInstAccess(DCIMetricId *metricIdList,DCIAccess *accessList,
    UMAUint4 numIds,DCIReturn **bufferAddress, UMAUint4 bufferSize,
    UMATimeVal *timeout);

/* Event Delivery */
DCIStatus dciWaitEvent(DCIHandle handle, DCIMetricId *metricIdList,
    UMAUint4 numIds, DCIEventReturn **bufferAddress,
    UMAUint4 bufferSize, void **dataAddress, UMAUint4 *dataSize,
    UMATimeVal *timeout , UMAUint4 eventFlags);
DCIStatus dciPostEvent(DCIMetricId *metricId,
    UMAUint4 eventDataCount, UMAUint4 eventDataSize,
    UMAVarLenData *eventData);
#endif /* _NO_PROTO */

#endif /* _SYS_DCI_H_ */
```

Glossary

API
Application Programming Interface. A standard interface for program access to a set of services. The DCI API is defined in this document.

Data Capture Interface
The API for the Data Capture Layer.

Data Capture Layer
The lowest layer in the UMA metrics architecture. It is concerned with the collection of raw data from the system.

DCI
Data Capture Interface

DCI server
An abstraction provided by the DCI. The DCI server provides a set of services to metrics providers and consumers.

DCL
Data Capture Layer

event, event metric
An event is a metric. An occurrence of some activity of interest to a metrics consumer (for example, thread termination).

i18n
abbreviation used in this specification for the term *internationalization* (which has 18 letters between its first and last letters).

metric
A single measurement. Metrics have unique identifiers defined through the metrics name space. A metric is either a polled metric (indicating some statistic or other data) or it is an event. Polled metrics have metric values that can be obtained by querying the DCI. Events may have associated data returned with the event when it occurs. (See Section 3.4.3 on page 47).

metric class
Metrics are grouped into metric classes. Classes are organised in a hierarchy. A metric class holds no metric values. It is simply a placeholder in the namespace. It should be viewed as a template.

metric class instance
Metric values are associated with instantiations of metric classes. For example, there could be a class for per-thread statistics; associated with this class could be many instances, each one identified by its thread id. A particular thread's statistics would be available by querying the DCI for that specific metric class instance.

metrics consumer
Any application which needs to import metrics.

metrics provider
Any subsystem which has metrics to export. The subsystem can either be the operating system or applications.

metric value
The value of a polled metric is referred to as a metric value. Note that metrics that are events do not have metric values.

MLI
The Measurement Layer Interface. An upper level service, one of the possible DCI consumers, that provides a measurement control and data delivery mechanism.

multiprocessor system
Any machine which contains more than one processor but appears to the user to be running a single operating system.

octet
An eight bit unit of storage.

operating system
Privileged software which controls hardware resources.

PMWG
Performance Management Working Group. The group which has specified the Universal Measurement Architecture.

polled metric
A polled metric is a metric. It typically corresponds to a numerical count of some system activity, some statistic, or possibly some configuration information (for example, the number of processors).

security
That part of an operating system concerned with controlling access to information.

system space
An address space in which privileged programs, such as the operating system, are run.

user space
An address space in which unprivileged programs are run. Examples of such programs are applications or system services which do not require direct access to hardware resources.

UMA
Universal Measurement Architecture. The collection of the DCI, Data Pool and MLI that provides a complete performance measurement architecture.

Index

CAE Specification

Part 4:
UMA Data Pool Definitions (DPD)

The Open Group

Contents

List of Figures

List of Tables

Preface

This Document

This document is a CAE specification. It defines a performance data pool for the analysis and management of computer systems, and an organisation to facilitate the collection and use of such data. This set of performance metrics may be accessed by the two UMA interfaces:

- Measurement Layer Interface (MLI) which is described in Part 2 of this specification.

  The MLI provides the interface between measurement applications and a UMA data services layer, which interacts with the UMA measurement control layer to provide required performance data.

- Data Capture Interface (DCI) which is described in Part 3 of this specification.

  The DCI is the interface between the data capture layer and the measurement control layer of the UMA architecture.

The UMA Guide (see Part 1 of this specification). reviews the issues surrounding performance measurement in Open Systems, describes the general UMA architecture, and discusses user considerations in adopting the UMA.

Audience

The audience for the metrics defined in this document ranges from the end-user to the system developer. End-users (*customers*) will find them useful in measuring productivity. Performance analysts/engineers can use them for modelling, tuning and measuring/predicting capacity in systems/applications. Data centres and MIS organisations can use them to assess the quantity and quality of the computing services provided under Service Level Agreements with their customers. Hardware and software vendors can use them to assure the performance of their products during development and after release. Performance management application vendors can use them as standard metrics with the open application interface UMA provides, to develop their products.

Structure

- **Chapter 1, Introduction** — provides an overview of the datapool.
- **Chapter 2, About the Datapool** — explains key terminology, and how data is grouped into ‘‘messages’’, and the different classes/subclasses and ‘‘levels’’ within this organization.
- **Chapter 3, Data Capture Overview** — describes the assumptions about the data capture process, from obtaining the raw data to producing a finished metric for use by a Measurement Application Program (MAP).
- **Chapter 4, Datapool Metrics** — establishes the Level 0 metrics and a proposed set of Level 1 metrics, including relevant MLI and DCI information.

Acknowledgements

This specification was developed by the Performance Management Working Group. The PMWG was originally part of UNIX International, and is now part of the Computer Measurement Group.

X/Open gratefully acknowledges the work of the PMWG in the development of this specification and in the review process for this publication.

Major contributors to the Data Pool Definitions specification include:

Sara Abraham	Amdahl Corporation	Robert Berry	IBM Corporation
Adrian Cockcroft	SUN Microsystems	Lewis T. Flynn‡	Amdahl Corporation
Anthony J. Gaseor‡	AT&T Bell Laboratories	Javad Habibi	Amdahl Corporation
Marge Momberger	IBM Corporation	Henry Newman‡	Instrumental, Inc.
David Potter	Open Systems Performance	Jim Richard	Amdahl Corporation
Jim Van Sciver	Open Software Foundation	Yefim Somin	BGS Systems
Leon Traister	Amdahl Corporation	Manda Sury‡	IBM Corporation
Steve Whitney	Boeing Computer Services	Elizabeth Williams	Super Computer Research

Participants who have made contributions to the process of developing these specifications are listed below along with their corporate affiliation at the time of their contribution. Our sincere apologies to anyone whom we may have missed.

Subhash Agrawal	BGS Systems	Barrie Archer	ICL
Peter Benoit	Digital Equipment Corp.	Tom Beretvas	IBM Corporation
Wolfgang Blau	Tandem Computers, Inc.	Jim Busse	NCR Corporation
David Butchart	Digital Equipment Corp.	David Chadwick	Performance Awareness Corp.
Ram Chelluri	AT&T Global Information Solutions	Danny Chen	AT&T Bell Laboratories
Niels Christiansen	IBM Corporation	Paul Curtis	Hitachi computer Products (America), Inc.
Paul Douglas	Digital Equipment Corp.	Janice Dumont	AT&T Bell Laboratories
Ansgar Erlenkoetter	Tandem Computers, Inc.	Paul Farr	Aim Technology
Jerome Feder	UNIX System Laboratories	Mark Feldman	Sequent Computer Systems, Inc.
Thierry Fevrier	Hewlett-Packard	Ken Gartner	Hitachi Computer Products (America), Inc.
Joseph Glenski	Cray Research, Inc.	Dave Glover	Hewlett-Packard
Jay Goldberg	UNIX System Laboratories	William Hidden	Open Software Foundation
Liz Hookway	NCR Corporation	John Howell	Amdahl Corporation
Ken Huffman	Hewlett-Packard	Mario Jauvin	Bell Northern Research
Chester John	IBM Corporation	Sue John	IBM Corporation
Rebecca Koskela	Cray Research, Inc.	Bill Laurune	Digital Equipment Corp.
Ted Lehr	IBM Corporation	Greg Mansfield	Instrumental
	Shane McCarron	UNIX International	Michael Meissner
Bernice Moy	Open Software Foundation	Jee-Fung Pang	Digital Equipment Corp.
James Pitcairn-Hill	Open Software Foundation	Melur K. Raghuraman	Digital Equipment Corp.
O. T. Satyanarayanan	Amdahl Corporation	Steve Sonnenberg	Landmark Systems
Douglas R. Souders	UNIX System Laboratories	Jaap Vermeulen	Sequent Computer Systems, Inc.
Michael Wallulis	Digital Equipment Corp.	Ping Wang	Open Software Foundation
Willie Williams	Open Software Foundation	Neal Wyse	Sequent Computer Systems, Inc.
Seung Yoo	Amdahl Corporation		

† Editor

†† Past Editor

Chapter 1

Introduction

1.1 Purpose

This document is one of a family of documents that comprise the Universal Measurement Architecture (UMA), and which define interfaces and data formats for Performance Measurement. UMA was originally defined by the Performance Management Working Group (PMWG) and subsequently adopted by The Open Group.

This document defines a performance data pool for the analysis and management of computer systems, and an organisation to facilitate the collection and use of such data.

The UMA is defined in the following documents:

- Guide to the Universal Measurement Architecture (see reference **UMA**). This document provides an overview of the UMA.
- UMA Measurement Layer Interface Specification (see reference **MLI**). This document defines functional characteristics for a high-level open Application Program Interface (API) to be used by Measurement Application Programs (MAPs) to request and receive data. It also defines header formats to be appended to the data captured by a low-level Data Capture Interface (DCI).
- UMA Data Capture Interface Specification (see reference **DCI**). This document defines a standard programming interface for capturing data provided by systems and applications.
- UMA Datapool Specification (this document).

1.2 Audience

The metrics defined in this document span a wide range of uses. The audience for these metrics ranges from the end-user to the system developer:

- End-users (*customers*) are concerned about adequate response time for their particular application (that is, productivity).
- The performance analyst/engineer uses these metrics for modelling the performance of new systems/applications or changes to existing ones, tuning the overall performance of a system or an application to a particular customer environment, and measuring the current and predicting future capacity needs.
- Data centres and MIS organisations, being service oriented, are concerned about the quantity and quality of the computing services provided under Service Level Agreements with their customers. In this case, the metrics are used for accounting, real-time monitoring to detect and correct poor service, tracking service quality with control charts of the key metrics specified in Service Level Agreements with customers, and workload characterisation and balancing.
- Hardware and software vendors like to assure the performance of their products during development and after release to the market.
- Performance management application vendors need standard metrics with an open application interface to make it economically feasible to develop such products.

1.3 Scope

The metrics defined in this document attempt to meet the data use needs for the various audiences mentioned above. Although the metrics are heavily influenced by the currently available measurements, an attempt is made to recommend new metrics to correct the deficiencies experienced with existing technology. Metrics are grouped into "Classes" amd "Subclasses" based on their functionality and content. Furthermore, to reflect the current technology and to accommodate for future growth, each metric is assigned a "Level" of maturity. Specifically, each metric belongs to one of the following four categories:

- Level 0
- Level 1
- Optional
- Platform / Vendor Specific.

The first three categories are part of the Datapool Standard. The level 0 specification is an attempt to formalize existing common practice, and should be implementable on the bulk of the UNIX installed base, using OS releases that were available in 1995. The Level 1 specification is to provide direction for OS vendors, and defines a common set of metrics that are needed to implement performance management tools. Additional details on the levels are found in Chapter 2.

This document defines no interfaces or other architecture, only data and a data organisation.

Performance and capacity management of operating systems have been considered ‘‘internal’’ to the operating system and as such differ from one operating system to another and from one implementation to another. Most operating systems have, as a matter of necessity, performance analysis modules, narrowly targeted at the type of hardware, software and networking facilities implemented within the system.

Most operating systems provide ad-hoc developed or tailored performance metrics. Some of these tools are developed as internal support tools for benchmarking or on demand from performance analysts and capacity planners. These tools are generally also confined to one machine only and can not be interrogated remotely.

The new era of networking and interoperability views performance management and capacity planning from user’s perspective. Multiple machines and operating systems can be involved in the interaction with the user. This approach requires capture and presentation of performance metrics to be clearly defined and portable between platforms and operating systems.

In addition, the data used in this specification is presented as vendor and implementation independent as possible, however, a mechanism is provided for vendor data extensions.

1.4 Conformance

Support for Datapool level 0 is mandatory, while support for higher levels is optional.

Conformance to levels higher than zero means that metrics defined as mandatory in such levels must all be provided.

Chapter 2

About the Datapool

2.1 Data Organisation

In this data organization, data is grouped into ‘‘messages’’. These messages contain a standard header followed by one or more data items. A message class and subclass uniquely identify the contents of these messages.

The rationale behind this organization is the need to provide a well defined format suitable for postprocessing either locally or at another system. This format facilitates the writing of data reduction and display programs and would not require any program modification or recompilation merely because new data is made available. Current operating technologies do not lend themselves to this function and a different type of structure was perceived to be necessary.

The fact that the data uses a message format does not imply that the underlying implementation uses message passing. Any implementation is allowed so long as the data presented by the UMA Measurement Layer Interface (see reference **MLI**) is in the correct format.

The actual layout of the structures allows for both forward and backward compatibility. In particular, the arrangement of data items in these structures is done in such a fashion that new items may be added without requiring recompilation of existing applications. Items that become obsolete will be zero-filled, whereas new items are added at the end.

2.2 Data Standards

Data items have been categorized, in part, by the degree of standardization deemed necessary. This process included the division of the data items of each subclass into groups:

- *basic*
- *optional*
- *extension*.

The *basic* data items are those that should be given first priority to be made available in every implementation.

To distinguish technology currently available form future products, this group is further subdivided into Level 0 and Level 1.

The *optional* data items are those that may be implemented in every implementation (that is, they are standard but optional).

There are also *extension* data items, which are those that may be implemented by at least one vendor (that is, vendor specific). No data items of this type are defined in this document.

Therefore, each metric belongs to one of the following four categories:

- Level 0
- Level 1
- Optional
- Platform / Vendor Specific.

The first three categories are part of the Datapool Standard. The prefix in the DatumID of each metric reflects its level — with a ‘‘0’’ for Level 0, a ‘‘1’’ for Level 1 and the string ‘‘opt’’ for Optional.

2.2.1 Level 0 Metrics (UMA DP95)

The level 0 specification is an attempt to formalize existing common practice, and should be implementable on the bulk of the UNIX installed base, using OS releases that were available in 1995. To comply with the UMA Datapool standard, an implementation must provide all applicable level 0 subclasses, and all of the level 0 metrics in those subclasses must be provided.

