Technical Report

CMU/SEI-93-TR-19
ESC-TR-93-318

An Ada Binding to the SAFENET Lightweight Application Services

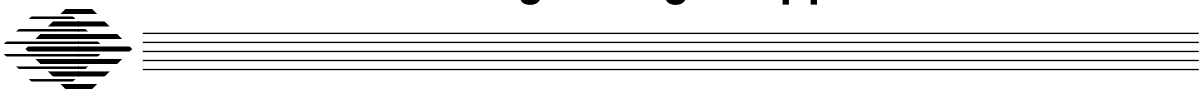
B. Craig Meyers
Gary Chastek

December 1993

Technical Report

CMU/SEI-93-TR-19
ESC-TR-93-318
December 1993

An Ada Binding to the SAFENET Lightweight Application Services



B. Craig Meyers
Gary Chastek

Open Systems Architecture

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

An Ada Binding to the SAFENET Lightweight Application Services

Abstract: This document describes an Ada binding to the Survivable Adaptable Fiber Optic Embedded Network (SAFENET) lightweight application services. The major goal in the design of the binding was schedulability. The document contains the Ada package specifications for the binding as well as a rationale for the design.

1. Introduction

The Navy Next Generation Computer Resources (NGCR) Program seeks to adopt and apply standards to the Navy real-time computing domain. There are three principal components of the NGCR Program relevant to the real-time domain. These are standards for an operating system (POSIX [IEEE91a]), a backplane bus (Futurebus+ [IEEE89]) and a local area network. The local area network standard is the Survivable Adaptable Fiber Optic Embedded Network (SAFENET), specified in reference [NGCR92a].

The purpose of this document is to define an Ada language binding to the SAFENET lightweight application services. These services are intended for use with a *lightweight* protocol that is believed particularly relevant to the needs of tactical real-time systems.[1] The Ada binding defined here represents one form of an application program interface (API).

This document is intended for two audiences. First, since it defines an Ada binding to the SAFENET lightweight application services, it is relevant to application developers. Second, the document is oriented toward implementors of the Ada specification. In both cases, the document includes a rationale that explains why certain choices were made. An important aspect of the rationale is the ability for an application to develop a schedulable system. To achieve schedulability, certain functionality is required in support of an application. Adherence to scheduling principles also places corresponding expectations on an implementation.

It is assumed that the reader has a general understanding of distributed systems as well as the SAFENET standard [NGCR92a]. The specification of the xpress transfer protocol (XTP) [PE92] on which the SAFENET lightweight protocol is based, is not required. However, readers who are interested in underlying details may wish to consult the references con-

[1] ''The lightweight profile is intended for situations in which either data transfer latency is critical, or multicast data transfer is required, or system-specific support services need to be implemented in place of ISO standard session, presentation, and application layer services.'' [NGCR92b]

tained in the SAFENET standard, particularly the specification for XTP [PE92]. It is important to note that the Ada binding is not to XTP *per se*. Rather, it is a binding to services for which XTP is the intended underlying protocol.

This document is organized as follows. Section 2 provides an overview of SAFENET including a description of the lightweight protocol suite. Section 3 describes the guidelines that were applied to the development of the Ada binding. Section 4 provides an overview of the API and its use by an application program. The following sections discuss the major components of the API, namely: protocol management, address management, communications support, connection management, group management, data transfer services, transaction services, event management, buffer management, and performance measurement. A brief summary of the document appears in Section 15. The Ada package specifications can be found in the appendices.

The work reported in this document was performed by the Open Systems Architecture Project at the Software Engineering Institute (SEI). The SEI is a federally funded research and development center operated under contract to the Department of Defense by Carnegie Mellon University. This work was supported in part by the Navy NGCR Program.

This report was originally released in support of NGCR activities associated with real-time distributed systems[2]. It was also used as a base document in the development of IEEE P1003.21 (POSIX) titled ''Real-time Distributed Systems Communication.'' For further information about these activities, readers are referred to the P1003.21 working group's documents.[3]

It is a pleasure to acknowledge colleagues Mike Gagliardi and John Goodenough for discussions related to this report. We also acknowledge Kent Meyer, a former resident affiliate at the SEI, for discussions related to implementation issues. We also thank Joe Gwinn for extensive discussions concerning real-time requirements.

[2]The original version of this report is available via anonymous FTP at site ftp.sei.cmu.edu, in the directory pub/posix/docs.

[3]In particular, there exists a document that identifies requirements for real-time distributed communications: see IEEE P1003.21 N008R6.

2. Overview of Safenet

2.1. SAFENET Protocol Suites

SAFENET can be considered in terms of three protocol suites.[4] These are

1. An Open Systems Interconnection (OSI) protocol suite that provides for OSI-compliant networking. The OSI protocol suite also includes directory services and network management.
2. A lightweight protocol suite that provides real-time data transfer. The protocols available to the lightweight protocol suite are the xpress transfer protocol (XTP) and the OSI connectionless transport protocol.
3. A combined protocol suite that is essentially the union of the OSI and lightweight protocol suites.

Figure 2-1 is a pictorial representation of the SAFENET protocol suites with the lightweight protocol suite shown in bold. At the top of the figure is the application interface to SAFENET. In the case of the lightweight protocol suite, for example, the application interface would be provided by an Ada binding such as presented in this report.

The standards that are referenced in the SAFENET standard [NGCR92a] illustrate the complexity of the SAFENET domain. The lower layers, which are common to both protocol suites, require

- Logical link control (LLC) [IEEE85].
- FDDI media access control (MAC) [ISO89b].
- FDDI physical layer protocol (PHY) [ISO89a].
- FDDI physical layer, media dependent (PMD) [ISO89c].
- FDDI station management (SMT) [ANSI91].

In the case of the lightweight protocol suite, the following are relevant:

- Lightweight application services; defined in reference [NGCR92b].
- OSI connectionless mode transport protocol [ISO87b].
- Xpress transfer protocol (XTP) [PE92].

Finally, for the OSI protocol suite, the following are relevant:

- OSI connection-oriented transport protocol [ISO86], [ISO89d].
- OSI connectionless mode transport protocol [ISO87b].
- Manufacturing automation protocol (MAP) [COS88].

[4] SAFENET is expected to also include the Internet protocols in a forthcoming revision. The use of Internet protocols is beyond the scope of this report.

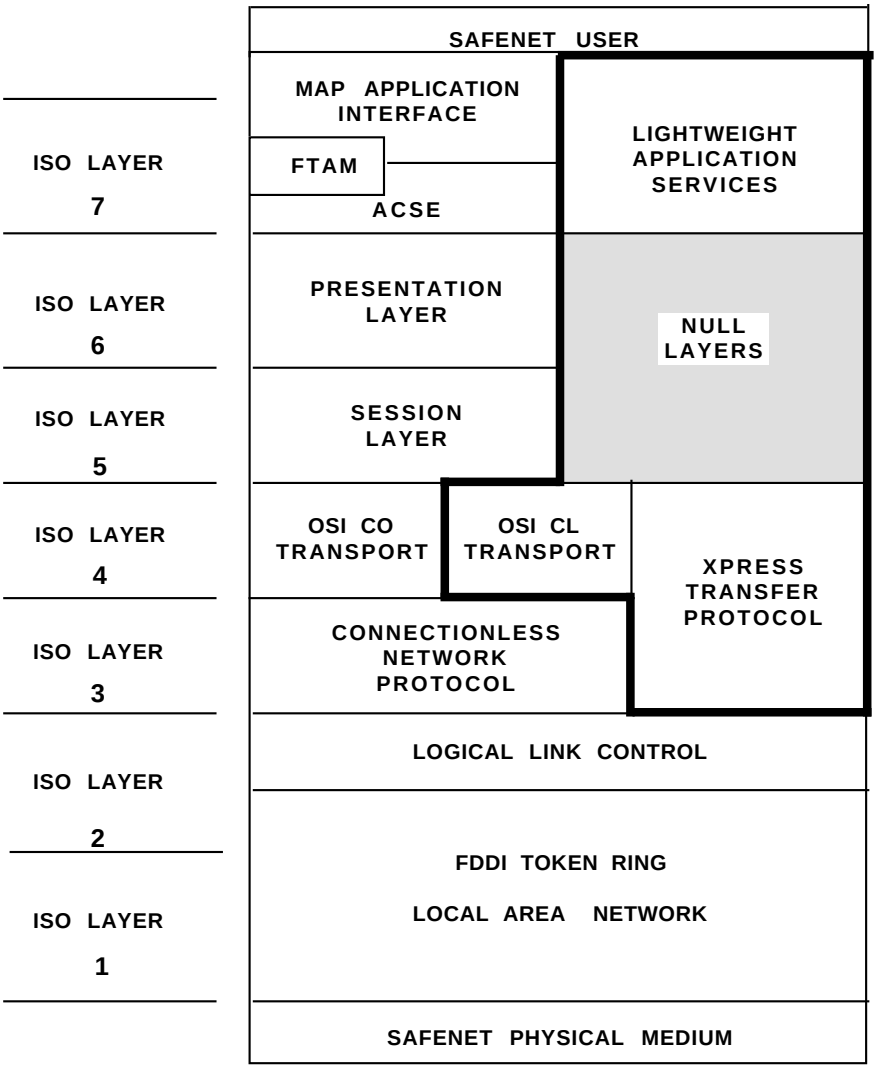


Figure 2-1: SAFENET Protocol Suites

The above references are principally oriented toward the protocols included in SAFENET. The standard refers to additional documents to complete the functionality provided for local area network operation. Two examples of this are a specification for network time services and the stable implementation agreements for open systems, developed by the National Institute of Standards and Technology (NIST).

The preceding has provided a cursory overview of the protocol suites that are defined in SAFENET. Note that an application can employ either the OSI suite, the lightweight suite, or both suites. The availability of several different protocol suites and the information required for their specification illustrates the complexity, and challenge, that faces the developer of an NGCR-based system that uses SAFENET for its local area network component.

2.2. Lightweight Protocol Suite

2.2.1. Overview

The lightweight protocol suite, defined in reference [NGCR92b], is oriented toward real-time data transfer.[5] This protocol is based on the xpress transfer protocol (XTP) defined in reference [PE92]. XTP is designed to meet the needs of real-time distributed systems. The protocol includes support for the following:

- Message priority and scheduling.
- Reliable multicast mechanism.
- Real-time datagram capability (reliable and unreliable).
- Flow and rate control.
- Selective retransmission and acknowledgement.
- Traditional stream capability.
- Compatibility with standard addressing schemes, such as ISO or Internet protocol.

XTP is a *lightweight transport* that provides end-to-end reliable sequenced data delivery. Part of the motivation for the development of XTP was that traditional protocols (such as TCP and TP4) were not believed sufficient to meet the needs of real-time distributed systems.[6]

2.2.2. Specification

The SAFENET standard specifies activities associated with the lightweight protocol suite in terms of service definitions and service primitives. Each of these is discussed in the following subsections.

2.2.2.1. Lightweight Application Services

The functionality specified for the lightweight protocol stack[7] is defined by a set of lightweight application services. Two classes of service are defined in reference [NGCR92b]. The first of these is known as *directory services* and provides for registration and deletion of individual and group logical names. The following services are provided:

---

[5]The SAFENET standard defines a real-time event as ‘‘An event which must be accomplished within an allocated amount of time or the accomplishment of the action has no diminishing or negative value.’’ [NGCR92a] Real-time systems are typically more concerned with latency requirements than throughput requirements.

[6]Note also the null layer above XTP in Figure 2-1. In the planned hardware implementation of XTP, only a minimal LLC functionality will be provided. Both of these points make XTP attractive to the real-time design community.

[7]We will use the terms *protocol suite* and *protocol stack* interchangeably. The latter is more representative of a data flow model.

1. SLA-REGISTER[8]
2. SLA-CANCEL
3. SLA-OPEN-GROUP
4. SLA-CLOSE-GROUP
5. SLA-JOIN-GROUP
6. SLA-LEAVE-GROUP

The second class of service defined in reference [NGCR92b] for data transfer services are

1. SLA-CONNECT
2. SLA-DISCONNECT
3. SLA-STREAM-DATA
4. SLA-UNITDATA
5. SLA-MCAST-UNITDATA
6. SLA-MCAST-STREAM
7. SLA-TRANSACTION
8. SLA-NOTIFY

2.2.2.2. Means of Specification

The majority of the specification of the lightweight protocol suite is in terms of textual material. In support of this, primitives and time sequence diagrams are also used.

The definition of services is supported through the use of service *primitives*. These are defined in terms of an abstract relationship between a user of the service and a provider of the service. Each primitive is implementation-dependent and has an associated direction of service, such as from a service user to a service provider.

The four primitives used by SAFENET in the lightweight application service definition are as follows:[9]

- *Request:* An interaction in which a service user requests some action of a service provider. The direction associated with this primitive is from service user to service provider.
- *Indicate:* An interaction in which the service provider indicates to a service user that it has performed some action. The action performed may have been initiated either by the service provider itself or by the service user at the remote service access point. The direction associated with this primitive is from service provider to service user.
- *Response:* An interaction in which a service user reports to the service provider that it has completed some action in response to an *indicate* primitive. The direction associated with this primitive is from service user to service provider.

[8]The acronym SLA, used in the following, refers to SAFENET lightweight application.

[9]The following definitions are from reference [NGCR92b].

- *Confirm:* An interaction in which a service provider reports to a service user the completion of some action previously initiated by a *request* primitive. The direction associated with this primitive is from service provider to service user.

The temporal evolution of a service can be represented using primitives in a *time sequence diagram*. For example, a *connect* service allows an application to establish a connection with another application. In terms of a time sequence diagram, the *connect* service is illustrated in Figure 2-2 below. In this figure, time increases down the page and the vertical lines denote *service access points*.

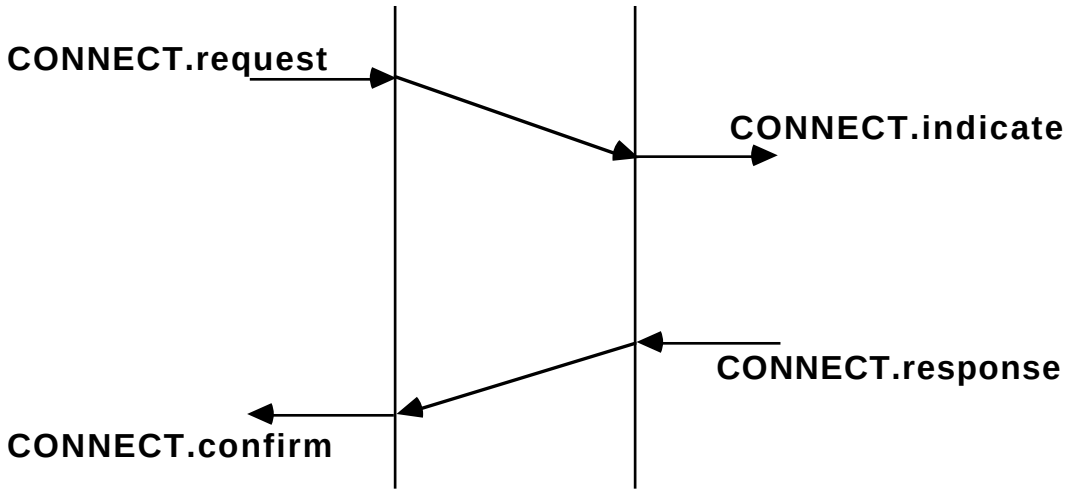


Figure 2-2: Time Sequence Diagram for Connect Service

The service primitives, discussed above, can be thought of in a data transfer sense, in that they denote directions of data transfer. The lightweight application services, defined in reference [NGCR92b], are principally defined in terms of the service definitions, which often have associated parameters. The service definitions are especially relevant to the Ada binding presented in this report.

2.3. SAFENET in the NGCR Domain

SAFENET represents one of the NGCR selected standards, two others being an operating system standard (POSIX) and a backplane standard (Futurebus+). It is worthwhile to examine the relationships among these components in the context of a system development.

Figure 2-3 shows a system partition based on an operating system and SAFENET components. For purposes of discussion, we assume that multiprocessor connectivity is achieved through a Futurebus+ backplane. We will also assume that the application software is written in Ada.