2.2.2 Level 1 Metrics (UMA DP97)

To provide direction for OS vendors, a common set of metrics that are needed to implement performance management tools is defined. These should be implementable in a reasonable timeframe, such that a future (1997 or later) release of the OS could be expected to provide the additional metrics. It is expected that many OS vendors will provide some of the level 1 metrics on their intermediate releases. Individual level 1 metrics should be provided where available in advance of the full set. To be compliant with the level 1 standard, all applicable level 1 subclasses must be provided.

2.2.3 Optional Metrics

Optional metrics are defined to be implementation specific. They will never be required on all OS platforms, but they are expected to be available on platforms that share common implementations. To ensure consistency they are defined as optional rather than as platform specific. Optional metric identification numbers start at 85.

2.2.4 Platform or Vendor Specific Metrics

Metrics that are only expected to exist on one platform will not be considered for inclusion in the UMA Datapool standard. They can be specified and provided as vendor extensions in vendor specific classes.

2.2.5 Management Application Assumptions

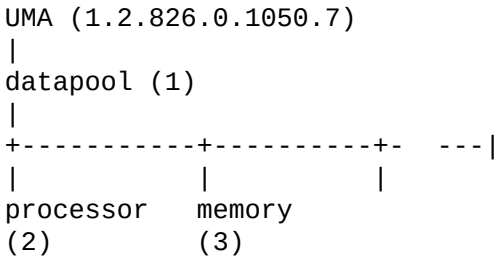
A management application can assume that all level 0 metrics will exist in a class. When interfacing with UMA DP95 compliant systems, existence of additional level 1 and optional metrics will need to be determined at run time. When interfacing with a UMA DP97 compliant system, the management application can assume that all level 1 metrics will exist, and only need test for optional metrics.

2.3 Uniqueness of Identifiers

In order to provide a mechanism for the assignment of unique identifiers to metrics, a naming convention based on object identifiers has been adopted. The following prefix has been allocated:

```
ISO(1); National Member Body(2); UK(826); DISC(0); X/Open(1050); UMA(7)
```

The resultant naming structure is shown below:



The use of object identifiers allows for *standard* metrics to be allocated identifiers within the UMA naming tree. It provides for extensibility by allowing implementors to define identifiers within their own parts of the naming tree. Thus vendor-specific metrics can be easily incorporated into the UMA without the need to have an identifier allocated by an external registration authority.

It is anticipated that, in the future, further standard metrics will be defined, and mechanisms will be established for the administration of the namespace.

2.4 Default UMA WorkInfo Types

The default UMAWorkinfo enumeration has the following fields:

UMA_WORKINFO_PROJECT

UMA_WORKINFO_GROUP_ID

UMA_WORKINFO_EFFECTIVE_GROUP_ID

UMA_WORKINFO_USER_ID

UMA_WORKINFO_EFFECTIVE_USER_ID

UMA_WORKINFO_SESSION_ID

UMA_WORKINFO_TTY

UMA_WORKINFO_NQS

UMA_WORKINFO_SCHEDULING_CLASS

UMA_WORKINFO_SCHED_GRP

UMA_WORKINFO_TRANSACTION_ID

UMA_WORKINFO_PROCESS_GRP

UMA_WORKINFO_PARENT_PROCESS_ID

UMA_WORKINFO_COMMAND_NAME

UMA_WORKINFO_PROCESS_ID

UMA_WORKINFO_THREAD_ID

Chapter 3

Data Capture Overview

The following is a brief overview of the data capture process from obtaining the raw data to producing a finished metric for use by a Measurement Application Program (MAP).

3.1 Data Capture Modes

The modes of capturing data for either presentation as reports or subsequent use by other tools includes:

- *Sampled Data*
  Data which is measured by repetitive capture (at a sampling rate) and presumably accumulated in a counter
- *Interval Data*
  Data which represents the *incremental* activity within a certain time interval
- *Event Data*
  Data which provides notification of the occurrence of a particular state within a subsystem
- *Trace Data*
  Data which captures a succession of subsystem states, usually in substantial detail.

  Traces in UMA are implemented as high frequency events and are normally directed to a file.

Note: Event and interval extension headers meet the needs of all four data capture modes. Interval messages represent both sampled and interval data, and event messages represent both event and trace data.

3.2 Metrics

The end metrics that the MAPs see are usually formed from raw counters that are kept by the system, subsystem or application. For interval data, the metric is formed by taking the difference of two samples of a counter which is continually incremented since the last restart. For sampled data, two metrics are formed for an interval: a count of events and the number of samples.

Note: Data items in this document are the end metrics that the MAPs see and not the raw counter data.

3.3 From Raw Data to End Metric

The process of providing, collecting, transforming and delivering the data to the MAPs is one of the main concerns for the UMA. Figure 3-1 depicts the basic architecture. The following is a very simplistic overview of the process under discussion. The MAPs request a class/subclass of metrics through the Measurement Level Interface (MLI) to the data services/measurement control layer. Measurement control merges the requests, synchronizes the capture, provides headers and timestamps and requests the current raw counter data through the Data Capture Interface (DCI). Once it has the data, measurement control/data services can difference interval data, transform the machine dependent data to a standard form and provide other services. The Data Capture Layer is basically responsible for gathering the counter data from the kernel, subsystem or application and passing it through the DCI. The DCI document explains the different approaches for gathering the data and the necessary programming interfaces for the kernel, subsystems and applications to provide the data.

To address the problem of generating too much data at the per process/thread level, a MAP can request granularity levels higher than a process. This higher granularity is based on the user, session, transaction, fair share group, etc, identifiers in the **UMAWorkInfo** structure. Messages/records would only be cut for the level(s) of granularity requested, with the lowest levels requested only during emergencies or testing. Another control on the volume of data produced is the selection of event or interval data. Event data may be fine for rudimentary accounting (that is, end of process, job, session, login) whereas interval data would provide near real-time knowledge of long or never ending processes for accounting, resource management, problem resolution, etc.

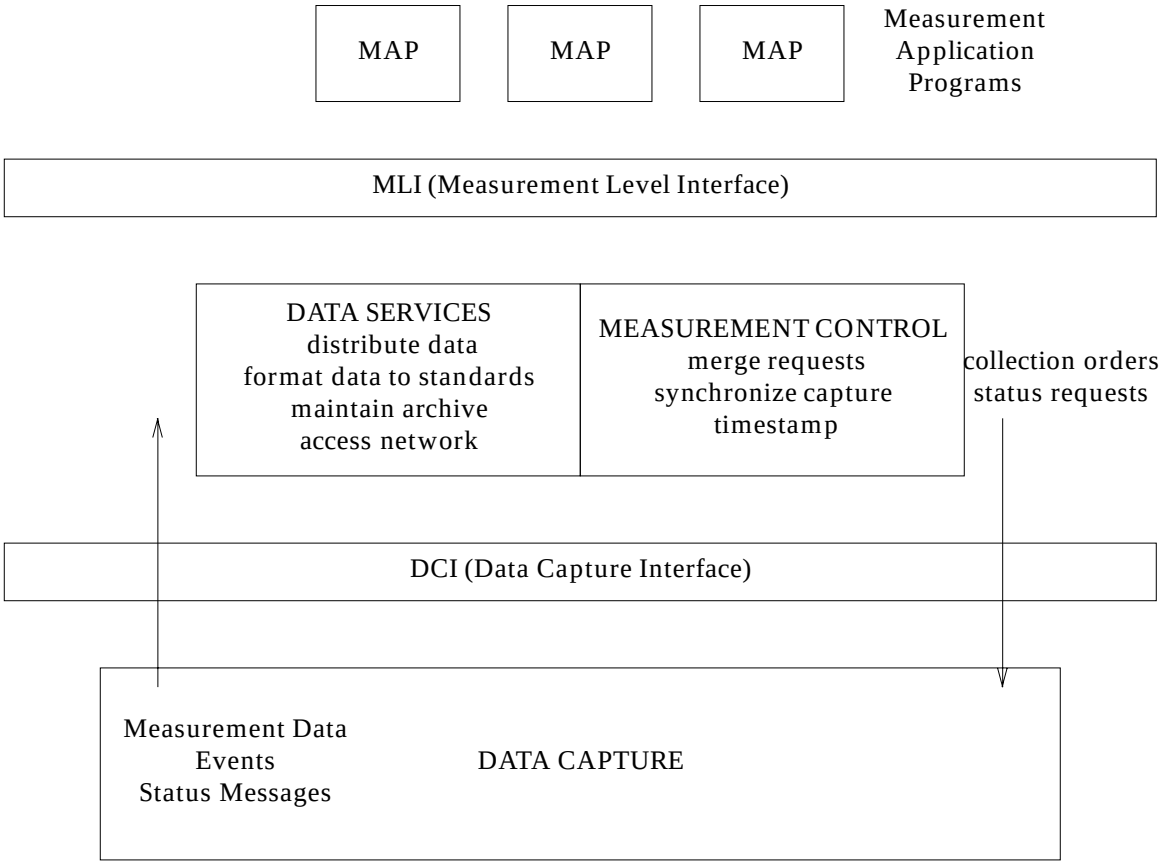


Figure 3-1 The Layers and Interfaces for the UMA

3.4 From Raw Data to Statistics

Most of the end metrics in the data pool are interval data which are easily calculated by taking two samples of the raw counters (usually from the kernel, which have been incremented since the last boot) and taking the difference (while watching for overflow conditions). This sampling and differencing according to the UMA is done at the data services layer and not inside the kernel. These interval values then could be used by the MAPs to calculate metrics such as rates (for example, blocks/sec), service times (for example, sec/block) and utilization (for example, 50% busy) for the interval. The MAPs could also take a set of these interval samples for a given period (for example, 1st, 2nd and 3rd shifts, peak hours during prime shift, a benchmark run) and produce statistics about the distribution of the values such as the following: mean, maximum, variance, standard deviation, 95th percentile, and distribution histograms. Such statistics can provide valuable information about the distribution of a set of data. But, except for the mean, these statistics based on interval values are incorrect and misleading unless the underlying raw counters are ''well behaved'' so that the distribution of the interval values reflect the distribution of the raw data.

3.4.1 Variable Metrics and Sum of Squares

As indicated above, when a kernel counter can change by widely different values during an interval (for example, one logical read could request a gigabyte while the next might be for a single byte), the statistics based on a set of interval values based on this counter will almost always be false. But, since information about the distribution is needed to understand a variable metric, it will be necessary for metric providers (the kernel in the case of logical read requests) to provide additional counters to support distributional statistics. The simplest such addition is to add a counter that accumulates a ''sum of squares'' of the changes to the primary counter. This can be done with very small overhead to the provider of a multiply, an add, and some memory references. For the logical read request example, the kernel will already have two counters, the number of requests and the sum of number of bytes per request; the additional counter would be the sum of the square of the number of bytes per request. The addition of such counters will allow the true calculation of variance and standard deviation for a metric like the number of bytes per request. So far the sum of squares counter is the only one included in the UMA data pool. The inclusion of additional counters to support other distributional statistics is still an open issue.

In the following sections that describe metrics in UMA subclasses, sum of squares metrics have been defined for certain metrics that are based on counters that can change by a variable amount during an interval. They are are listed in the optional data segment for a message subclass in which they occur.

3.4.2 Statistics Across Multiple Intervals

To calculate the mean over multiple intervals, two counters must be incremented by the kernel. One is the count of occurrences of the value being measured and the other is the sum of the values. For example, if read statistics are required, one counter is the number of reads during the interval; the second counter is the sum of the number of bytes read with each read during the interval. For a single interval, the mean is the sum of the number of bytes read divided by the number of reads for that interval. The maximum of a metric over multiple intervals is easily calculated as the maximum of the set of maximums for each interval. The mean for multiple intervals, however, cannot be computed as the mean of the set of means for each interval. It is calculated as the total over multiple intervals of the sum of the number of bytes read with each read divided by the total over multiple intervals of the number of reads.

To compute the variance and standard deviation over a single interval as well as multiple intervals, a third counter, which is the sum of the squares of the value, must be incremented by the kernel. Using the example of read statistics, this counter is the sum of squares of the number of bytes read with each read during the interval. The variance then is calculated as the mean of the sum of squares minus the square of the mean, that is, the sum of squares of the number of bytes read divided by the number of reads minus the square of the sum of the number of bytes read divided by the number of reads. As in the case above for the mean, the variance over multiple intervals requires all three counters described above, each summed over the multiple intervals.

3.4.3 Kernel Level Sampling versus Data Services Level

Some sampling data can be broken down into basic kernel counters and gathered as interval data. Two such metrics are the average run-queue length and occupancy. At every clock tick, a sample count is incremented, the number of runnable but unloaded processes added to a running counter, and a count incremented for the samples when the runnable but unloaded process count was not zero. The average run-queue length is the sampled sum of runnable but unloaded processes divided by the non-zero sample count. The run-queue occupancy is the non-zero sample count divided by the sample count. This kernel-level sampling technique should only be used when the overhead is justified by the frequency of change of the metric and the metric importance.

Some sampling data should not be collected by the kernel. One example is the current number of logins in a fair share group, on a system or on a particular front-end-processor. Here the sample data that does not change frequently enough to justify kernel level tracking. Several metrics have been proposed which request the peak values for a given interval. If the kernel kept this value, it could only keep one measure and would need to reset the value at the beginning of the interval (the smallest of several intervals requested). (Note that peak value since last boot would not be useful.) Instead an adequate number of samples (30 minimally) of instantaneous values should be collected (possibly at the end of the interval) for the period in question (for example, a shift, test period, peak period). Then the 95th percentile or maximum of these samples could be determined by the MAP for that period.

3.5 Resource versus Workload Analysis

It is likely that in the future there will be a shift of focus from system oriented resource analysis to an end-user oriented workload analysis. The typical analysis focuses on the resources (CPU, memory, disk, networks, etc.) being consumed but nothing about the applications consuming them. These resources can be tuned against some rules of thumb but one will never know what positive or negative impact this has on the end-user transaction response time. A more effective approach would be to monitor for worsening application transaction response and then tune the resources that are causing the problem. This requires collection of the transaction response time components (that is, delays at the CPU, disks, memory, networks, etc.).

The importance of end-user workload analysis has and will continue to have a profound influence on the selection of metrics and the formation of classes and subclasses. Note that some classes center around key system resources (for example, processor, memory, disks, streams, IPC, networks). The global or device subclasses attempt to tell whether a particular resource may be in trouble. The per process subclasses attempt to tell what resources may be causing an application trouble. In many cases, the connection from the process to the resource passes through several buffers where direct tracking for the individual process is lost. For example, one cannot track the I/O for a process through a memory buffer cache to a disk or through streams to other I/O devices. Where this connection is lost, one must rely on some statistical correlation between the individual process response and the global resource response. To form this correlation, the per process data must be collected on the same interval as the global resource data. Although this is a pragmatic answer, one should demand (long term) real measurements even if they are hard to develop. Tracing response to resource is necessary for true capacity planning. The end goal is a transaction model at the application level.

Chapter 4

Datapool Metrics

4.1 Introduction

This chapter establishes the Level 0 metrics and a proposed set of Level 1 metrics. The data is organized by classes in the following order:

- Configuration Information
- Processor
- Memory
- IPC
- Scheduling
- Disk Device data
- Global File System
- Network Protocol
- Accounting.

4.2 Definitions for MLI Attributes

This section defines the headings used in the ‘‘MLI Attributes’’ columns of the tables that define the contents of metric and configuration classes.

Instance Array
: Instance identifiers are typically mapped level-by-level to a sequential list of instance tags. Alternatively, the instance identifiers may be mapped as a data array. If this is the case, this will be so indicated in this column.

Offset
: Applies to the built-in, or ‘‘Canonical C’’, form of data mapping in MLI data UDUs. This offset is the distance in bytes from the start of data entries in the subclass segment to the current data object or descriptor.

VLDS Object (Metric Enumeration)
: Variable length data objects such as arrays and certain text strings are mapped into a section of an MLI data UDU message segment called the VLDS (Variable Length Data Section). The data objects are pointed to by descriptors in the fixed section of the segment.

4.3 Configuration Information Class

This class must provide enough information to determine a performance rating for the system. It is also intended to allow the topology of a system to be determined. For example the infomation that several disk instances share a common SCSI bus or are striped together into a logical volume may allow diagnosis of a problem.

Configuration data is expected to change infrequently, and an event signals that a change has occurred. All configuration classes generate an event to indicate when they need to be reread. To flag this in this document each subclass has a pseudo-metric event indicator.

To allow data in one subclass to refer to instances in other subclasses a new DCI type is required called a cross reference. It contains the last four digits of an OID and an instance identifier. UMA defines the root of the OID (1.2.826.0.1050.7) + datapool. The additional parameters define:

- UNIX_datapool(1) / vendor_datapool(x)
- class
- subclass
- metric
- instance.

The implementation of this type is not yet defined. The current suggestion is that it be coded as an ASCII variable length string, which would be parsed by a management application. The suggested format of the string is to have the ‘‘.’’ separated OID followed by an optional ‘‘:’’ and the metric instance. If in the future it would be useful to refer to OID’s outside the datapool (for example, SNMP MIB), a full OID could be prefixed by a ‘‘/’’, analogous with the UNIX root filesystem. In practice all the initial UNIX datapool definitions will start with ‘‘1.’’. For example, ‘‘1.4.3.7:22’’ indicates the UNIX datapool, the 4th class, the 3rd subclass, the 7th metric and the 22nd instance.

The I/O configuration Subclasses, and the logical volumes are organized using the xref concept to tie them together. Where possible the xref is optional, so that the system could be represented as a collection of disconnected buses, controllers and disks if there is no way to determine the hierarchy. For logical volumes the xref is needed to tie the disks together into volumes.

The full device hierarchy is representable. It is not simple but neither are large systems. The information is available in all systems so it is part of the level 0 datapool. It can be cross-referenced by other metrics as needed and can be displayed by a management application. It is flexible enough to handle arbitrary I/O bus types, and storage controllers (channel, SCSI, fibrechannel) including targets that contain sub devices such as disk arrays, tape silos and optical jukeboxes. It maps to the c0t0d0s0 format used by several UNIXes, but allows c0t0s0 and sd0 formats to be used by skipping levels in the hierarchy.

Specific per-device configuration data cross-references a position in the hierarchy. Network and other io devices are included. Vendor specific utilization data can reference this hierarchy to indicate situations where too many devices are causing contention on a bus.

One simple example (Intel PC) and one complex (Multiprocessor Server) example are provided to show how the I/O configuration is intended to be used in practice.

The hierarchy is ordered in a top down manner, with upwards pointing references so that many lower level entries can reference one higher level entry. For example many I/O cards could be plugged into one I/O bus.

In the diagrams below (see Figure 4-1 and Figure 4-2), a few key metrics are shown for each class. Most classes have additional metrics that provide more detail.

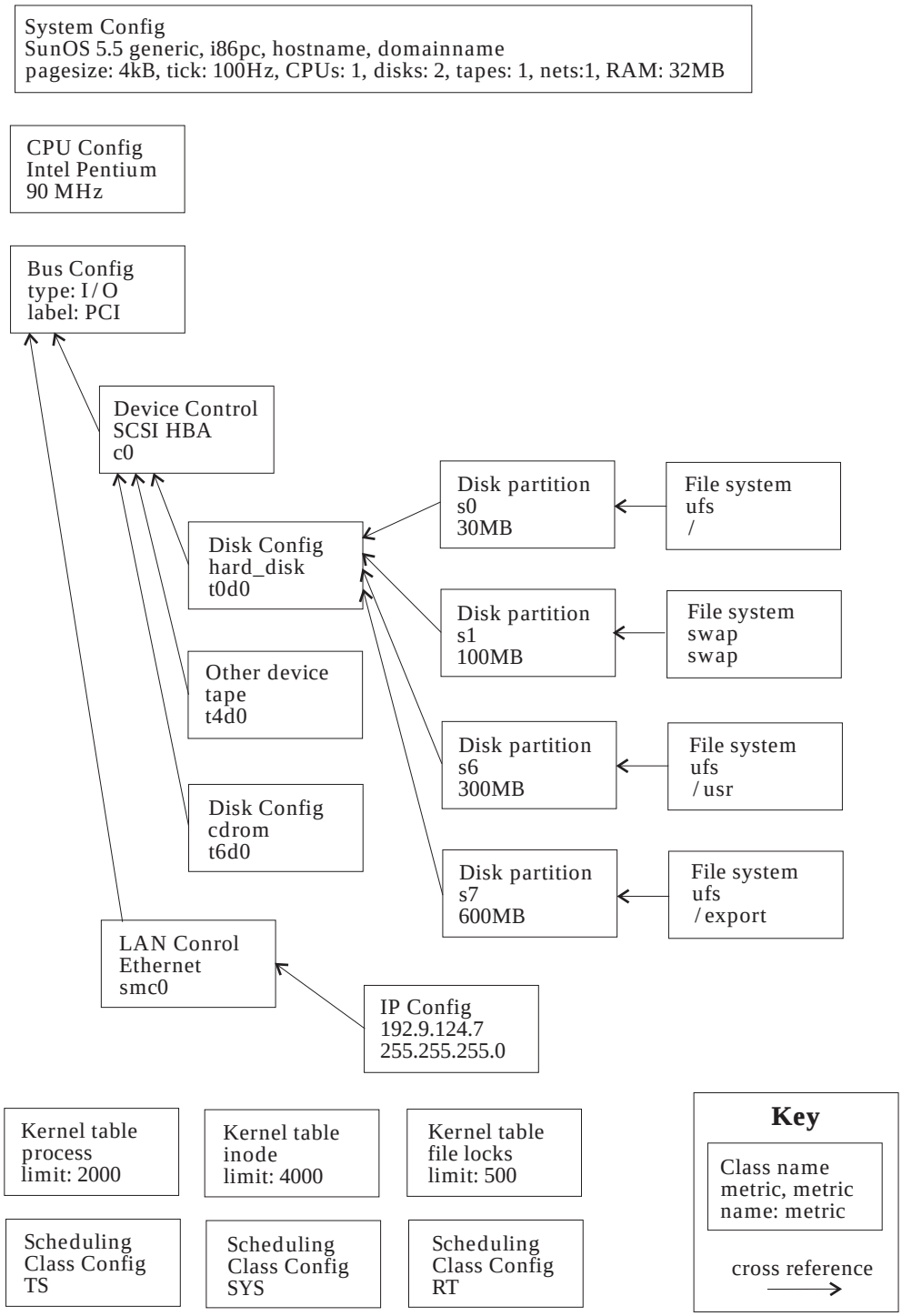


Figure 4-1 Simple Configuration Example - Intel PC Running Solaris

The system configuration class outlines the system and the single CPU is described. The PCI bus has two devices, a SCSI host bus adapter (HBA) and an ethernet controller. The SCSI HBA controls three devices, a hard disk, a tape and a cdrom. The disk is divided into four partitions, each holding a filesystem or swap space. The ethernet is configured with an IP address. Three kernel tables have defined limits. Three possible scheduling classes are described.

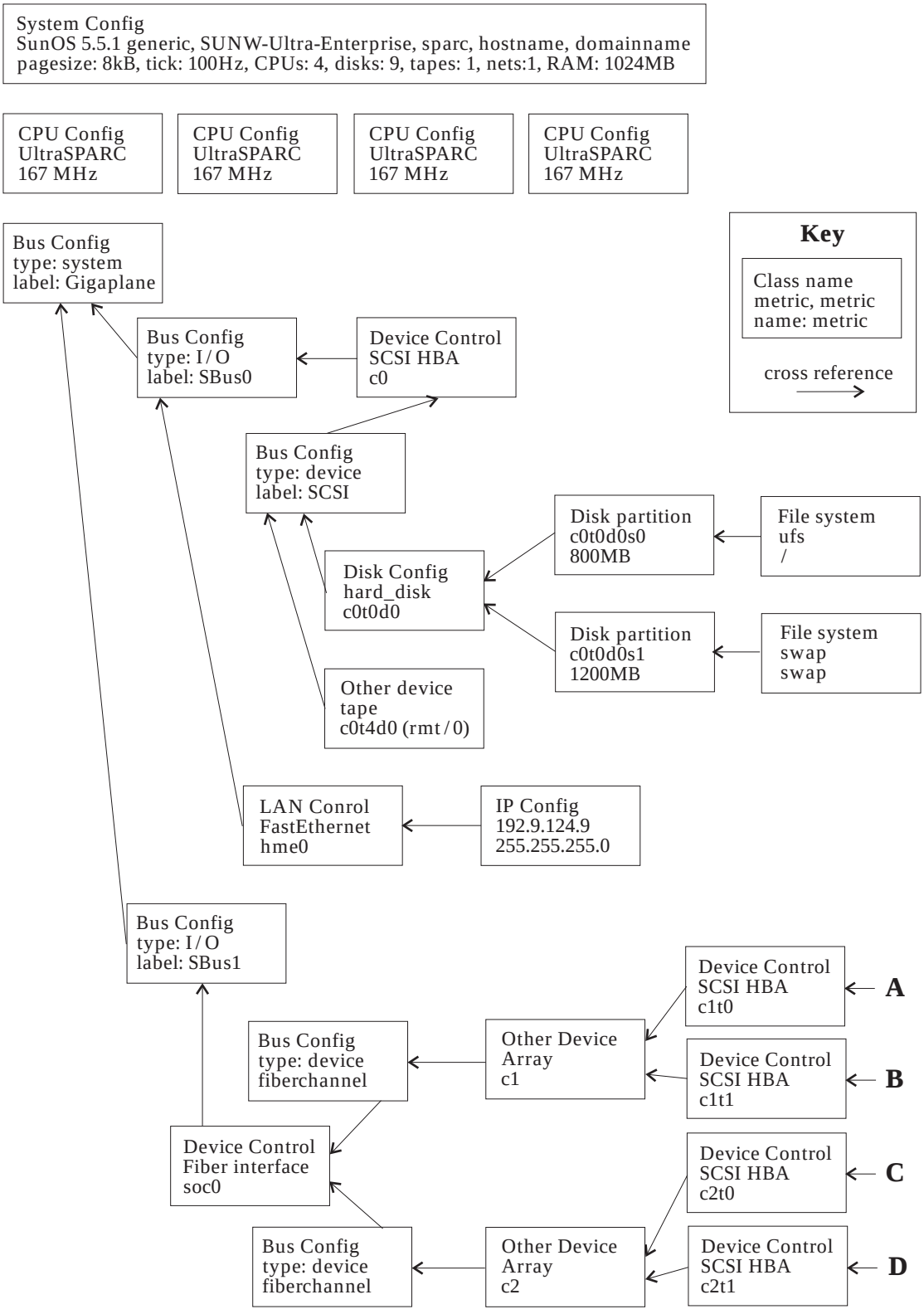


Figure 4-2 Complex Configuration - MP Server with Mirrored Disk Arrays

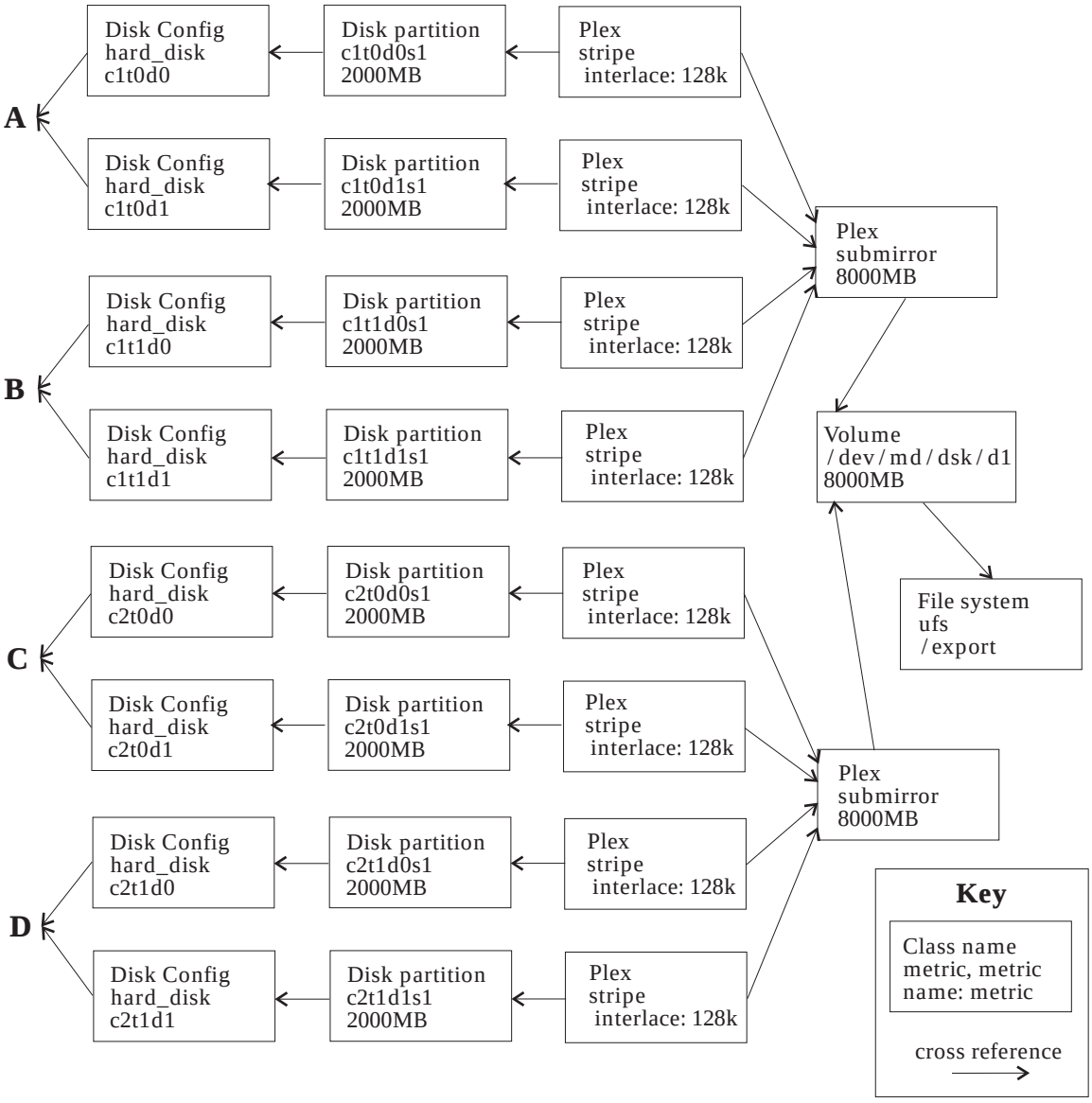


Figure 4-3 Volume Config for Striped Filesystem Mirrored Across Controllers

The system configuration again gives the basic parameters and the four CPUs are described. The system backplane contains two independent I/O buses. The first SBus contains a SCSI HBA and Fast Ethernet controllers. The SCSI bus itself is described explicitly this time, its speed and width can be reported. A disk containing the OS and swap space, and a tape drive are connected to it. The Fast Ethernet has an IP address configured (multiple IP addresses and other protocols could be configured on the same interface). The second Sbus contains a Fiberchannel serial optical controller (soc) that has dual fiber interfaces. Two independent fiberchannel "buses" are described, each is connected to an intelligent storage array controller. Each array controller has multiple SCSI HBAs. In the diagram two per-array are shown, with off-page markers A,B, C and D.