Figure 2-3 is partitioned into six regions. The characteristics of these regions are as follows:

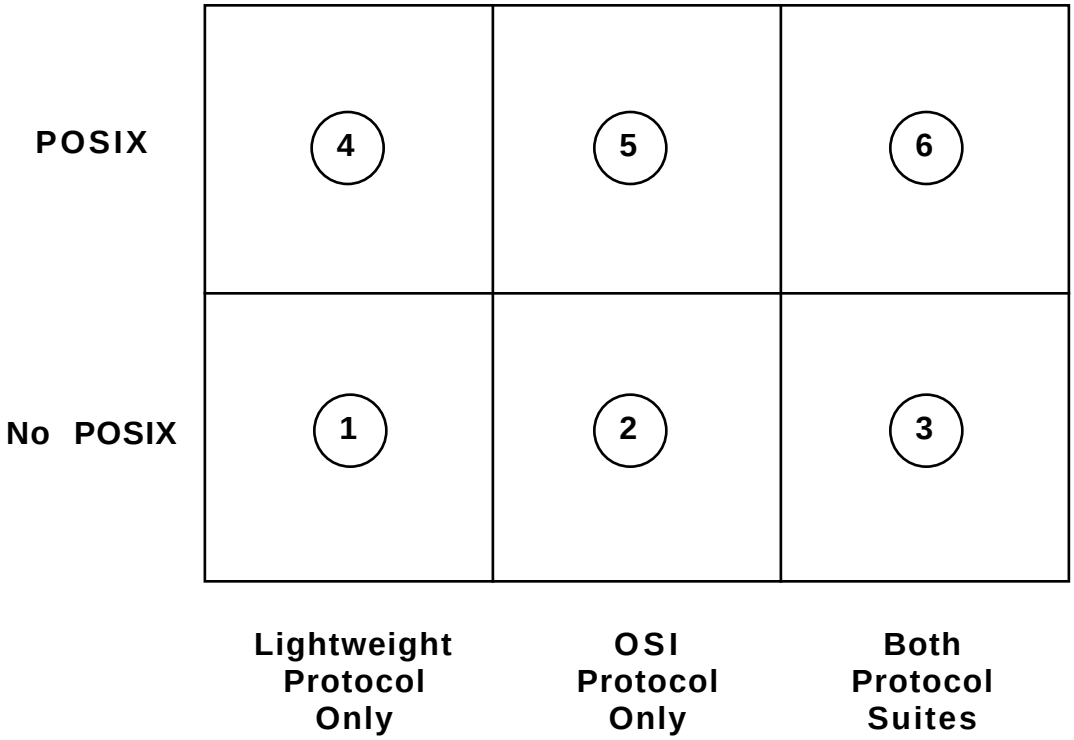


Figure 2-3: Scope of NGCR Domain

- Regions 1 and 2 have a single Ada program and only one protocol stack present. This is one of the simplest cases of an NGCR-based distributed system. The simplicity is achieved in that the application program can be viewed as the owner of the network resources.
- Region 3 is characterized by the presence of a single Ada application program that uses both protocol stacks. This is illustrated by a real-time application that uses the OSI stack for file transfers and the lightweight stack for data transfers over the LAN.
- Regions 4 and 5 may have one or more Ada programs present with only one protocol stack.
- Region 6 may have one or more Ada programs and both protocol stacks present.

The preceding partitioning of an NGCR-based system illustrates the range of complexity that such systems can have. The partitioning also identifies several issues that must be addressed by an application developer:

- *Is it possible to reuse software components for the lightweight and OSI protocol stacks?* For example, both the OSI and lightweight protocol stacks require address management. A common address management component that is independent of a protocol stack would be desirable. We will see later that this may not be possible. However, in the Ada bindings presented in this report, the dependencies on a particular protocol stack are localized.
- *How is arbitration for the network resources handled when there are two*

protocol stacks present and no operating system? Region 3 in Figure 2-3 implies a single Ada program operating with both protocol stacks. One of the issues is how to schedule requests for network services, such as for message transmission. For example, what are the implications of a file transfer over the OSI stack occurring concurrently with a message transfer over the lightweight stack?

- *How is arbitration for the network resources handled when there are two protocol stacks present as well as an operating system?* This case is a generalization of the previous case. A possible major difference is that there may be multiple Ada programs, accessing multiple protocol stacks, with POSIX providing the operating system services.[10] Again, how is the scheduling of the network accomplished? There are also questions about protection of possibly shared data that must be resolved.

Each of the cases above raises *system*-level issues that must be addressed to successfully create an NGCR-based system.[11] We believe that part of the activity associated with a successful development of a real-time system is an analytic approach to performance assessment. In this case, there are a host of questions that must be asked — and answered — if one seeks to build a predictable system.

[10]It is beyond the scope of this report to address issues relating to the interaction of POSIX and Ada. It is worth noting, however, that there is an ongoing effort within the POSIX community to develop a standard for a protocol independent interface to a LAN (POSIX P1003.12 and P1003.21).

[11]A system issue related to the development of an API to the SAFENET lightweight application services is whether a different API should be used, depending on the presence or absence of an operating system.

3. Guidelines for Development of Ada Binding

It is important to discuss the guidelines used in the development of the API to SAFENET. The API is defined in accordance with certain principles for the development of predictable real-time software. At the same time, it is necessary to consider portability issues, realizing that the Ada binding could be used on multiple platforms with implementation languages other than Ada. The following sections discuss the guidelines applied to the development of the Ada binding.

3.1. Goals

The goal of this work is to provide an Ada binding to the SAFENET lightweight application services. In the near term, one would like to be able to prototype the Ada binding to demonstrate its functionality. In the longer term, however, there is the intent to have a standard developed for the real-time distributed communications domain. In the latter case, the Ada binding could be used as a base document for standards purposes. The approach taken here is oriented more toward a long-term view.

The utility of the Ada binding for an application development may depend on many things. Two items of special importance are *functionality* and *performance*.[12] The ability to develop an Ada binding that will meet the needs of a plethora of users presents an almost insurmountable problem. Application developments often have unique requirements. Some typical issues that merit attention are the following:

- *Performance:* Performance is a consideration in all real-time developments, but in the hard real-time domain, it assumes additional importance. The specification of the Ada binding places requirements on an implementor which, in turn, can affect performance. Tradeoffs, such as between latency and throughput, must be considered. In this specification, we have placed greater importance on latency.
- *Safety Criticality:* There are applications that have special needs relating to safety for which special requirements are applicable. In addition, verification and validation also must be addressed. Access to source code is often required for this work. The above remarks also apply to trusted systems.
- *Error Handling:* The ability to detect and respond to errors in a timely manner can be important to an application. It can be vital that the application not lose control when an error occurs. An application can also require the ability to perform mode change operations. Such functionality is better supplied by the Ada binding than by the application.
- *Buffer Management:* An application can have special needs for buffer management. For example, in a performance-constrained system, a buffer management package might be included as part of the Ada binding. Moreover, the

[12] Note that there are inherent tradeoffs between functionality and performance. Frequently, greater functionality implies reduced performance.

buffer management could have unique needs, such as allocating buffers according to different boundary alignments (even address, odd address). In such cases there might be a need to allocate buffers using pointers and to attempt to minimize the time spent in data movement and access.

Other criteria relevant to the suitability of the Ada binding are discussed in the report. For example, a major design criteria is to permit an application to develop a *schedulable* system. The point in the preceding is that an Ada binding can be viewed from different perspectives; the ability to satisfy all possible users is, frankly, impossible. This report attempts to provide the functionality that is suitable for a broad range of real-time systems.

3.2. Conformance to SAFENET Specification

The Ada language binding must conform to the SAFENET standard. The SAFENET lightweight application services are defined in reference [NGCR92b] in terms of *service definitions*. We interpret a service definition as a statement of intent with regard to some implied functionality. Conformance can be interpreted to mean that the Ada binding should provide, as a minimum, the functionality specified in the standard and adhere to any explicitly stated requirements. One may construct a hierarchy of specifications that begins with a service definition and leads to a language-dependent binding.[13]

Conformance to the SAFENET standard need not be one-to-one. That is, there need not be a one-to-one relationship between a service definition in the standard and a corresponding Ada language element (such as a subprogram, or, in some cases, an exception). In the following we will see that there are times where a one-to-one specification can be provided in a straightforward manner. However, in other cases, it may be preferable to implement a particular service definition in more than one way. The Ada binding may also include functions other than those specified in the standard, provided such additional features do not contradict the standard.

Another point that should be recognized is that there are many items in the SAFENET lightweight application services that are not defined. Examples include the length of a group logical name, its representation, and whether the group name should be restricted to consist of only upper or lower case letters. We have selected values for the parameters and, where possible, provided a certain freedom to the implementor. The reason that many parameters are not specified in the SAFENET standard is that it would be viewed as possibly restrictive upon an implementor.