The second diagram shows that two disks are connected to each SCSI HBA. In this case the SCSI bus configuration itself has been omitted for simplicity. A single large partition is configured on

each of the eight disks. A set of four plexes are used to produce a stripe with a 128KB interlace. The plex configuration has two cross references, as it has to refer to a disk partition and a place in the volume hierarchy. In this case a pair of second level plexes configured as submirrors. They in turn refer to the volume device, which refers to the mounted file system configuration.

The kernel table and scheduler configuration classes have been omitted from the diagram through lack of space. They would be the same as the first example, but with different limit values.

4.3.1 Subclass — System Configuration

This class provides the basic configuration parameters for a system. The system description includes OS and machine vendor and compatability information, the identity of the machine (node name and optional domain name), and configuration counts for the primary system components. To see how many components are online or enabled their respective class instances have a status indicator that needs to be checked.

Some systems have the ability to dynamically increase these counts online, or may not recognise the presence of a device until it is first accessed. A *config_change_event* is generated if these values change.

The metric OID and description is shown along with a sample source for the metric and example value.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
OS_name	0/1.1.1	TEXTSTRING	NOUNITS	uname -s (sunOS)	0	Yes	No
OS_release	0/1.1.2	TEXTSTRING	NOUNITS	uname -r (5.5)	8	Yes	No
OS_version	0/1.1.3	TEXTSTRING	NOUNITS	uname -v (Generic)	16	Yes	No
vendor_model_name	0/1.1.4	TEXTSTRING	NOUNITS	uname -i (SUNW,SPARCstation-10)	24	Yes	No
hardware_class	0/1.1.5	TEXTSTRING	NOUNITS	uname -m (sun4m)	32	Yes	No
Processor_type	0/1.1.6	TEXTSTRING	NOUNITS	uname -p (sparc)	40	Yes	No
nodename	0/1.1.7	TEXTSTRING	NOUNITS	uname -n (fred)	48	Yes	No
page_size	0/1.1.8	UINT4	BYTES	sysconf(_SC_PAGESIZE)	56	No	No
clock_tick_freq	0/1.1.9	UINT4	HZ	sysconf(_SC_CLK_TCK)	60	No	No
cpu_timestamp_unit	0/1.1.10	UINT8	HZ	hertz time unit of measurement	64	No	No
cpus_configured	0/1.1.11	UINT4	COUNT	sysconf (_SC_NPROCESSORS_CONF)	72	No	No
disks_configured	0/1.1.12	UINT4	COUNT		76	No	No
tapes_configured	0/1.1.13	UINT4	COUNT		80	No	No
networks_configured	0/1.1.14	UINT4	COUNT		84	No	No
physical_memory_size	0/1.1.15	UINT4	MBYTES		88	No	No
boot_timestamp	0/1.1.16	UINT8	TIMESTAMP		92	No	No
domainname	0/1.1.17	TEXTSTRING	NOUNITS	(smcc.eng.sun.com)	100	Yes	No
defaultrouter	0/1.1.18	TEXTSTRING	NOUNITS	blank if no default	108	Yes	No
config_change_event	0/1.1.19	UINT4	NOUNITS	-	-	-	-

Table 4-1 System Configuration

4.3.2 Subclass — per-CPU Configuration

There is one instance of this subclass for each configured CPU. The status indicates whether the CPU is online, offline or failed. In some cases a CPU that has been taken offline may still be the target for device interrupts, but it will not have jobs scheduled onto it. Processor id is a system dependent value that should be used to label per-CPU data instead of the instance number. It is often non-contiguous on systems that do not have every CPU configured. Clock frequency of the CPU is vendor dependent. The cpu_clock_resolution is the unit for CPU usage metrics. It is often the same as the system clock tick frequency, but a high resolution CPU timer can be used to give more precise metrics. It is normally the same for every CPU, but some systems allow mixed CPU types and mixed clock rates, so it is replicated on a per-CPU basis.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
vendor_name	0/1.2.1	TEXTSTRING	NOUNITS	e.g. Intel	0	Yes	No
vendor_cpu_designation	0/1.2.2	TEXTSTRING	NOUNITS	e.g. Pentium	8	Yes	No
processor_id	0/1.2.3	UINT4	NOUNITS		16	No	No
cpu_status	1/1.2.4	ENUMERATION	NOUNITS		20	No	No
		offline	= 0	not in use at all			
		onforintr	= 1	not scheduled, but takes interrupts			
		online	= 2	normal operation			
		failed	= 3	failed implies offline			
cpu_clock_frequency	1/1.2.5	UINT8	HZ	e.g. 90,000,000 Hz	24	No	No
cpu_timestamp_resolution	1/1.2.6	UINT8	HZ	for CPU time measurement	32	No	No
config_change_event	0/1.2.7	UINT4	NOUNITS	-	-	-	-

Table 4-2 CPU Configuration

4.3.3 Subclass — Backplane, I/O or Device Bus Instance

This subclass identifies the buses in the system. There may be a backplane or system bus and a number of I/O buses containing controllers. Strings of devices may use a common bus to connect to a controller. Cross references between the instances create the hierarchy. The controllers on each I/O bus xref the instance they belong to. Bus performance metrics belong in a vendor specific subclass that should xref this configuration subclass and instance.

The defined optional configuration data includes an xref to another bus instance that is typically the backplane that a number of I/O buses share. The bus_version is dependent on the iobus_type. The bus_clock_frequency and bus_width can be used to calculate a peak bandwidth. The bus_data_packet_max indicates the maximum number of bytes sent in a single transfer. The bus_status needs to be defined for each type of bus, as each bus implements a protocol with different failure states.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
bus_type	0/1.3.1	ENUMERATION generic_io generic_sys generic_device	NOUNITS = 0 (generic I/O bus) = 1 (generic system backplane) = 2 (device bus like SCSI)		0	No	No
bus_id	0/1.3.2	UINT4	NOUNITS		4	No	No
bus_label	0/1.3.3	TEXTSTRING	NOUNITS	string naming this bus, e.g. "SBus"	8	Yes	No
parent_inst_xref	opt/1.3.85	TEXTSTRING	XREF	xref to extend hierarchy	0	Yes	No
bus_version	opt/1.3.86	TEXTSTRING	NOUNITS		8	Yes	No
bus_clock_frequency	opt/1.3.87	UINT4	HZ		16	No	No
bus_width in bytes	opt/1.3.88	UINT4	BYTES		20	No	No
bus_data_packet_max in bytes	opt/1.3.89	UINT4	BYTES		24	No	No
bus_status	opt/1.3.90	ENUMERATION offline online failed	NOUNITS = 0 = 1 = 2	type-specific status, for generic types	28	No	No
config_change_event	0/1.3.91	UINT4	NOUNITS	-	-	-	-

Table 4-3 Backplane, I/O or Device Bus Instance

4.3.4 Subclass — Device Controller Instance

Controllers are typically I/O cards of many different types that plug into an I/O bus, and control a string of devices. The type and label must be specified. The cross reference normally indicates which I/O bus the controller is attached to.

Optional metrics are defined that map onto mainframe channel controllers.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
controller_type	0/1.4.1	TEXTSTRING	NOUNITS		0	Yes	No
controller_label	0/1.4.2	TEXTSTRING	NOUNITS	string naming this controller (c0)	8	Yes	No
iobus_inst_xref	opt/1.4.85	TEXTSTRING	XREF		0	Yes	No
controller_vendor	opt/1.4.86	TEXTSTRING	NOUNITS		8	Yes	No
controller_model	opt/1.4.87	TEXTSTRING	NOUNITS		16	Yes	No
num_channels	opt/1.4.88	UINT4	COUNT		24	No	No
max_devices	opt/1.4.89	UINT4	COUNT		28	No	No
min_buf_mem	opt/1.4.90	UINT4	KBYTES		32	No	No
max_buf_mem	opt/1.4.91	UINT4	KBYTES		36	No	No
total_buf_mem	opt/1.4.92	UINT4	KBYTES		40	No	No
num_channel_paths	opt/1.4.93	UINT4	COUNT		44	No	No
status	opt/1.4.94	ENUMERATION	NOUNITS		48	No	No
		offline	= 0				
		online	= 1				
		failed	= 2				
path_address	opt/1.4.95	TEXTSTRING	NOUNITS		52	Yes	No
config_change_event	0/1.4.96	UINT4	NOUNITS	-	-	-	-

Table 4-4 Device Controller Instance

4.3.5 Subclass — Local Area Network Controller Instance

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
lan_type	0/1.5.1	TEXTSTRING	NOUNITS	e.g. Ethernet	0	Yes	No
lan_label	0/1.5.2	TEXTSTRING	NOUNITS	e.g. le0, nf0	8	Yes	No
MAC address	0/1.5.3	TEXTSTRING	NOUNITS	e.g. 8:0:20:1f:ab:cd	16	Yes	No
MTU	0/1.5.4	UINT4	BYTES		24	No	No
iobus_inst_xref	opt/1.5.85	TEXTSTRING	XREF	reference to iobus	28	Yes	No
lan_vendor	opt/1.5.86	TEXTSTRING	NOUNITS		36	Yes	No
lan_model	opt/1.5.87	TEXTSTRING	NOUNITS		44	Yes	No
status	opt/1.5.88	ENUMERATION	NOUNITS		52	No	No
		offline	= 0				
		online	= 1				
		failed	= 2				
config_change_event	0/1.5.89	UINT4	NOUNITS	-	-	-	-

Table 4-5 Local Area Network Controller Instance Label

Notes

WAN interfaces are not part of the datapool. Wide area point-to-point interfaces should have their own class. ATM is contentious but when used as a LAN substitute it belongs here. They are really the domain of Network Management, rather than System Performance Management.

4.3.6 Subclass — Disk Instance Configuration

A disk is a partitionable block device. Anything else is an "other". A hardware RAID disk subsystem that appears to the system as a single large disk belongs in this subclass. If the individual disks are apparent then each disk gets its own instance, and the RAID unit is dealt with as an "Other Device" and a volume description.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
disk_type	0/1.6.1	ENUMERATION Unknown harddisk cdrom floppy worm Hard_raid	NOUNITS = 0 = 1 = 2 = 3 = 4 = 5	one instance per-disk, coded as	0	No	No
disk_label	0/1.6.2	TEXTSTRING	NOUNITS	e.g. sd0 or c0t0 or c0d0 or c0t0d0	4	Yes	No
capacity	0/1.6.3	UINT4	KYTES		12	No	No
sector size	0/1.6.4	UINT4	BYTES		16	No	No
major	0/1.6.5	UINT4	COUNT		20	No	No
minor	0/1.6.6	UINT4	COUNT		24	No	No
status	1/1.6.7	ENUMERATION offline online failed	NOUNITS = 0 = 1 = 2		28	No	No
vendor	1/1.6.8	TEXTSTRING	NOUNITS		32	Yes	No
vendor_designation	1/1.6.9	TEXTSTRING	NOUNITS		40	Yes	No
controller_inst_xref	opt/1.6.85	TEXTSTRING	XREF	reference to a controller	0	Yes	No
controller2_inst_xref	opt/1.6.86	TEXTSTRING	XREF	optional 2nd controller for dual port (e.g. SCSIdisk)	8	Yes	No
device_cache_size	opt/1.6.87	UINT4	KBYTES	onboard cache	16	No	No
device_queue_size	opt/1.6.88	UINT4	COUNT	e.g. SCSI tagged command queue size	20	No	No
volume_label		TEXTSTRING	NOUNITS		24	Yes	No
config_change_event	0/1.6.89	UINT4	NOUNITS	-	-	-	-

Table 4-6 Disk Instance Configuration

4.3.7 Subclass — Other Device Instance Configuration

This subclas includes tapes, juke boxes and RAID units. The storage configuration of a RAID unit is not provided here as it is implicit in the way volumes are described. Tape config information is provided, but no performance stats are available on most systems so the stats are level 1.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
type	0 / 1.7.1	ENUMERATION unknown tape jukebox array fep	NOUNITS = 0 = 1 = 3 (optical or tape changer) = 4 (disk array / RAID controller) = 5 (mainframe front end)	target-type: one instance per-target, coded as	0	No	No
size	0 / 1.7.2	UINT4	KBYTES	target size, zero if not applicable	4	No	No
label	0 / 1.7.3	TEXTSTRING	NOUNITS	target label (c0t0, sd0, st0 etc)	8	Yes	No
major	0 / 1.7.4	UINT4	COUNT		16	No	No
minor	0 / 1.7.5	UINT4	COUNT		20	No	No
status	1 / 1.7.6	ENUMERATION offline online failed	NOUNITS = 0 = 1 = 2		24	No	No
vendor	1 / 1.7.7	TEXTSTRING	NOUNITS		28	Yes	No
vendor_designation	1 / 1.7.8	TEXTSTRING	NOUNITS		36	Yes	No
controller_inst_xref	opt / 1.7.85	TEXTSTRING	XREF	reference to a controller	0	Yes	No
controller2_inst_xref	opt / 1.7.86	TEXTSTRING	XREF	optional 2nd controller	8	Yes	No
device_cache_size	opt / 1.7.87	UINT4	KBYTES	e.g. RAID cache	16	No	No
device_queue_size	opt / 1.7.88	UINT4	COUNT	SCSI tagged command queue size	20	No	No
config_change_event	0 / 1.7.89	UINT4	NOUNITS	-	-	-	-

Table 4-7 Other Device Instance Configuration

4.3.8 Subclass — Disk Partition Instance Configuration

Also called slices, and subdisks. Each partition xrefs the disk it comes from, and is in turn xref'd by the volume hierarchy. Filesystem info is not provided here, as filesystems may span multiple disk partitions and partitions may be used raw.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
partition_label	1/1.8.1	TEXTSTRING	NOUNITS	slice/partition/subdisk label (s0, a, disk01-1)	0	Yes	No
partition_size	1/1.8.2	UINT4	KBYTES	target size, zero if not applicable	8	No	No
partition_start	1/1.8.3	UINT4	KBYTES	offset into disk of start	12	No	No
major	1/1.8.4	UINT4	COUNT		16	No	No
minor	1/1.8.5	UINT4	COUNT		20	No	No
disk_inst_xref	opt/1.8.85	TEXTSTRING	XREF	xref into a disk instance	0	Yes	No
config_change_event	0/1.8.86	UINT4	NOUNITS	-	-	-	-

Table 4-8 Disk Partition Instance Configuration

4.3.9 Subclass — Volume Group Instance Configuration

The following classes are architected to refer to the I/O hierarchy, and to allow for Veritas LVM, HP, AIX and SunSoft DiskSuite based software implementations, and hardware RAID configurations.

Volume groups, disk groups and disk sets are names for a collection of related volumes, they are used both for administrative convenience, and to provide for HA failover, and disk sharing between systems in parallel database applications. In the absence of an explicit group, volumes not in any group omit the cross-reference and there will be no volume group instances.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
volume_group_label	1/1.9.1	TEXTSTRING	NOUNITS	group name e.g. rootdg	0	Yes	No
status	1/1.9.2	ENUMERATION offline online failed	NOUNITS = 0 = 1 = 2		8	No	No
config_change_event	0/1.9.3	UINT4	NOUNITS	-	-	-	-

Table 4-9 Volume Group Instance Configuration

4.3.10 Subclass — Volume/Metadisk Instance Configuration

Volumes are the entities that have data stored in them. They can be used raw, to provide swap space or for raw database tables. They can have a filesystem built on them so the filesystem can be mounted. A volume can contain an xref to a disk group, and is xref'd by plexes. In DiskSuite terminology a volume is a metadisk.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
volume_size	1/1.10.1	UINT4	KBYTES	size of volume in Kbytes	0	No	No
volume_label	1/1.10.2	TEXTSTRING	NOUNITS	volume label e.g. vol3, md23, lv01	4	Yes	No
status	1/1.10.3	ENUMERATION offline online failed	NOUNITS = 0 = 1 = 2		12	No	No
vg_inst_xref	opt/1.10.85	TEXTSTRING	XREF	reference to disk_group	0	Yes	No
config_change_event	1/1.10.86	UINT4	NOUNITS	-	-	-	-

Table 4-10 Volume/Metadisk Instance Configuration

4.3.11 Subclass — Plex/Metapartition Instance Configuration

Plexes are the components used to tie disk partitions to volumes. Plex instances are strongly ordered, so that a volume is made up of plexes concatenated or mirrored in the order in which they appear in the plex instances. This info can be used by an application to build a doubly linked tree to find the plexes given a volume. A common configuration change is to add another plex at the end of a volume to increase its size, adding another plex at the end of the list of instances satisfies this requirement. Plexes can reference other plexes in order to allow constructs such as mirrored striped volumes. Plexes are also known as metapartitions.

For example, to encode a simple mirror of two pieces of disk, two type 0 plexes would xref the same volume, and would each xref a disk partition.

Sizes are not redundant, as mirroring or striping two plexes results in the minimum of the two as the size.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
plex_label	1/1.11.1	TEXTSTRING	NOUNITS		0	Yes	No
plex_type	1/1.11.2	ENUMERATION	NOUNITS	plex type, coded as	8	No	No
		raid0/submirror	= 0				
		raid1/stripe	= 1				
		raid2	= 2				
		raid3	= 3				
		raid4	= 4				
		raid5	= 5				
		raid4parity	= 14				
		parity disk log	= 15	journal filesystem log			
		hotspare	= 16				
plex_interlace	1/1.11.3	UINT4	KBYTES	interlace used for stripe and raid	12	No	No
plex_size	1/1.11.4	UINT4	KBYTES	size of plex	16	No	No
status	1/1.11.5	ENUMERATION	NOUNITS		20	No	No
offline	= 0						
online	= 1						
failed	= 2						
vol_pl_inst_xref	opt/1.11.85	TEXTSTRING	XREF	xref to a volume or a higher level plex	0	Yes	No
part_pl_inst_xref	opt/1.11.86	TEXTSTRING	XREF	xref to a partition or a lower level plex	8	Yes	No
config_change_event	1/1.11.87	UINT4	NOUNITS	-	-	-	-

Table 4-11 Plex / Metapartition Instance Configuration Label

4.3.12 Subclass — File System Instance Configuration

A filesystem can cross-reference a (logical) volume or a simple disk partition only. Filesystems that store data are represented here. Special filesystem types like **procfs** are not.

The data reported by the BSD form of **df** is the basis of this class

```
% df
Filesystem            kbytes   used   avail   capacity   Mounted on
/edev/edsk/ec0t3d0s0  189858 169406    1472        99%          /re
```

The event is generated when the filesystem is mounted, unmounted or a hard error state is entered. File system full is not an event condition. Applications can poll this subclass to monitor filesystem free space.

The filesystem type is a string, since there are more filesystem types than can be enumerated in the standard. Since there are many common types, these exact strings should be used if the filesystem matches one of the entries.

Label String	Type
raw	raw disk (raw database tablespace)
swap	swap space
ufs	BSD4 based UNIX file system
hsfs	cdrom High Sierra file system
pcfs	MSDOS pc
sys5	system V
jfs	journal fs
xfs	extent fs
tmpfs	RAM based temporary fs
cachefs	ONC+ caching fs
nfs_v2	NFS version 2
nfs_v3	NFS version 3
afs	Andrew FS
dfs	DCE
rfs	Sys V Remote FS

Table 4-12 File System Types Label String

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
filesystem_type	0/1.12.1	TEXTSTRING	NOUNITS		0	Yes	No
device_string	0/1.12.2	TEXTSTRING	NOUNITS	mounted device	8	Yes	No
size	0/1.12.3	UINT4	KBYTES	size in KBytes	16	No	No
mount_point	0/1.12.4	TEXTSTRING	NOUNITS	mount point path name string	20	Yes	No
status	1/1.12.5	ENUMERATION unmounted mounted failed	NOUNITS = 0 = 1 = 2		24	No	No
options	1/1.12.6	TEXTSTRING	NOUNITS	mount options string	28	Yes	No
block_size	1/1.12.7	UINT4	BYTES	filesystem block size (e.g. stat.st_blksize)	36	No	No
part_vol_inst_xref	opt/1.12.85	ID_INST	XREF	xref to a disk partition or volume	0	No	No
config_change_event	0/1.12.86	UINT4	NOUNITS	-	-	-	-

Table 4-13 File System Instance Configuration

4.3.13 Subclass — Dynamic Kernel Table Counter Configuration

The number and definition of fixed size kernel tables varies even from one release of an OS to the next. In general many systems are removing fixed size limits. This class specifies the names of the tables that are reported on by the Dynamic Kernel Table Counter class. Each instance of this class defines the name of the corresponding instance in that class.

There are four values that are commonly reported by the SVR4 **sar** -*v* option. If these tables are provided then standard strings must be used to indicate the names. Process, inode, file and lock tables are often reported by **sar**.

```
% sar -v 1
23:45:06   proc-sz   ov   inod-sz     ov   file-sz   ov   lock-sz
23:45:07   49/506    0    1874/1874   0    297/297   0    0/0
```

Level	Table name string	Type
0	process	process table
0	inode	ufs inode table
0	file	systemwide open file table
0	lock	systemwide file lock table

Table 4-14 Kernel Table Types

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
table_name	0/1.13.1	TEXTSTRING	NOUNITS		0	Yes	No
table_limit	0/1.13.2	UINT8	NOUNITS		8	No	No

Table 4-15 Dynamic Kernel Table Counter Configuration

Some tables may not have a limit in certain implementations, so return 0 as the limit.

4.3.14 Subclass — per-IP Configuration

Some systems support multiple protocols, or multiple IP addresses on one interface, so this is separated from the network interface itself. Table 16:

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
ip_address	0/1.14.1	TEXTSTRING	NOUNITS		0	Yes	No
subnet mask	0/1.14.2	TEXTSTRING	NOUNITS		8	Yes	No
broadcast	0/1.14.3	TEXTSTRING	NOUNITS		16	Yes	No
network_interface_xref	opt/1.14.85	TEXTSTRING	XREF	xref to a network interface	0	Yes	No
config_change_event	0/1.14.86	UINT4	NOUNITS	-	-	-	-

Table 4-16 Subclass — per-IP Configuration

4.3.15 Subclass — System Call Configuration

This class was designated as level 1 at one point. In fact it can be level 0, and if no data is available no instances need be defined. Each implementation is free to define its own set of system calls and to decide which calls are instrumented and reported.

Based on the data normally displayed by **sar** -*c*, certain system calls may be counted globally. The name of each instance is provided by this class. The counters are read by the System Call Counter class. Some sample **sar** -*c* output is shown below.

```
% sar -c 1

SunOS crun 5.5 Generic sun4u    07/08/96

15:50:02 scall/s sread/s swrit/s fork/s exec/s rchar/s wchar/s
15:50:03     234      19      16   0.00   0.00    3801    2614
```

The first five values are considered to have predefined name strings as shown. The last two values shown in the **sar** example are not call counts.

Table name string	Type
syscalls	total number of system calls
read	read syscall
write	write syscall
fork	fork syscall
exec	exec syscall

Table 4-17 System Call Names

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
syscall_name	0/1.15.1	TEXTSTRING	NOUNITS		0	Yes	No

Table 4-18 System Call Configuration

4.3.16 Subclass — Scheduling Class Configuration

One instance per-scheduling class. Secondary ranges exist in some System V implementations. This class is needed so that it is clear whether high numerical values refer to high or low priorities, and to indicate the relative ordering and overlap of different scheduler classes.

Data Attributes					MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Example**	**Offset**	**VLDS Object**	**Instance Array**
sched_class_name	0/1.16.1	TEXTSTRING	NOUNITS	e.g. TS	0	Yes	No
low_end_priority	0/1.16.2	UINT4	COUNT		8	No	No
high_end_priority	0/1.16.3	UINT4	COUNT		12	No	No
default_start_priority	0/1.16.4	UINT4	COUNT		16	No	No
low_end_priority_secondary	opt/1.16.85	UINT4	COUNT		0	No	No
high_end_priority_secondary	opt/1.16.86	UINT4	COUNT		4	No	No
config_change_event	0/1.16.87	UINT4	NOUNITS	-	-	-	-

Table 4-19 Scheduling Class Configuration

4.4 Processor Classes

4.4.1 Measured Per-processor Times

One instance per-processor. Measured times are accumulated at the state transitions, and are more accurate, but higher overhead than the sampled per-processor times.

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
user_time	0/2.1.1	UINT8	TIMEVAL	0	No	No
system_time	0/2.1.2	UINT8	TIMEVAL	8	No	No
idle_time	0/2.1.3	UINT8	TIMEVAL	16	No	No

Table 4-20 Measured Per-processor Times

Constraints

user_time + system_time + idle_time = 100% of the elaspsed time.

Definitions

idle_time includes all wait time. Idle time is broken down further in the per-processor wait times subclass.

4.4.2 Sampled Per-processor Times

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
user_time	0/2.2.1	UINT8	TIMEVAL	0	No	No
system_time	0/2.2.2	UINT8	TIMEVAL	8	No	No
idle_time	0/2.2.3	UINT8	TIMEVAL	16	No	No
idle_disk_wait_time	1/2.2.4	UINT8	TIMEVAL	24	No	No

Table 4-21 Sampled Per-processor Times

Constraints

user_time + system_time + idle_time = 100% of the elapsed time

idle_disk_wait_time <= idle_time

Definitions

idle_time includes all wait time.

idle_disk_wait_time is accumulated when the sample occurs while the system is waiting for a disk I/O.

The idle_disk_wait_time is inflated on multiprocessor systems, as multiple idle CPUs are all sampled as waiting for a single I/O. When reporting a composite system wide wait time, it is recommended that the total idle_disk_wait_time is divided by the number of CPUs and the excess is allocated as idle time.

4.4.3 Per-processor Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
system_calls	0/2.3.1	UINT8	COUNT	0	No	No
hardware_interrupts	0/2.3.2	UINT8	COUNT	8	No	No
total_switches	0/2.3.3	UINT8	COUNT	16	No	No
voluntary_switches	1/2.3.4	UINT8	COUNT	24	No	No
traps	1/2.3.5	UINT8	COUNT	32	No	No
program_interrupts	1/2.3.6	UINT8	COUNT	40	No	No
mutex_stalls	opt/2.3.85	UINT8	COUNT	0	No	No

Table 4-22 Per-processor Counters

Constraints

involuntary_switches = total_switches - voluntary_switches

Definitions

hardware_interrupts include external I/O device interrupts.