Two final remarks concerning conformance are appropriate. First, one reason why the

[13]For example, beginning with a service definition, the following sequence of refinements may be realized: a language-independent specification that carries no information concerning parameters, a language-independent specification that carries information about parameters, and a language-dependent binding. The role of parameter bindings has subtle implications for the ultimate language binding; for example, a parameter of integer type can be specified that is 32 bits wide, where the high-order bit is interpreted as data, not as a sign bit.

SAFENET lightweight application services are specified as services is to allow an implementation to be tailored to meet its specific requirements. In this sense, there is an apparent contradiction between conformance and the ability to tailor a standard. We would welcome a formal definition of conformance for service definitions that can be tailored by an implementation.

The second remark relates directly to the Ada binding presented in this report. It must be understood that different implementations of the Ada binding may not necessarily have the same *behavior*. There is no requirements specification to accompany the Ada binding; hence, its behavior may differ for different implementations. Note that this document does not provide a detailed semantic definition of the Ada binding. The issues associated with conformance, tailorability, and the lack of a detailed specification have clear implications for portability and interoperability, which are major goals of the NGCR Program.

3.3. Scheduling Theory

A major goal in the development of any real-time system is *predictable performance*. A real-time system must be not only logically correct but must also satisfy its timing deadlines. Thus the development of the Ada binding was heavily influenced by scheduling theory. The following subsections briefly describe scheduling theory and its impact on the Ada binding.

3.3.1. Scheduling Paradigms and Priorities

Given a set of possibly concurrent activities, the purpose of a scheduling paradigm is to specify the manner in which a given activity is selected for execution. Basically, scheduling paradigms can be divided into two classes.

1. *Explicit*: The scheduling decision is based on an explicit representation of time. A deadline-driven scheduling policy would be included in this category, for example.
2. *Implicit*: The scheduling decision is based on an implicit representation of time. Here, a scheduling policy is based on some relative criteria.

For a given scheduling policy, the decision of which task should be scheduled is resolved through the use of a priority mechanism. We assume the existence of a preemptive scheduling mechanism. In an explicit scheduling policy, the representation of priority is some abstraction of time. In an implicit scheduling policy, the representation of priority is typically a mapping into the integers.

A problem that must sometimes be addressed in dealing with priorities is the issue of *priority mapping*. For example, an explicit scheduling policy may only represent an abstraction of time to a granularity of 10 milliseconds. In this case, two dissimilar application priorities could be mapped onto the same value. In the case of an implicit scheduling policy the underlying system may only provide support for four levels of priority. The application program, on the other hand, may require more than four levels of priority.

3.3.2. Schedulability

Consider a set of tasks that can be executed on a particular system. The term *schedulability* refers to the ability for a task to satisfy its deadline(s). The execution time of a particular task, say τ, is a function of three activities.

1. The *preemption* of task τ by those tasks having higher priority than τ.
2. The *execution* time required for task τ.
3. The *blocking* of task τ by other tasks having lower priority than task τ.

To meet its deadline a task must be able to complete its execution in the presence of preemption as well as blocking. Blocking, or priority inversion, may occur for many reasons.

- The lack of a sufficient priority representation may require two dissimilar application priorities to be mapped into a single system priority. In this case, typically, one does not have access to the details of the underlying scheduling policy to know when a particular task is scheduled.
- The use of FIFO queues in message transfer where a high-priority message is forced to wait pending the transmission of a low-priority message.
- The locking of a shared resource in a non-preemptable manner by a lower priority task while some high-priority task requires access to the shared resource.

The possibility for priority inversion is a natural consequence of shared resources, such as queues or devices. Note also that priority inversion is not always bad, as in the case of a shared resource where data integrity is a major consideration. What is important to recognize, however, is that unbounded priority inversion is unacceptable in a real-time system. In fact, one of the goals in the design of a real-time system should be to minimize the duration of priority inversion.

It is possible to place the notion of schedulability on a quantitative basis. The completion time $t*$ of a task τ is a function of its own execution time, the preemption time, and the blocking time due to other tasks. To analytically represent the completion time $t*$, we partition the task set into those tasks whose priority is greater than τ and those tasks whose priority is less than τ. The former contribute to the preemption of task τ and will be denoted P^+. Similarly, the latter task group contributes to blocking experienced by task τ and will be denoted P^-.

Mathematically, the completion time $t*$ can be represented by

$$t*(\tau) = C(\tau) + \sum_{\tau_i \in P^+} P(\tau_i; \tau) + \sum_{\tau_j \in P^-} B(\tau_j; \tau).$$

The first term in the above expression represents the execution time consumed on behalf of task τ. The second term represents the contribution of preemption from tasks τ_i, while the third term represents the effect of blocking of task τ by tasks τ_j. The summation in the second term is over those tasks whose priority is greater than that of task τ and the summation in the third term is over those tasks whose priority is lower than that of task τ. For purposes

of discussion, we assume that each task has a unique priority assignment.[14]

The execution component of task τ, denoted $C(\tau)$, can be represented as follows:

$$C(\tau) = C_0(\tau) + \sum_{\tau_k} I(\tau_k; \tau)$$

where $C_0(\tau)$ is the execution of task τ independent of any other task and $I(\tau_k; \tau)$ represents the amount of time that task τ interacts with some other task τ_k. The summation in the above is over all such tasks that interact with task τ.

The preceding analytic development applies to the schedulability of an application that is using the SAFENET lightweight protocol stack as part of a distributed system. A full discussion of this is beyond the scope of this document, but several examples can be noted, such as

- Consider the case in which an application task τ is sending a message to another application. If the process of sending a message blocks task τ in a non-preemptable manner, the completion of task τ is delayed due to the (direct) blocking, *and* the blocking serves as a source of priority inversion to tasks of higher priority than τ.
- Consider the case in which an application task is opening a connection with another application. If this activity is performed in a non-preemptable manner, it may possibly block for the duration of time required to establish the connection. In the event of an error in the process of establishing a connection, the blocking could potentially be unbounded. Similar to the above case of message transfer, this blocking may have *system* consequences.

The preceding illustrates two cases of how an analytic approach to scheduling theory can help a designer develop a perspective of the system in terms of schedulability. In fact, it was the recognition of these principles that forms the basis for considerations in the development of the Ada binding presented here. For example, many of the subprograms that are part of the Ada binding have a timeout parameter to bound the blocking experienced by an application task. It is the recognition of an analytic and disciplined approach to schedulability considerations that will help software developers assure that essential timing requirements can be satisfied.

3.3.3. Scheduling Policies in the Lightweight Protocol Suite

It is important to note that the SAFENET standard does not explicitly state any scheduling policies. Nevertheless, there are certain implicit scheduling policies. In the following, these policies are discussed in the context of the analytic approach discussed above.[15]

---

[14]The case in which several tasks have identical priorities must be treated with care. It requires explicit knowledge of the scheduling policy, for example, if a fairness doctrine is imposed.

[15]We assume familiarity with the details of the SAFENET lightweight application services specified in reference [NGCR92b]. The equivalent material for connection management and data transfer services is discussed in Sections 8 and 10, respectively. The reader may wish to return to this section after reading that material.

An implicit scheduling policy is present in the lightweight application services because of so-called quality of service (QOS) parameters. These parameters are designed to provide an application with certain performance characteristics.[16] Of particular relevance are the following three quality of service parameters:

1. *Transit Delay*: The maximum time for delivery of a single message to a remote user.[17]
2. *Maximum Latency*: The maximum time between a request and its corresponding *confirm*.[18]
3. *Priority*: The importance of the request relative to other requests.

Quality of service parameters can be viewed as elements of an implicit scheduling policy. This is an important point. For example, assume that an application has an end-to-end deadline for transmission of a message to an external system. The time at which the message is transmitted can be viewed as a function of the quality of service parameters. In a simple case, that time includes the transit delay as a scheduling component.

The above quality of service parameters can be specified for several of the lightweight application services. In particular, the following are to be noted:

- Either maximum latency or priority can be specified for opening a connection.
- Either transit delay or priority can be specified for unicast and multicast services.
- Either maximum latency or priority can be specified for unicast transaction services.
- Only priority can be specified for the closing of a connection, stream data, and *notify* services.

It is quite clear that the scheduling policy of the underlying implementation must account for the quality of service parameters defined above. Yet it is not clear what (functional) relationship is assumed to exist for the quality of service parameters. For example, what is the relationship between priority and maximum latency? Moreover, an application can request services in an inconsistent manner. For example, one would assume that the smaller the maximum latency, the higher the priority in performing the service. If an application makes a request that specifies relatively small latency but low priority, what action should the underlying scheduler perform?