program_interrupts include cross calls between processors.

mutex_stalls are the number of times a processor is stalled because it could not obtain a mutual exclusion lock held by another processor.

4.4.4 Per-processor Per-system Call Counters

The system call configuration subclass defines and names each system call instance. There are two levels of instances here, per-processor and per-system call.

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
syscall_count	0/2.4.1	UINT8	COUNT	0	No	No
syscall_cpu_time	opt/2.4.85	UINT8	TIMEVAL	0	No	No

Table 4-23 Per-processor Per-system Call Counters

4.4.5 Per-work Unit Processor Times

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
user_time	0/2.5.1	UINT8	TIMEVAL	0	No	No
system_time	0/2.5.2	UINT8	TIMEVAL	8	No	No
user_child_time	0/2.5.3	UINT8	TIMEVAL	16	No	No
system_child_time	0/2.5.4	UINT8	TIMEVAL	24	No	No
rt_priority_time	opt/2.5.85	UINT8	TIMEVAL	0	No	No
interrupt_time	opt/2.5.86	UINT8	TIMEVAL	8	No	No

Table 4-24 Per Work Unit Processor Times

Definitions

rt_priority_time is time spent by this work unit running in the realtime priority class.

interrupt_time is time stolen from this work unit while processing interrupts. The interrupt time is a subset of the user_time and system_time.

user_child_time and system_child_time
only accumulate the time for child processes that have exited. They are updated when the parent process reaps the child.

4.4.6 Per-work Unit Per-system Call Counters

The system call configuration subclass defines and names each system call instance. There are two levels of instances here, per-work unit (process) and per-system call.

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
syscall_count	1/2.6.1	UINT8	COUNT	0	Yes	Yes
syscall_cpu_time	opt/2.6.85	UINT8	TIMEVAL	0	Yes	Yes

Table 4-25 Per-work Unit Per-system Call Counters

4.4.7 Wait Times

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
idle_disk_wait_time	1/2.7.1	UINT8	TIMEVAL	0	No	No
idle_page_swap_time	1/2.7.2	UINT8	TIMEVAL	8	No	No

Table 4-26 Wait Times Subclass

Constraints

user_time + system_time + idle_time = 100% of the accumulated processor time

idle_page_swap_time <= idle_disk_wait_time <= idle_time

Definitions

idle_time includes all wait time.

idle_disk_wait_time is accumulated when the CPU sleeps waiting for a disk I/O.

idle_page_swap_time is accumulated when the CPU sleeps waiting for a disk I/O that is paging or swapping to the swap space. Unlike sampled per-processor times, the idle_disk_wait_time will not be inflated on multiprocessor systems, as only the CPU that initiated the I/O will be measured. The wait time ends when the I/O ends, even if a different CPU processes the completion.

4.5 Memory Class

4.5.1 Global Physical Memory Usage

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
free_memory	0/3.1.1	UINT4	KBYTES	0	No	No
file_cache_memory	0/3.1.2	UINT4	KBYTES	4	No	No
kernel_memory	0/3.1.3	UINT4	KBYTES	8	No	No
other_memory	0/3.1.4	UINT4	KBYTES	12	No	No
wired_memory	1/3.1.5	UINT4	KBYTES	16	No	No
shared memory	1/3.1.6	UINT4	KBYTES	20	No	No
dirty_memory	opt/3.1.85	UINT4	KBYTES	0	No	No

Table 4-27 Global Physical Memory Usage

Constraints

total physical memory = free_memory + file_cache_memory + kernel_memory + other_memory

Definitions

total physical memory (physmem)
: is in the system configuration class

wired_memory
: is memory that cannot be paged out.

shared memory
: is all memory that is multiply referenced.

dirty_memory
: is memory that is modified with respect to its backing store.

4.5.2 Global Virtual Memory Usage

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
total_swap	0/3.2.1	UINT4	PAGES	0	No	No
swap_available	0/3.2.2	UINT4	PAGES	4	No	No
swap_allocated	0/3.2.3	UINT4	PAGES	8	No	No

Table 4-28 Global Virtual Memory Usage

The intention is to know how much virtual memory is in use, and how much is left, such that when there is no swap available it is not possible to grow or start processes. These are the four underlying measures provided by SVR4 and Solaris 2 (for example):

swap_reserved
: reserved swap in pages — space reserved but not written to.

swap_allocated
: allocated swap in pages — space that has been written to.

swap_available
: unreserved swap in pages — space available to be reserved.

swap_free
: unallocated swap in pages — space that has yet to be written to.

The metrics are derived from these measures as follows:

total_swap = swap_allocated + swap_reserved + swap_available

Note that swap_free is not used in this calculation, although it is the value reported by **sar** on some systems. swap_available is the value reported by **vmstat**.

4.5.3 Per-processor Demand Paging Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
major_faults	0/3.3.1	UINT8	COUNT	0	No	No
minor_faults	0/3.3.2	UINT8	COUNT	8	No	No
pages_in	0/3.3.3	UINT8	PAGES	16	No	No
pages_out	0/3.3.4	UINT8	PAGES	24	No	No
page_reclaims	0/3.3.5	UINT8	COUNT	32	No	No
page_in_ops	1/3.3.6	UINT8	COUNT	40	No	No
page_out_ops	1/3.3.7	UINT8	COUNT	48	No	No
zero_fill_pages	1/3.3.8	UINT8	PAGES	56	No	No
copy_on_write_faults	1/3.3.9	UINT8	COUNT	64	No	No
copy_on_write_pages	1/3.3.10	UINT8	PAGES	72	No	No
pages_scanned	1/3.3.11	UINT8	PAGES	80	No	No
pager_run	1/3.3.12	UINT8	COUNT	88	No	No
pages_freed	1/3.3.13	UINT8	PAGES	96	No	No
pages_attached	1/3.3.14	UINT8	PAGES	104	No	No
ssq_pages_in	opt/3.3.85	UINT8	PAGES	0	No	No
ssq_pages_out	opt/3.3.86	UINT8	PAGES	8	No	No

Table 4-29 Per-processor Demand Paging Counters

4.5.4 Per-processor Swapping Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
pages_swapped_in	0/3.4.1	UINT8	PAGES	0	No	No
pages_swapped_out	0/3.4.2	UINT8	PAGES	8	No	No
processes_swapped_in	0/3.4.3	UINT8	PROCESSES	16	No	No
processes_swapped_out	0/3.4.4	UINT8	PROCESSES	24	No	No

Table 4-30 Per-processor Swapping Counters

4.5.5 Per-work Unit Memory Usage

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
total_virtual_memory_size	0/3.5.1	UINT8	PAGES	0	No	No
total_resident_set_size	0/3.5.2	UINT8	PAGES	8	No	No
private_resident_memory	1/3.5.3	UINT8	PAGES	16	No	No
shared_resident_memory	1/3.5.4	UINT8	PAGES	24	No	No
wired_memory	1/3.5.5	UINT8	PAGES	32	No	No
sys5_shared_memory	1/3.5.6	UINT8	PAGES	40	No	No

Table 4-31 Per-work Unit Memory Usage

The level 0 metrics are basically the SIZE and RSS fields reported by the UNIX **ps** command for each process.

4.5.6 Per-work Unit Demand Paging Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
minor_faults	0/3.6.1	UINT8	COUNT	0	No	No
major_faults	0/3.6.2	UINT8	COUNT	8	No	No
child_minor_faults	1/3.6.3	UINT8	COUNT	16	No	No
child_major_faults	1/3.6.4	UINT8	COUNT	24	No	No

Table 4-32 Per-work Unit Demand Paging Counters

Definitions

Minor faults are resolved within the memory system, typically without sleeping.

Major faults require a disk I/O to resolve, causing the work unit to sleep until the I/O completes.

Child fault counts are accumulated for children that have exited at the time the children are reaped.

4.5.7 Per-work Unit Swapping Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
swap_outs	0/3.7.1	UINT8	COUNT	0	No	No

Table 4-33 Per-work Unit Swapping Counters

4.5.8 Dynamic Kernel Table Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
current_inuse	0/3.8.1	UINT8	COUNT	0	No	No
allocated_size	0/3.8.2	UINT8	COUNT	8	No	No
maximum_reached	1/3.8.3	UINT8	COUNT	16	No	No

Table 4-34 Dynamic Kernel Table Counters

Notes

Process, inode, file and lock tables and limits are listed in the configuration section for level 0.

Allocated size may be the same as the limit if the whole table is preallocated, may be the same as current_inuse if the table is allocated one item at a time, or may be a little larger than current_inuse if allocations are made in slabs. Some implementations that use a dynamically allocated linked list of structures do not have a hard limit, but when exceeded the system will reclaim inactive entries immediately. Each instance of this subclass reports on a different kernel table.

4.5.9 Memory Object Subclass

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
memory_object_type	opt/3.9.85	ENUMERATION sys5_shared_memory mmap_shared_memory file	NOUNITS = 0 = 1 = 2	0	No	No
mount_point	opt/3.9.86	TEXTSTRING	NOUNITS	8	Yes	No
inode	opt/3.9.87	UINT8	NOUNITS	16	No	No
resident_memory_size	opt/3.9.88	UINT8	PAGES	24	No	No
virtual_memory_size	opt/3.9.89	UINT8	PAGES	32	No	No
locked_memory_size	opt/3.9.90	UINT8	PAGES	40	No	No

Table 4-35 Memory Object Subclass

4.6 IPC Class

4.6.1 IPC subclass

These counters are intended to map onto the common ‘‘System V’’ IPC message queue and semaphore implementation data (returned by **sar** *-m* on some platforms).

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
ipc_message_sent	0/4.1.1	UINT8	COUNT	0	No	No
ipc_message_received	0/4.1.2	UINT8	COUNT	8	No	No
semaphore_operations	0/4.1.3	UINT8	COUNT	16	No	No

Table 4-36 Global IPC Counters subclass

4.7 Scheduling Class

4.7.1 Global Runqueue Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
runq_samples	0/5.1.1	UINT8	COUNT	0	No	No
runload_sum	0/5.1.2	UINT8	PROCESSES	8	No	No
runnonload_sum	0/5.1.3	UINT8	PROCESSES	16	No	No

Table 4-37 Global Runqueue Counters

4.7.2 Per-work Unit Scheduling Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
sched_class_id	0/5.2.1	UINT4	NOUNITS	0	No	No
global_priority	0/5.2.2	INT4	COUNT	4	No	No
nice_value	0/5.2.3	INT4	COUNT	8	No	No

Table 4-38 Per-work Unit Scheduling Counters

4.8 Disk Device Data Class

4.8.1 Global Physical I/O Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
buffer_reads	0/6.1.1	UINT8	COUNT	0	No	No
logical_reads	0/6.1.2	UINT8	COUNT	8	No	No
buffer_writes	0/6.1.3	UINT8	COUNT	16	No	No
logical_writes	0/6.1.4	UINT8	COUNT	24	No	No
physical_reads	0/6.1.5	UINT8	COUNT	32	No	No
physical_writes	0/6.1.6	UINT8	COUNT	40	No	No
buffer_reads_KB	1/6.1.7	UINT8	KBYTES	48	No	No
logical_reads_KB	1/6.1.8	UINT8	KBYTES	56	No	No
buffer_writes_KB	1/6.1.9	UINT8	KBYTES	64	No	No
logical_writes_KB	1/6.1.10	UINT8	KBYTES	72	No	No
physical_read_KB	1/6.1.11	UINT8	KBYTES	80	No	No
physical_write_KB	1/6.1.12	UINT8	KBYTES	88	No	No

Table 4-39 Global Physical I/O Counters

This class is based on the data commonly provided by **sar** – **b**:

```
% sar -b
10:44:11 bread/s lread/s %rcache bwrit/s lwrit/s %wcache pread/s pwrit/s
```

Notes

The filesystem buffer cache measures are commonlly provided but there are widely different implementations so the measures are not always useful. Some (traditional UNIX) implementations use the buffer cache for all filesystem I/O. Others (e.g. UNIX SVR4, Solaris 1, and Solaris 2) do not, and only use it to store filesystem metadata such as inodes, indirect blocks and cylinder group blocks.

The physical I/O counters are the counts of raw device accesses rather than block device (filesystem) accesses. These are commonly reported by **sar** as *pread* and *pwrite*.

4.8.2 Per-disk Device Data

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
read_bytes	0/6.2.1	UINT8	BYTES	0	No	No
write_bytes	0/6.2.2	UINT8	BYTES	8	No	No
reads	0/6.2.3	UINT8	COUNT	16	No	No
writes	0/6.2.4	UINT8	COUNT	24	No	No
wait_time	0/6.2.5	UINT8	TIMEVAL	32	No	No
wait_length_time	0/6.2.6	UINT8	TIMEVAL	40	No	No
active_time	0/6.2.7	UINT8	TIMEVAL	48	No	No
active_length_time	0/6.2.8	UINT8	TIMEVAL	56	No	No

Table 4-40 Per-disk Device Data

A disk device requires two queues to be instrumented. The wait queue consists of commands that have not yet been issued to the disk. The active queue consists of commands that have been issued to the disk but have not yet completed. Modern SCSI disks can accept a large active queue.

Definitions

wait_time — is a running sum of the time that the queue is non-empty.

wait_length_time — is a running sum of the product of queue length and elapsed time at that length.

The active queue is instrumented the same way.

active_length_time — is a running sum of the product of queue length and elapsed time at that length.

Notes

At each entry or exit from the queue, the elapsed time since the previous state change is added to the wait_time if the queue length is non-zero, and the product of the time and the queue length is added to wait_length_time.

In the following example, assume that two measurements (new and old) were separated by some elapsed_time. The higher level disk statistics are obtained by the following calculations:

- read_KB_per_sec = (new.read_bytes - old.read_bytes) / (1024 * elapsed_time)
- writes_per_sec = (new.writes - old.writes) / elapsed_time
- utilisation = busy_percent = 100 * (new.active_time - old.active_time) / elapsed_time
- wait_queue_length = (new.wait_length_time - old.wait_length_time) / elapsed_time
- active_queue_length = (new.active_length_time - old.active_length_time) / elapsed_time
- wait_time_milliseconds = 1000 * wait_queue_length / (reads_per_sec + writes_per_sec)
- service_time_milliseconds = 1000 * active_queue_length / (reads_per_sec + writes_per_sec)
- response_time_milliseconds = wait_time_milliseconds + service_time_milliseconds.