The point brought out in the above, aside from the issue of possible user conflicts, is that merely stating the quality of service parameters does not provide a *consistent* scheduling

---

[16]Other quality of service parameters relate to reliability of data transfer and will not be discussed here; refer to Section 10 for a discussion of how these parameters are interpreted in the Ada binding.

[17]While we recognize the utility of a time distribution for transit delays, we feel that a maximum time is sufficient for our purposes.

[18]Note that this definition could be interpreted in both a local or remote context.

policy. The specification of a consistent scheduling policy requires knowledge of the representation of priority (or equivalently, for example, deadline) as well as the priority mapping function. For example, what is the function that converts a latency into a priority and vice versa? None of this information is specified in reference [NGCR92b]. The lack of a consistent scheduling policy in the SAFENET lightweight application services is due to (1) the use of multiple parameters for specifying the scheduling decision, and (2) the lack of information about the priority representation and the mapping function. We believe that the above scheme can introduce more problems than it purports to solve.[19] A full discussion of scheduling policies in the SAFENET lightweight protocol suite is beyond the scope of this report. The following section will discuss the approach taken in the development of the Ada binding with regard to scheduling.

3.3.4. Approach Taken

The API was designed to support the construction of application programs with predictable behavior. This is accomplished, in part, through the use of a scheduling mechanism, priorities, and means to limit priority inversion. Briefly, the API will be based on the following:

- An implicit scheduling policy will be used. This means that deadline-driven scheduling is not supported *within the API*.[20]
- The arbitration for network requests, such as connection establishment and data transfers, is based only on the use of priorities. There is no requirement defined or assumed in the Ada binding concerning the priorities associated with data transfers and the priorities of application tasks that may invoke data transfer services.
- The quality of service parameter *transit delay* will be included in data transmissions and interpreted as a guide to the maximum lifetime for the data on the network.

We emphasize that the above choices are motivated by the goal of seeking a consistent scheduling policy. In the case of a distributed system, this includes the operating system and backplane bus scheduling policies. In the SAFENET context, one must also consider the underlying protocol specifications, such as XTP and FDDI.

[19]Note that the QOS parameters relevant to the connectionless transport (which is available in the lightweight suite) are *priority* and *transit delay*; maximum latency is *not* relevant for this protocol. This indicates another case in which an inconsistency could be present in the scheduling policy.

[20]There is nothing to prevent an application from mapping deadlines onto priorities. Readers should also note that XTP has removed the use of timing as a scheduling mechanism.

3.4. Message Class

SAFENET specifies the service parameter *special-data* as an eight-octet value supplied by a sending application. This value is to be transferred to each corresponding receiving application in the *btag* field of the XTP packet. Note that the mapping of the *special-data* service parameter onto the *btag* field is an explicit protocol dependence.

The special-data parameter can be interpreted as a *message class*. The application can associate a message class with a particular type of data. This allows applications to perform operations on a message class.

The introduction of a message class provides considerable versatility to application designers. For example,

- An application can request to receive only those messages of a particular message class.
- The notion of message class can be applied in other contexts. For example, one can implement multicast operations by using message class to identify a particular multicast group.

Other, more sophisticated uses of message class are possible. For example, one could partition the message class into two components, such as *multicast group* and *message type*. An application could then wait for the receipt of messages from a particular multicast group *and* possibly one or more message types. Note that in SAFENET this is equivalent to an *application-defined protocol*. In the Ada binding we did not allow such interpretations of message class. Instead, we provided *get* and *send* operations on message class as a single entity.

3.5. Design Considerations

3.5.1. Synchronous and Asynchronous Behavior

Two recognized operating system interface techniques applicable to real-time systems involve synchronous and asynchronous communication. The classical model of synchronous communication involves an application making a request and then waiting until a response is provided. In this model the blocking experienced by the application may be unbounded. A refinement of the classical approach would include a timeout that indicates the maximum amount of time that an application is willing to be blocked, awaiting completion of the requested activity. A schedulability guideline is that one should seek to minimize (or bound) the amount of blocking that an application will experience.

In an asynchronous communication, a request is made and then the requester continues execution. At some later time, the requester makes a second request to determine the result of the activity. This can be achieved by a periodic polling on the activity or by inquiring about the state of the activity in an aperiodic manner. Note that blocking can be experienced in the asynchronous case if the application task (that receives the event) does not gain control after the arrival of the response.

The Ada binding provides support for both synchronous and asynchronous communication. This choice was dictated by the needs of real-time systems, particularly in the distributed domain. In the case of a synchronous operation, a timeout parameter can be specified. The purpose of the timeout is to limit the blocking that an application can experience. By setting the Boolean variable *suspend* to *true*, the invoker of a synchronous operation is suspended. If *suspend* is *false*, the requesting task will wait on either (1) completion of the request, or (2) expiration of a specified timeout. In the case of asynchronous operation, the following model is used:

- Each asynchronous operation has an associated identifier that uniquely identifies the request. The identifier is assigned by the Ada binding and returned to the application.
- A function is provided that the application can invoke to determine the state of the activity. An application can use this function to build different response models (such as periodic polling, timeouts, or an indefinite wait).
- The application can include a timeout parameter that is interpreted as a deadline for completion of the activity.[21]

The Ada binding uses a simple mechanism to allow an application to specify a synchronous or asynchronous operation. This is achieved in the following manner:

- In making a request, an application specifies the type of activity (either synchronous or asynchronous).
- As part of requested operation, an activity index is returned to the application. That index identifies the requested activity. In the case of a synchronous request, the value will be returned as zero. In the asynchronous case, the index can be used to determine the status of the activity.
- A timeout parameter can be specified in either the synchronous or asynchronous case and is interpreted as the maximum time the application is willing to wait for completion of the requested activity. The purpose of the timeout parameter is to bound the blocking that the application can experience during the requested activity.

The purpose of the mechanism indicated above is to limit the amount of code required to support both synchronous or asynchronous operations.

3.5.2. Treatment of Error Conditions

Real-time systems require the ability to detect and recover from errors. Certain critical applications can require *timely* detection and recovery from error conditions. Further, errors cannot cause a loss of control. A major design factor in the development of the Ada binding was the ability to provide error detection and recovery mechanisms. A challenging problem

[21] It is important to note that the inclusion of a deadline does not introduce an additional *scheduling* policy. The scheduling mechanism used throughout the Ada binding is based solely on priorities. The purpose of the timeout parameter, including its use in the synchronous case, is to limit the amount of time that an application will wait for an activity to complete.

is to develop a scheme that is versatile, recognizing that different applications can have different needs.

A variety of approaches to error detection are possible in Ada. One approach is to use an enumerated type whose elements indicate possible outcomes from a procedure call. Consider the case of a multicast transfer. One way to recognize an error would be to return an object of type *Multicast_Result* defined as follows.

```
type MULTICAST_RESULT is (OK, INVALID_GROUP_NAME,
        UNKNOWN_GROUP, GROUP_IS_EMPTY);
```

When a procedure completes, the result is checked and if an error is present, the application responds appropriately.

A second approach to error detection is through the use of the Ada exception mechanism. In the above example, one could define an exception *Multicast_Error* that is raised if an error is detected. It is possible to also define multiple exceptions to determine the specific cause of the error.[22]

There are several drawbacks with the approaches presented above. An application might not be able to perform error recovery in a timely manner; hence, returning an enumerated type whose members indicate the details of an error may not be useful. On the other hand, raising a single exception, such as *Multicast_Error* in the above example, may not be satisfactory in the case in which an application requires details about the nature of the error. Ada does not allow exceptions to pass parameters.

One design approach that seeks to effect a compromise between the details associated with an error and the way in which an application would respond makes use of exceptions, supplemented by *inquiry functions*. This approach involves the following:[23]

- A minimal number of exceptions (preferably one) is used; the exception is raised upon recognition of an error.
- Inquiry functions are provided that can be used to determine the specific nature of the error if an exception is raised. Indeed, one could use the inquiry functions *before* a procedure is called to be assured that an exception will not be raised. Here, one seeks to prevent an exception from being raised.[24]

Consider an example of the use of inquiry functions for the case of a multicast transfer, where two possible inquiry functions are as follows:

---

[22]The use of multiple exceptions was considered in the POSIX Ada bindings; the decision was not to use such an approach.

[23]The use of exceptions and inquiry functions was conceived by J. Goodenough.

[24]Race conditions occur when using inquiry functions. This problem can be worse when the function have a remote scope.

```
function GROUP_IS_KNOWN (NAME : in GROUP_NAME)
    return boolean;

function GROUP_IS_EMPTY (NAME : in GROUP_NAME)
    return boolean;
```

The above functions can be used to determine if there is an error associated with the indicated multicast group. The use of inquiry functions and a minimal number of exceptions can be viewed as a multilevel error detection scheme, allowing an application to provide a general or specific error detection scheme.

The use of a minimal number of exceptions supplemented by inquiry functions has certain performance penalties. For example, if a record is passed to a procedure, it may be necessary to have many inquiry functions (possibly one inquiry function per record component). There are also performance issues associated with the use of Ada exceptions. The use of inquiry functions can also be time-consuming in the real-time domain.

The approach used in the Ada binding was motivated by the following perceived application needs:

- Applications can have multi-level error detection requirements. Some applications might need an indication that an error has been recognized; other applications might need detailed information about the nature of the error.
- Applications may need to respond to errors in different ways. For example, it is easy to envision the need for a priority associated with an error.
- Applications may need to respond to asynchronously generated errors.
- Multiple application entities may require knowledge of the existence of an error.

A general description of the approach used to satisfy the needs listed above makes use of an event registration and recognition capability. This is discussed in the following section. Such a procedure is believed sufficiently general to meet the needs of most applications.

Given that an error has been detected, different applications have different requirements for error recovery. The nature of the error will frequently determine the scope of the recovery required. When more serious errors are detected, the application response may be correspondingly larger in scope. Provisions for mode change operations are incorporated in the Ada binding. For example an application might need to perform the following operations:

- Close a connection in either a graceful or hard manner.
- Close all connections in either a graceful or hard manner.
- Terminate all asynchronous activities, such as pending receipt of a message.
- Cancel an ongoing transaction or all transactions.
- Terminate all asynchronous responses expected from operations that were initiated before some specified time.

The Ada binding provides the capability to meet each of the needs above. It is also possible for an application to build upon the functionality provided by the Ada binding for its specific needs.

3.5.3. Event Handling

Real-time systems are characterized by the need to respond to asynchronously generated events. Examples of this include receipt of data from external sources or processing for alert conditions. The API includes support for handling asynchronously generated events. In addition, the event handling packages also provide error detection facilities.

To illustrate the motivation for the design of the event handling capabilities, we will consider an example. One mechanism of communication in a distributed system involves the use of connection-oriented transfer. When using this method, it is necessary for an application to establish a connection with some external entity. Figure 3-1 shows such a case where application tasks τ_i are communicating with external applications using connection-oriented transfer.

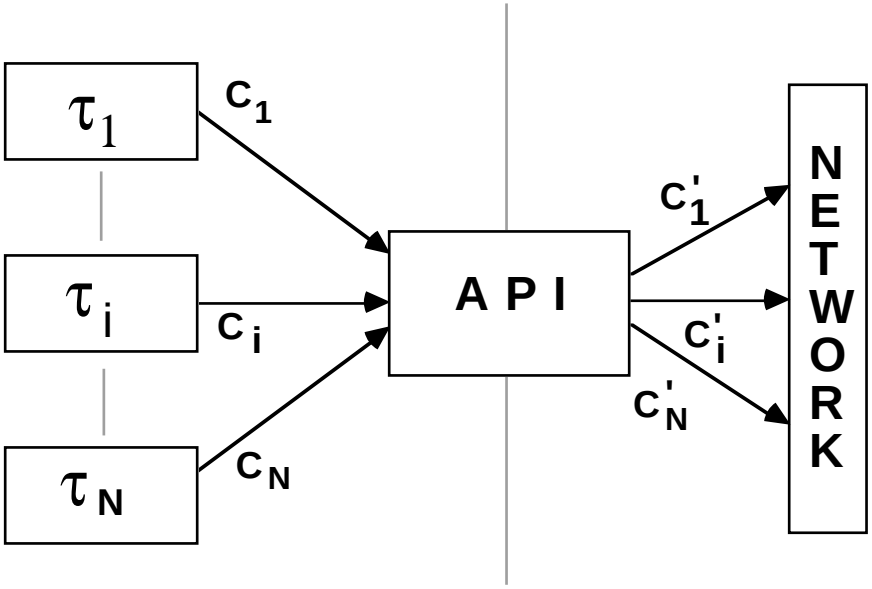


Figure 3-1: Application Using Multiple Connections Managed by API

The application tasks shown in Figure 3-1 will invoke the Ada binding to establish connections with external sources. It is important to recognize that there are multiple views of a connection. In the case of application task τ_1, for example, its view of a connection is denoted C_1. The Ada binding, which manages the connection on behalf of task τ_1, has the view indicated by C'_1. This distinction is important in the following sense: if a timeout condition is recognized for connection C_1, it will be recognized by the API. It is then necessary for the Ada binding to inform the application that a connection timeout has been recognized. This creates the need for asynchronous communication between the Ada binding and the application.

In the application design process, there can be several factors relevant to connection management. Some examples of these factors are as follows:

- Not all connections are of equal importance. For example, a connection to a critical (network) resource is viewed as more important than a connection to some other resource.
- Fault processing can be very application-specific. For example, if a task τ_i establishes a connection C_i, it is not necessarily true that this task will be responsible for processing errors associated with connection C_i. It is quite possible that some other application task will perform error processing for C_i. Moreover, an application could choose to have multiple tasks independently informed in the event of failures.

The above examples illustrate that applications have different requirements that can be implemented in different ways. A goal of the API specification is to provide support for these differing requirements. Continuing with the above example, the following are relevant to the problem of asynchronously generated events.

- Events can be generated for a variety of reasons; the application should be able to configure which events are generated depending on some specified reason.
- Given that there can be multiple events for a connection, the application should have the ability to determine how the events will be processed. For example, an application may need to process events in a prioritized manner or in a FIFO manner. In the latter case, the application may need to receive events in an oldest-first or newest-first manner.
- An application might need to wait some specified time period in anticipation of an event.

The event handling facilities provided by the Ada binding must be based, in part, on the manner in which an application will use the event facility. There are several different design techniques that can be applied, and Figure 3-2 illustrates three cases. This figure shows an application task τ_j that will receive some event generated by the API.

The first design model is one in which the API generates an alert. In an Ada design, one way to do this is to raise an exception. This approach was discarded for several reasons. First, an application requires information associated with the event, such as the time of the event; such information is not provided by the raising of an exception. A second reason is that when an exception is raised, the ability to return to the previous execution point is difficult at best. A third reason relates to scope considerations in handling an exception.

A second technique for implementing event handling in method 1 is for the API to explicitly call some application task. It could be possible for the API to call a *predefined* application task when an event is recognized. For example, it could be required of all applications to have a task such as the following:

```
task APPLICATION_EVENT_HANDLER is

   entry EVENT;

end APPLICATION_EVENT_HANDLER;
```

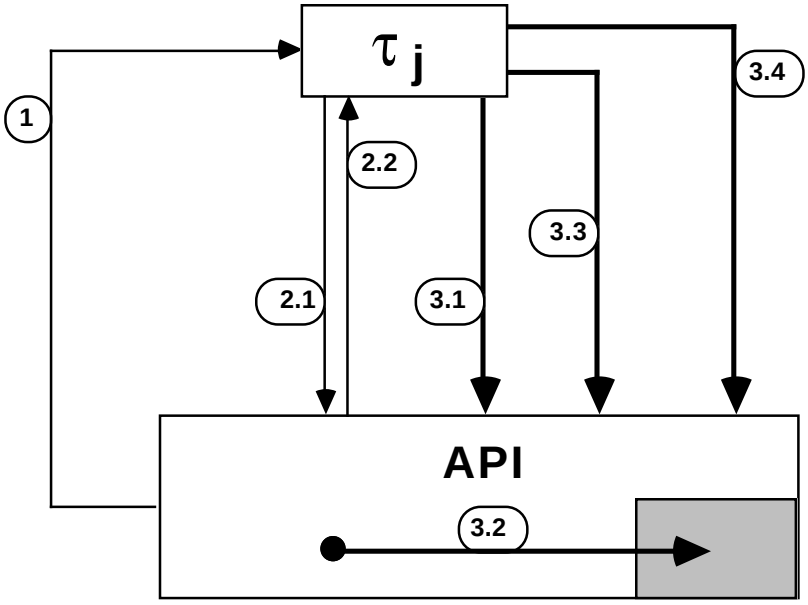


Figure 3-2: Three Design Models for Event Handling

The labels on the arrows refer to three different techniques for error handling; see text for discussion.