4.8.3 Per-work Unit I/O

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
inblock	0/6.3.1	UINT8	BYTES	0	No	No
outblock	0/6.3.2	UINT8	BYTES	8	No	No

Table 4-41 Per-work Unit I/O

Notes

These measures count the filesystem block I/O made by the work unit. Care must be taken to convert the values into bytes from whatever the operating system reports them in (may be 512 byte blocks).

4.9 Global File Systems Class

4.9.1 Global File Service Counters

This subclass accumulates all activity related to NFSV2, NFSV3, DFS, AFS, etc.

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
inbound_reqs	1/7.1.1	UINT8	COUNT	0	No	No
read_reqs	1/7.1.2	UINT8	COUNT	8	No	No
send_bytes	1/7.1.3	UINT8	BYTES	16	No	No
write_reqs	1/7.1.4	UINT8	COUNT	24	No	No
recv_bytes	1/7.1.5	UINT8	BYTES	32	No	No
duplicate_requests	1/7.1.6	UINT8	COUNT	40	No	No

Table 4-42 Global File Service Counters

4.9.2 ONC RPC Client Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
client_rpc_calls	0/7.2.1	UINT4	COUNT	0	No	No
client_rpc_badcalls	0/7.2.2	UINT4	COUNT	4	No	No
client_rpc_retransmits	0/7.2.3	UINT4	COUNT	8	No	No
client_rpc_badxids	0/7.2.4	UINT4	COUNT	12	No	No
client_rpc_waits	0/7.2.5	UINT4	COUNT	16	No	No
client_rpc_newcreds	0/7.2.6	UINT4	COUNT	20	No	No

Table 4-43 ONC RPC Client Counters

4.9.3 ONC NFS Version 2 Client Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
client_nfs_sleepcalls	0/7.3.1	UINT4	COUNT	0	No	No
client_nfs_calls	0/7.3.2	UINT4	COUNT	4	No	No
client_nfs_gets	0/7.3.3	UINT4	COUNT	8	No	No
client_nfs_badcalls	0/7.3.4	UINT4	COUNT	12	No	No
client_nfs_nullrcvs	0/7.3.5	UINT4	COUNT	16	No	No
client_nfs_getattrs	0/7.3.6	UINT4	COUNT	20	No	No
client_nfs_setattrs	0/7.3.7	UINT4	COUNT	24	No	No
client_nfs_root	0/7.3.8	UINT4	COUNT	28	No	No
client_nfs_lookup	0/7.3.9	UINT4	COUNT	32	No	No
client_nfs_readlink	0/7.3.10	UINT4	COUNT	36	No	No
client_nfs_read	0/7.3.11	UINT4	COUNT	40	No	No
client_nfs_writecache	0/7.3.12	UINT4	COUNT	44	No	No
client_nfs_write	0/7.3.13	UINT4	COUNT	48	No	No
client_nfs_create	0/7.3.14	UINT4	COUNT	52	No	No
client_nfs_remove	0/7.3.15	UINT4	COUNT	56	No	No
client_nfs_rename	0/7.3.16	UINT4	COUNT	60	No	No
client_nfs_link	0/7.3.17	UINT4	COUNT	64	No	No
client_nfs_symlink	0/7.3.18	UINT4	COUNT	68	No	No
client_nfs_mkdir	0/7.3.19	UINT4	COUNT	72	No	No
client_nfs_rmdir	0/7.3.20	UINT4	COUNT	76	No	No
client_nfs_readdir	0/7.3.21	UINT4	COUNT	80	No	No
client_nfs_filesysstat	0/7.3.22	UINT4	COUNT	84	No	No

Table 4-44 ONC NFS Version 2 Client Counters

4.9.4 ONC RPC Server Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
server_rpc_calls	0/7.4.1	UINT4	COUNT	0	No	No
server_rpc_badcalls	0/7.4.2	UINT4	COUNT	4	No	No
server_rpc_nullrcvs	0/7.4.3	UINT4	COUNT	8	No	No
server_rpc_badlens	0/7.4.4	UINT4	COUNT	12	No	No
server_rpc_xdrcalls	0/7.4.5	UINT4	COUNT	16	No	No

Table 4-45 ONC RPC Server Counters

4.9.5 ONC NFS Version 2 Server Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
server_nfs_calls	0/7.5.1	UINT4	COUNT	0	No	No
server_nfs_badcalls	0/7.5.2	UINT4	COUNT	4	No	No
server_nfs_null	0/7.5.3	UINT4	COUNT	8	No	No
server_nfs_getattr	0/7.5.4	UINT4	COUNT	12	No	No
server_nfs_setattr	0/7.5.5	UINT4	COUNT	16	No	No
server_nfs_root	0/7.5.6	UINT4	COUNT	20	No	No
server_nfs_lookups	0/7.5.7	UINT4	COUNT	24	No	No
server_nfs_readlink	0/7.5.8	UINT4	COUNT	28	No	No
server_nfs_reads	0/7.5.9	UINT4	COUNT	32	No	No
server_nfs_writecache	0/7.5.10	UINT4	COUNT	36	No	No
server_nfs_writes	0/7.5.11	UINT4	COUNT	40	No	No
server_nfs_creates	0/7.5.12	UINT4	COUNT	44	No	No
server_nfs_removes	0/7.5.13	UINT4	COUNT	48	No	No
server_nfs_renames	0/7.5.14	UINT4	COUNT	52	No	No
server_nfs_links	0/7.5.15	UINT4	COUNT	56	No	No
server_nfs_symlinks	0/7.5.16	UINT4	COUNT	60	No	No
server_nfs_mkdir	0/7.5.17	UINT4	COUNT	64	No	No
server_nfs_rmdir	0/7.5.18	UINT4	COUNT	68	No	No
server_nfs_readdir	0/7.5.19	UINT4	COUNT	72	No	No
server_nfs_filesysstat	0/7.5.20	UINT4	COUNT	76	No	No

Table 4-46 ONC NFS Version 2 Server Counters

4.9.6 ONC NFS Version 3 Client Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
client_nfs_sleepcalls	0/7.6.1	UINT4	COUNT	0	No	No
client_nfs_calls	0/7.6.2	UINT4	COUNT	4	No	No
client_nfs_gets	0/7.6.3	UINT4	COUNT	8	No	No
client_nfs_badcalls	0/7.6.4	UINT4	COUNT	12	No	No
client_nfs3_nullrcvs	0/7.6.5	UINT4	COUNT	16	No	No
client_nfs3_getattrs	0/7.6.6	UINT4	COUNT	20	No	No
client_nfs3_setattrs	0/7.6.7	UINT4	COUNT	24	No	No
client_nfs3_lookup	0/7.6.8	UINT4	COUNT	28	No	No
client_nfs3_access	0/7.6.9	UINT4	COUNT	32	No	No
client_nfs3_readlink	0/7.6.10	UINT4	COUNT	36	No	No
client_nfs3_read	0/7.6.11	UINT4	COUNT	40	No	No
client_nfs3_write	0/7.6.12	UINT4	COUNT	44	No	No
client_nfs3_create	0/7.6.13	UINT4	COUNT	48	No	No
client_nfs3_mkdir	0/7.6.14	UINT4	COUNT	52	No	No
client_nfs3_symlink	0/7.6.15	UINT4	COUNT	56	No	No
client_nfs3_mknod	0/7.6.16	UINT4	COUNT	60	No	No
client_nfs3_remove	0/7.6.17	UINT4	COUNT	64	No	No
client_nfs3_rmdir	0/7.6.18	UINT4	COUNT	68	No	No
client_nfs3_rename	0/7.6.19	UINT4	COUNT	72	No	No
client_nfs3_link	0/7.6.20	UINT4	COUNT	76	No	No
client_nfs3_readdir	0/7.6.21	UINT4	COUNT	80	No	No
client_nfs3_readdirplus	0/7.6.22	UINT4	COUNT	84	No	No
client_nfs3_filesysstat	0/7.6.23	UINT4	COUNT	88	No	No
client_nfs3_filesysinfo	0/7.6.24	UINT4	COUNT	92	No	No
client_nfs3_pathconf	0/7.6.25	UINT4	COUNT	96	No	No
client_nfs3_commit	0/7.6.26	UINT4	COUNT	100	No	No

Table 4-47 ONC NFS Version 3 Client Counters

4.9.7 ONC NFS Version 3 Server Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
server_nfs_sleepcalls	0/7.7.1	UINT4	COUNT	0	No	No
server_nfs_calls	0/7.7.2	UINT4	COUNT	4	No	No
server_nfs_gets	0/7.7.3	UINT4	COUNT	8	No	No
server_nfs_badcalls	0/7.7.4	UINT4	COUNT	12	No	No
server_nfs3_nullrcvs	0/7.7.5	UINT4	COUNT	16	No	No
server_nfs3_getattrs	0/7.7.6	UINT4	COUNT	20	No	No
server_nfs3_setattrs	0/7.7.7	UINT4	COUNT	24	No	No
server_nfs3_lookup	0/7.7.8	UINT4	COUNT	28	No	No
server_nfs3_access	0/7.7.9	UINT4	COUNT	32	No	No
server_nfs3_readlink	0/7.7.10	UINT4	COUNT	36	No	No
server_nfs3_read	0/7.7.11	UINT4	COUNT	40	No	No
server_nfs3_write	0/7.7.12	UINT4	COUNT	44	No	No
server_nfs3_create	0/7.7.13	UINT4	COUNT	48	No	No
server_nfs3_mkdir	0/7.7.14	UINT4	COUNT	52	No	No
server_nfs3_symlink	0/7.7.15	UINT4	COUNT	56	No	No
server_nfs3_mknod	0/7.7.16	UINT4	COUNT	60	No	No
server_nfs3_remove	0/7.7.17	UINT4	COUNT	64	No	No
server_nfs3_rmdir	0/7.7.18	UINT4	COUNT	68	No	No
server_nfs3_rename	0/7.7.19	UINT4	COUNT	72	No	No
server_nfs3_link	0/7.7.20	UINT4	COUNT	76	No	No
server_nfs3_readdir	0/7.7.21	UINT4	COUNT	80	No	No
server_nfs3_readdirplus	0/7.7.22	UINT4	COUNT	84	No	No
server_nfs3_filesysstat	0/7.7.23	UINT4	COUNT	88	No	No
server_nfs3_filesysinfo	0/7.7.24	UINT4	COUNT	92	No	No
server_nfs3_pathconf	0/7.7.25	UINT4	COUNT	96	No	No
server_nfs3_commit	0/7.7.26	UINT4	COUNT	100	No	No

Table 4-48 ONC NFS Version 3 Server Counters

4.10 Network Protocol Class

These subclasses are based entirely on the pre-existing standard SNMP MIB classes for network protocol monitoring. They are provided as part of the UMA datapool definition to allow the data to be stored alongside other data in a synchronized and standardized format. The metrics defined here are exactly those defined by the SNMP MIBs.

4.10.1 Per-network Interface Statistics

The per-interface data instances match the network device configuration instances.