When an event condition is realized, the Ada binding would call this predefined task. This mechanism for delivery of events was discarded for several reasons. First, it can be viewed as placing undue design constraints on an application. Second, it is not clear what data should be provided to the application; this is related to the issue of application configurability of events. Third, it introduces extra (unnecessary) tasks. A final reason for discarding this mechanism is based on schedulability concerns: if the Ada binding calls on the application task and the application task is not at the *accept* statement, the Ada binding will be blocked. A schedulability perspective dictates that blocking should be minimized. Clearly, the possibility of *unbounded* blocking must be avoided.

The second design approach for handling events involves a request-response mechanism. Here, the application calls the API requesting an event and the API responds to the request. This represents a polling approach. We chose not to employ this method for reasons that will be evident below.

A third method for handling the events generated by the API, shown in Figure 3-2, was chosen for the Ada binding. The method is based on a register, request, response, and removal model. The sequence of steps involved include the following:

- An application task *registers* with the event handling facility for the event(s) that it will process. The registration process should provide flexibility in terms of what events may be registered for.
- When an event is recognized by the API, the information appropriate to the event is saved (in some data structure protected by the API).

- An application determines that an event has been recorded. A procedure is provided to determine if this is the case (including a *wait* mechanism). Another scheme is for the application to poll the event handler. In either case, the application can then request the information associated with the event. The order in which the event data is returned can be specified in either a prioritized or (time-order) FIFO manner.
- When the application has completed event processing it requests that the event data be removed. This operation can be included in the above step; that is, the application can get and remove event data in a single operation.

The design approach outlined above is deemed sufficiently general to meet the needs of most applications. Some comment should be made about the use of polling for the application to determine that an event has occurred. Consider the case in which an application is using many connection-oriented data transfers. In a pure polling approach, it would be necessary for the application to poll *each connection* to determine if there is an event present. The design of the Ada binding permits an application to determine if *any* event has been detected for a connection and process it appropriately. In a sense, the design of the API permits an application to perform a more *efficient* polling scheme than would otherwise be possible.

Further discussion related to the event-handling facility in the Ada binding can be found in Section 12.

3.6. Implementation Considerations

The Ada binding to the SAFENET lightweight application services included with this report is presented as Ada package specifications. It is anticipated that the implementation may be in some language other than Ada and some remarks about this are provided are below.

In interfacing Ada to another language, care must be taken to assure a consistent view of data structures. Ada provides implementation-dependent support to meet this requirement, and we will now consider an example of this. Consider the case in which there is an enumerated type to identify whether the XTP or OSI connectionless transport is to be used in a connectionless transfer. This could be represented in the form

```
type PROTOCOL is (XTP, OSI_CONNECTIONLESS);
```

An Ada compiler has freedom as to the manner in which the above statement will be implemented. However, when interfacing to some other language, the other language requires implementation-dependent information. In the above example, this could be accomplished in the following manner:

```
type PROTOCOL is (XTP, OSI_CONNECTIONLESS);
for PROTOCOL use (XTP => 0, OSI_CONNECTIONLESS => 1);
for PROTOCOL'size use 32;
```

The enumeration representation clause specifies the values to be used to indicate the XTP

and OSI connectionless protocol. The last statement restricts the size allocated to store objects of type *Protocol* to 32 bits. The latter two statements above can be declared in the private part of the package specification. The Ada binding presented in the appendices of this report does not declare any representation clauses, as these are left to the implementor.

There are certain restrictions placed on the implementor of the Ada packages presented in the appendices. These are

- An implementor may not define (and *with*) any packages.
- Subject to the items listed below, an implementor shall not change the visible part of the Ada package specifications.
- An implementor is free to modify the private part of any package specification. This is required to allow the packages to be implemented in some language other than Ada.
- An implementor is free to include Ada language pragmas in the visible part of a package. The inclusion of pragmas is related to performance rather than functionality.
- An implementor shall not preclude the use of the Ada binding in a reentrant manner. This requirement is included to support Ada multitasking applications.

The only way to implement the Ada binding in another language while still conforming to the requirements listed above is to provide a package body and interface functions (to the other language). This point is considered further in Appendix L.

The implementor is provided with certain freedom in the definition of the visible part of the packages. The implementation dependencies are detailed in the report. Some examples of where an implementor has freedom with regard to the package specifications include the following:

- An implementor is free to change the parameters in the package *LW_Protocol_Management*. The code appearing in Appendix B is provided as a suggested model. This option is provided because the definition of parameters associated with XTP are implementation-dependent.
- An implementor may replace or extend the reasons associated with the event management handling in Appendix I. This permits an implementor to extend the reasons for which errors are reported, although the model for error handling is defined by the Ada binding.
- An implementor may supplement the functionality of the package *LW_Performance_Measurement*.

In each case in which an implementor changes the visible part of the package, the implementor shall document the changes. The above options allow an implementor to extend the Ada binding to some degree. It is an open issue whether or not the implementor should be allowed to add functionality to any package beyond that specified in this report.

4. Overview of the Lightweight Architecture

This section presents a high-level overview of the software architecture to support the SAFENET lightweight application services. Figure 4-1 shows the packages and their dependencies.[25]

The following subsections provide a brief overview of each package; detailed discussions appear in Sections 5 - 14. The Ada package specifications are presented in the appendices. All of the Ada package specifications included in this report have been compiled with the VAX/Ada compiler. In addition, the packages have been compiled using the Systems Designers (XD, Version 1.2) compiler and the Telesoft TeleGen2 Sun3 cross-compiler (Version 1.4a), both for a Motorola 68020 target. This section concludes with some remarks about how an application would use the Ada binding.

4.1. Protocol Management

Certain operations are required for an application to use XTP. These operations are largely for initialization, such as queue sizes used by XTP. The following operations are defined in the package *LW_Protocol_Management*:

- Ability to initialize or terminate the lightweight protocol suite.
- Ability to get or modify the parameters required for the lightweight protocol suite.

The Ada binding for lightweight protocol management is discussed in Section 5 and the Ada package specification appears in Appendix B.

4.2. Address Management

Elements of a distributed system are identified by a *logical name* and an associated *physical address*. It is easier for an application program to deal with a logical name than a physical address (which can be dynamic if a reconfiguration occurs, for example). The purpose of the package *LW_Address_Management* is to support the mapping between logical names and physical addresses. This package provides for

- Binding a logical name to a physical address. This binding is referred to as an address *mapping*.
- Deleting a previously defined mapping. This can be accomplished by specifying both logical name and physical address, or either a logical name or physical address.
- Providing information about mappings, such as if a given logical name or physical address is currently bound, obtaining the logical name or physical address, and determining the number of current mappings.

[25] An arrow from package i to package j means that package j *withs* package i.

The Ada binding for address management is discussed in Section 6 and the Ada package specification appears in Appendix C.