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
per_interface_packets_in	0/8.1.1	UINT8	COUNT	0	No	No
per_interface_packets_out	0/8.1.2	UINT8	COUNT	8	No	No
per_interface_collisions	0/8.1.3	UINT8	COUNT	16	No	No
per_interface_kbytes_in	1/8.1.4	UINT8	COUNT	24	No	No
per_interface_kbytes_out	1/8.1.5	UINT8	COUNT	32	No	No

Table 4-49 Per-network Interface Statistics

4.10.2 IP Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
ip_total	1/8.2.1	UINT4	COUNT	0	No	No
ip_checksums	1/8.2.2	UINT4	COUNT	4	No	No
ip_tooshort	1/8.2.3	UINT4	COUNT	8	No	No
ip_toosmall	1/8.2.4	UINT4	COUNT	12	No	No
ip_badhlen	1/8.2.5	UINT4	COUNT	16	No	No
ip_badlen	1/8.2.6	UINT4	COUNT	20	No	No
ip_fragment	1/8.2.7	UINT4	COUNT	24	No	No
ip_fragdropped	1/8.2.8	UINT4	COUNT	28	No	No
ip_fragtimeout	1/8.2.9	UINT4	COUNT	32	No	No
ip_forward	1/8.2.10	UINT4	COUNT	36	No	No
ip_cantforward	1/8.2.11	UINT4	COUNT	40	No	No
ip_redirectsent	1/8.2.12	UINT4	COUNT	44	No	No
ip_unsupprot	1/8.2.13	UINT4	COUNT	48	No	No
ip_delivered	1/8.2.14	UINT4	COUNT	52	No	No
ip_localoutput	1/8.2.15	UINT4	COUNT	56	No	No
ip_odropped	1/8.2.16	UINT4	COUNT	60	No	No
ip_reassembled	1/8.2.17	UINT4	COUNT	64	No	No
ip_fragmented	1/8.2.18	UINT4	COUNT	68	No	No
ip_ofragments	1/8.2.19	UINT4	COUNT	72	No	No
ip_cantfrag	1/8.2.20	UINT4	COUNT	76	No	No
ip_badoptions	1/8.2.21	UINT4	COUNT	80	No	No
ip_noroute	1/8.2.22	UINT4	COUNT	84	No	No

Table 4-50 IP Counters

4.10.3 TCP Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
tcp_sndtotal	1/8.3.1	UINT4	COUNT	0	No	No
tcp_totaldata	1/8.3.2	UINT4	COUNT	4	No	No
tcp_datsbytesent	1/8.3.3	UINT4	COUNT	8	No	No
tcp_sendrecdatapks_retransmitted	1/8.3.4	UINT4	COUNT	12	No	No
tcp_sendrecbytes_retransmitted	1/8.3.5	UINT4	COUNT	16	No	No
tcp_total_rec_packets	1/8.3.6	UINT4	COUNT	20	No	No
tcp_total_rec_packets_inseq	1/8.3.7	UINT4	COUNT	24	No	No
tcp_total_rec_bytes_inseq	1/8.3.8	UINT4	COUNT	28	No	No
tcp_checksumerrors	1/8.3.9	UINT4	COUNT	32	No	No
tcp_num_packet_badoffsets	1/8.3.10	UINT4	COUNT	36	No	No
tcptcp_tooshort	1/8.3.11	UINT4	COUNT	40	No	No
tcp_dup_packets_received	1/8.3.12	UINT4	COUNT	44	No	No
tcp_dup_bytes_received	1/8.3.13	UINT4	COUNT	48	No	No
tcp_connection_initiated	1/8.3.14	UINT4	COUNT	52	No	No
tcp_connection_accepts	1/8.3.15	UINT4	COUNT	56	No	No
tcp_connections_dropped	1/8.3.16	UINT4	COUNT	60	No	No
tcp_connection_closed	1/8.3.17	UINT4	COUNT	64	No	No
tcp_connections_estabilish_and_dropped	1/8.3.18	UINT4	COUNT	68	No	No
tcp_closed	1/8.3.19	UINT4	COUNT	72	No	No
tcp_segstimed	1/8.3.20	UINT4	COUNT	76	No	No
tcp_rttupdated	1/8.3.21	UINT4	COUNT	80	No	No
tcp_timeoutdropped	1/8.3.22	UINT4	COUNT	84	No	No
tcp_retrans_timeout	1/8.3.23	UINT4	COUNT	88	No	No
tcp_keepalive_timeout	1/8.3.24	UINT4	COUNT	92	No	No
tcp_presisit_timeout	1/8.3.25	UINT4	COUNT	96	No	No
tcp_send_ack	1/8.3.26	UINT4	COUNT	100	No	No
tcp_send_probe	1/8.3.27	UINT4	COUNT	104	No	No
tcp_send_update_packet	1/8.3.28	UINT4	COUNT	108	No	No
tcp_send_window_update	1/8.3.29	UINT4	COUNT	112	No	No
tcp_send_control	1/8.3.30	UINT4	COUNT	116	No	No
tcp_rec_partial_dup_packet	1/8.3.31	UINT4	COUNT	120	No	No
tcp_rec_partial_dup_bytes	1/8.3.32	UINT4	COUNT	124	No	No
tcp_rec_out_order_packet	1/8.3.33	UINT4	COUNT	128	No	No
tcp_rec_out_order_bytes	1/8.3.34	UINT4	COUNT	132	No	No
tcp_rec_packet_closed_window	1/8.3.35	UINT4	COUNT	136	No	No
tcp_rec_bytes_closed_window	1/8.3.36	UINT4	COUNT	140	No	No
tcp_rec_after_closed	1/8.3.37	UINT4	COUNT	144	No	No
tcp_rec_windown_probe	1/8.3.38	UINT4	COUNT	148	No	No
tcp_rec_dupilicate_packet	1/8.3.39	UINT4	COUNT	152	No	No
tcp_rec_dupilicate_packet	1/8.3.40	UINT4	COUNT	156	No	No
tcp_rec_ack_unsent	1/8.3.41	UINT4	COUNT	160	No	No
tcp_rec_ack_packets	1/8.3.42	UINT4	COUNT	164	No	No
tcp_rec_ack_bytes	1/8.3.43	UINT4	COUNT	168	No	No
tcp_rec_window_updates	1/8.3.44	UINT4	COUNT	172	No	No
tcp_pcbcachemiss	1/8.3.45	UINT4	COUNT	176	No	No
tcp_predict_header_date	1/8.3.46	UINT4	COUNT	180	No	No
tcp_predict_header_ack	1/8.3.47	UINT4	COUNT	184	No	No
tcp_rec_dupilicate_packet	1/8.3.48	UINT4	COUNT	188	No	No
tcp_segments_dropped_paws	1/8.3.49	UINT4	COUNT	192	No	No

Table 4-51 TCP Counters

4.10.4 UDP Subclass

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
udp_ipacket	1/8.4.1	UINT4	COUNT	0	No	No
udp_hdrops	1/8.4.2	UINT4	COUNT	4	No	No
udp_badlength	1/8.4.3	UINT4	COUNT	8	No	No
udp_checksum	1/8.4.4	UINT4	COUNT	12	No	No
udp_noport	1/8.4.5	UINT4	COUNT	16	No	No
udp_noport_broadcast	1/8.4.6	UINT4	COUNT	20	No	No
udp_fullsocket	1/8.4.7	UINT4	COUNT	24	No	No
udp_opackets	1/8.4.8	UINT4	COUNT	28	No	No
udp_pcbcachemiss	1/8.4.9	UINT4	COUNT	32	No	No

Table 4-52 UDP Subclass

4.10.5 ICMP Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
icmp_errors	opt/8.5.1	UINT4	COUNT	0	No	No
icmp_badcode	opt/8.5.2	UINT4	COUNT	4	No	No
icmp_tooshort	opt/8.5.3	UINT4	COUNT	8	No	No
icmp_checksum	opt/8.5.4	UINT4	COUNT	12	No	No
icmp_badlen	opt/8.5.5	UINT4	COUNT	16	No	No
icmp_reflect	opt/8.5.6	UINT4	COUNT	20	No	No

Table 4-53 ICMP Counters

4.10.6 ICMP Histogram Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
icmphist_output_echoreply	opt/8.6.1	UINT4	COUNT	0	No	No
icmphist_output_destunreach	opt/8.6.2	UINT4	COUNT	4	No	No
icmphist_output_outofseq	opt/8.6.3	UINT4	COUNT	8	No	No
icmphist_output_redirect	opt/8.6.4	UINT4	COUNT	12	No	No
icmphist_output_echorequest	opt/8.6.5	UINT4	COUNT	16	No	No
icmphist_output_timelimite	opt/8.6.6	UINT4	COUNT	20	No	No
icmphist_output_parameterprob	opt/8.6.7	UINT4	COUNT	24	No	No
icmphist_output_timesrequest	opt/8.6.8	UINT4	COUNT	28	No	No
icmphist_output_timesrequest	opt/8.6.9	UINT4	COUNT	32	No	No
icmphist_output_addressmaskreq	opt/8.6.10	UINT4	COUNT	36	No	No
icmphist_output_addressmaskreply	opt/8.6.11	UINT4	COUNT	40	No	No
icmphist_input_echoreply	opt/8.6.12	UINT4	COUNT	44	No	No
icmphist_input_destunreach	opt/8.6.13	UINT4	COUNT	48	No	No
icmphist_input_outofseq	opt/8.6.14	UINT4	COUNT	52	No	No
icmphist_input_redirect	opt/8.6.15	UINT4	COUNT	56	No	No
icmphist_input_echorequest	opt/8.6.16	UINT4	COUNT	60	No	No
icmphist_input_timelimite	opt/8.6.17	UINT4	COUNT	64	No	No
icmphist_input_parameterprob	opt/8.6.18	UINT4	COUNT	68	No	No
icmphist_input_timesrequest	opt/8.6.19	UINT4	COUNT	72	No	No
icmphist_input_timesreply	opt/8.6.20	UINT4	COUNT	76	No	No
icmphist_input_addressmaskreq	opt/8.6.21	UINT4	COUNT	80	No	No
icmphist_input_addressmaskreply	opt/8.6.22	UINT4	COUNT	84	No	No

Table 4-54 ICMP Histogram Counters

4.10.7 IGMP Counters

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
igmp_rcv_total	opt/8.7.1	UINT4	COUNT	0	No	No
igmp_rcv_tooshort	opt/8.7.2	UINT4	COUNT	4	No	No
igmp_rcv_checksum	opt/8.7.3	UINT4	COUNT	8	No	No
igmp_rcv_queries	opt/8.7.4	UINT4	COUNT	12	No	No
igmp_rcv_badqueries	opt/8.7.5	UINT4	COUNT	16	No	No
igmp_rcv_reports	opt/8.7.6	UINT4	COUNT	20	No	No
igmp_rcv_badreports	opt/8.7.7	UINT4	COUNT	24	No	No
igmp_rcv_ourreports	opt/8.7.8	UINT4	COUNT	28	No	No
igmp_snd_reports	opt/8.7.9	UINT4	COUNT	32	No	No

Table 4-55 IGMP Counters

4.11 Accounting

4.11.1 Per-work Unit Termination Record

Data Attributes				MLI Attributes		
Label	**Level /DatumId**	**Data Type**	**Units**	**Offset**	**VLDS Object**	**Instance Array**
accounting_flag	0/9.1.1	UINT4	NOUNITS	0	No	No
exit_status	0/9.1.2	UINT4	NOUNITS	4	No	No
user_id	0/9.1.3	UINT4	NOUNITS	8	No	No
group_id	0/9.1.4	UNIT4	NOUNITS	12	No	No
tty_name	0/9.1.5	TEXTSTRING	NOUNITS	16	Yes	No
beginning_time	0/9.1.6	UINT8	TIMESTAMP	24	No	No
usr_time	0/9.1.7	UINT8	TIMEVAL	32	No	No
system_time	0/9.1.8	UINT8	TIMEVAL	40	No	No
elapsed_time	0/9.1.9	UINT8	TIMEVAL	48	No	No
memory	0/9.1.10	UINT8	BYTESECS	56	No	No
character_rw	0/9.1.11	UINT8	BYTES	64	No	No
block_rw	0/9.1.12	UINT8	BYTES	72	No	No
command	0/9.1.13	TEXTSTRING	NOUNITS	80	Yes	No

Table 4-56 Per-work Unit Termination Record

This maps directly to the common System V accounting structure. That structure uses a special compact datatype that must be expanded before reporting these values.

Definitions

accounting_flags

Flag	Value	Comment
AFORK	0001	has executed fork, but no exec
ASU	0002	used super-user privileges
ACOMPAT	0004	used compatibility mode
ACORE	0010	dumped core
AXSIG	0020	killed by a signal
AEXPND	0040	expanded acct structure
ACCTF	0300	record type: 00 = acct

exit_status returned by the work unit.

user_id for accounting purposes.

group_id for accounting purposes.

tty_name control typewriter name.

beginning_time when the work unit started.

user_time user time charged.

system_time system time charged.

elapsed_time elapsed time duration for the work unit.

memory memory usage in bytes*seconds units. The RSS value in pages is accumulated on each clock tick that the work unit was running. This should be converted from page*tick units to byte*seconds units.

character_rw	data transferred by read and write calls via character devices.
block_rw	data read or written via block devices, that is, local filesystem.
command	name.

Glossary

adjusted
In the context of this DPD specification, this term means that any shared resource size is divided by the number of processes sharing it.

API
Application Program Interface. A standard interface for program access to set of services. The API for UMA is the Measurement Level Interface (or MLI).

ASN.1/BER
Abstract Syntax Notation One / Basic Encoding Rules. The ASN.1 language describes all abstract syntaxes in the OSI architecture. An abstract syntax is a named group of types. BER, the Basic Encoding Rules, is a *transfer syntax* used to communicate data between open systems. It includes those aspects of the rules used in the formal specification of data which embody a specific representation of that data. BER is capable of encoding any abstract syntax that can be described using ASN.1.

Collection Interval
The time between successive captures of a specific set of data items. Sometimes the term ''interval'' is used to mean the data for a collection interval having a certain time stamp and duration.

CONS
Connection Oriented Network Service.

cpu
''central processor unit'': a set of one or more computational engines on which a system runs to provide computing services to applications. A cpu can be a single processor or a multiprocessor device.

DASD
Direct Access Storage Device (for example, disk).

DCI
Data Capture Interface. A standard UMA interface to access data sources such as kernel and subsystem data structures, hardware dependent data, and data which is event-generated.

DCL
Data Capture Layer. A UMA entity concerned with the collection of raw data from the UNIX kernel and other sources. Data is considered collected when it exists assembled into data structures of predefined class and subclass in storage controlled by services contained in the measurement model.

Data Class
The general system measurement entity to be collected. For example, the data classes for UMA include system configuration information, processor and memory usage information, and other like categories. The UMA classes and subclasses are defined in this document.

Data Services Layer
A UMA entity responsible for data distribution to measurement applications using the MLI, for archival data storage, for management of services and resources required for distributed measurement access and control, for measurement requesting, and for data format transformations required for recording and transmission.

Data Subclass
A specific grouping of data within a data class. Each data class may have several data subclasses. For instance, the class ‘‘processor’’ contains subclasses such as ‘‘Global Measured Processor Times’’ and ‘‘Global System Call Counters’’, etc.

Disk Partition
A portion of a disk. A disk partition can contain a file system or a raw data structure such as swap space or raw database management space. To optimise disk performance, a file system is often comprised of several partitions spread across several disks. This includes a file system mounted on a logical volume, in which case the whole physical space allocated to a logical volume on a particular disk can be thought of as a partition.

Event Data
In the context of UMA, this represents the reporting of one or more system events (for example, process termination, creation, signal delivery, logon, etc.).

FEP
Front End Processor.

GID
Group Identifier of a process.

IOP
Input/Output Processor.

IPC
Interprocess communication.

KCORE
The amount of physical memory used in kilobytes * the time occupying physical memory (user time + system time).

Logical
The Logical Volume Manager supports logical volumes by managing the disks in small chunks, usually 4 MB. A logical volume may span several disks, and its size may be increased without disturbing the file systems.

Measurement Application Layer
In the context of UMA, this functional layer contains the application primitives and tools used to report currently captured and archival performance data to the end-user (or to an automated stand-in). These application implementations are called Measurement Application Programs (MAPs).

Measurement Control Layer
A UMA entity responsible for managing the capture of data, including its synchronisation, and for providing any necessary buffer or queue management for data assembled by the data capture mechanism.

Measurement Interval
A continuous time interval during which measurement activity and reporting is requested by a measurement application program (MAP).

Message
In UMA, a basic unit of control or data information. Each UMA message contains a header portion which identifies the class and subclass of the information contained in the rest of the message.

MAP
Measurement Application Program. A UMA-based application program providing end-user services.

MIB
Management Information Base.

MLI
Measurement Level Interface. The MLI comprises the Application Programming Interface (API), and the management of UMA message transport.

NQS
New Queue Service.

OSI
Open Systems Interconnect.

PID
Process Identifier.

PPID
Parent Process Identifier.

PMWG
Performance Management Working Group. The working group, originally within UNIX International, now within the Computer Measurement Group, that developed the base document for this specification.

processor
One of several computational engines that comprise a multiprocessor cpu. A single processor (uniprocessor) cpu has one processor.

Reporting Interval
The union of one or more contiguous collection intervals to be seen by a MAP. Thus the reporting interval may be identical to a collection interval, or it may have a duration that is (nominally) a multiple of the collection interval duration.

Sampling Interval
The time between successive samples during data capture.

SNMP
Simple Network Management Protocol

SVID
UNIX System V Interface Definition.

system
The hardware and software associated with a single running image of the operating system.

Trace Data
In the context of UMA, reported trace data is data for a set of selected, related events.

UID
User Identifier of a process.

UDU
UMA Data Unit. The contents of a UMA API Message. The UDU consists of a header portion, and either a control segment or one or more data segments.

UMA
Universal Measurement Architecture. A common, flexible measurement control and data delivery mechanism.

VM
Virtual Memory.

Index