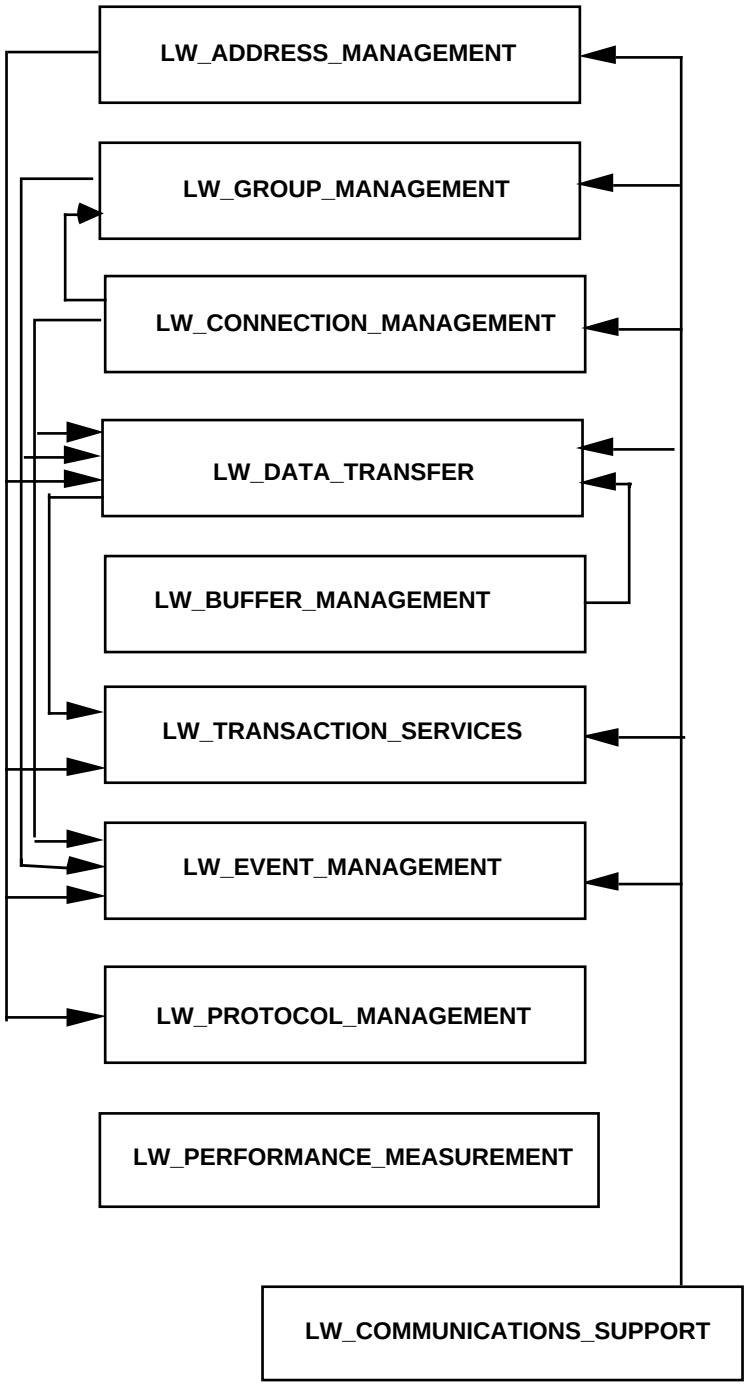


Figure 4-1: Overview of Lightweight Protocol Suite API

4.3. Communications Support

This package defines types and subprograms used by other packages. The types are as follows:

- Logical names
- Priority and timeout
- XTP-related parameters
- Data buffers

In addition, this package defines types and subprograms that are used in support of asynchronous communication.

The Ada binding for communications support is discussed in Section 7 and the Ada package specification appears in Appendix D.

4.4. Connection Management

The package *LW_Connection_Management* provides support for operations associated with connections. The connection management functions are used in data transfer services and are as follows:

- Establish a connection between two users with the option of including a message.
- Remove a connection between two users.

The above are specified in reference [NGCR92b]. In addition, the package *LW_Connection_Management* includes a state machine model of a connection. Functions to provide the state of a connection are defined in this package. Also, several additional functions are defined that are believed useful to an application, such as determining if a connection exists to an external application.

The Ada binding for connection management is discussed in Section 8 and the Ada package specification appears in Appendix E.

4.5. Group Management

The SAFENET standard specifies services related to multicast operations, namely, data transfers and transactions. The package *LW_Group_Management* is defined to provide support for groups that use these services. The basic functionality exported by this package is as follows:

- Opening a group
- Closing a group
- Joining a group

- Leaving a group

This package also provides functions to query if a group exists and to determine if some logical name is a member of a group.

The Ada binding for group management is discussed in Section 9 and the Ada package specification appears in Appendix F.

4.6. Data Transfer Services

The functions for the exchange of information between two application processes are defined in the package *LW_Data_Transfer_Services*. The basic functionality provided is as follows:

- Unicast data transfers (connectionless and connection-oriented)
- Multicast transfers
- Data transfer services can include a *message class* as part of the specification.

This package also provides for determining the number of messages that await processing. Certain mode change functionality is also provided, such as deleting all pending messages on a specified connection. The Ada binding for data transfer services is discussed in Section 10 and the Ada package specification appears in Appendix G.

4.7. Transaction Services

The SAFENET standard also specifies a transaction service that allows an application to engage in a request-response interaction with another application. A transaction service can be used, for example, to support a remote procedure call mechanism. The SAFENET standard specifies services for unicast transactions. How an application uses the transaction services is regarded as a local matter.[26]

The Ada binding for transaction services is discussed in Section 11 and the Ada package specification appears in Appendix H.

[26]For example, remote procedure calls require a method for encoding the information sent in a transaction. Various means are available to perform the encoding, but this is beyond the scope of the Ada binding. It is expected, however, that the way in which transaction services could be used in an application will appear as part of the planned SAFENET handbook.

4.8. Event Management

A characteristic need for real-time systems is the ability to respond to asynchronously generated events. The API contains an event management capability that allows an application to register for an event. Events are defined in terms of classes and subclasses. An example of a class is the set of events associated with groups, while a subclass within this class would include events related to the group as a whole, events related to a member of a group, or events generated upon recognizing an error.

The Ada binding for event management is discussed in Section 12 and the Ada package specifications appear in Appendix I.

4.9. Buffer Management

There are different techniques for passing data between the Ada binding and the implementation. This package provides buffer management operations that allow buffers to be passed by pointers. This package allows an application to create buffer *pools* from which actual data buffers are allocated. An application can also specify the boundary alignment required for a buffer.

The Ada binding for transaction services is discussed in Section 13 and the Ada package specification appears in Appendix J.

4.10. Performance Measurement

A performance measurement capability has been included in the Ada binding. This allows applications to select a class of operations, such as unicast or multicast data transfers. An implementation then records information about activities associated with that particular class. This package defines minimum functionality; an implementor has freedom to define what information is recorded. An implementor is free to add procedures in the visible part of the package to define additional performance measurement capabilities.

The Ada binding for performance measurement is discussed in Section 14 and the Ada package specification appears in Appendix K.

4.11. Relation to Application Program

The packages that comprise the API to the SAFENET lightweight application services are necessary, though not sufficient, for an application program to communicate with another application. The Ada binding does not specify any interactions with layers below XTP, which would be necessary in a system implementation.

To illustrate the way an application would use the API, consider the case in which the layer below XTP is FDDI. In this case, two additional bindings would be necessary, for example

- *FDDI station management*: Reference [ANSI91] is a standard for FDDI station management. The FDDI station management standard is an extensive specification that (1) defines frame formats and protocols used to manage a ring, (2) provides for remote management of nodes through status report frames and parameter management frames, and (3) provides for ring management. Part of parameter management includes the ability to define timers, such as the token rotation timer. This, and other timers, are required for an FDDI network.
- *Target dependencies*: The FDDI station management standard, while specifying most of the services required, is not complete, particularly when one considers hardware implementations. There are, for example, several manufacturers of FDDI chipsets. Each manufacturer can have a different mechanism for allocating on-chip memory to the different queues implied in the FDDI standard. The manufacturers can also differ in what and how certain interrupts can be generated. To use a particular chipset, the application will need to be able to access certain functions provided by the chipset not provided by the FDDI station management standard.

There are two other functions indicated in the right-hand side of Figure 4-2. These include the following support:

- *Network management:* There is no requirement in the SAFENET standard for network management when only the lightweight protocol suite is used. In this case, network management is viewed as a local matter.
- *Time services:* The SAFENET standard specifies certain time-related services, such as the service to return the current time (in a SAFENET predefined format). The standard also specifies a protocol for distributed time management operations. The use of time services specified in the SAFENET standard is required of all SAFENET implementations.

To summarize, Figure 4-2 shows the Ada binding for the SAFENET lightweight application services supplemented by bindings for FDDI, network management, and SAFENET time services. Note that the Ada binding provided in this report does not invoke any of the services shown in the right-hand side of Figure 4-2. For example, the Ada binding represents objects of type *time* in terms of Ada type *duration* rather than the type specified in the SAFENET time services.

It is noted that the SAFENET lightweight application services could operate with different implementations of lower layers of the protocol stack, FDDI being one example. Another possibility would be where the lower layers in the protocol were those defined by the IEEE 802.6 standard [IEEE90]. This is possible because the IEEE 802.6 standard also interfaces with logical link control (LLC) [IEEE85], as indicated in Figure 2-1. One would hope that the Ada binding presented here could be appropriately modified for use in an IEEE 802.6 subnetwork with minimal effort. The ability to achieve this would be due, in part, to the reuse expected of designs based on Ada.

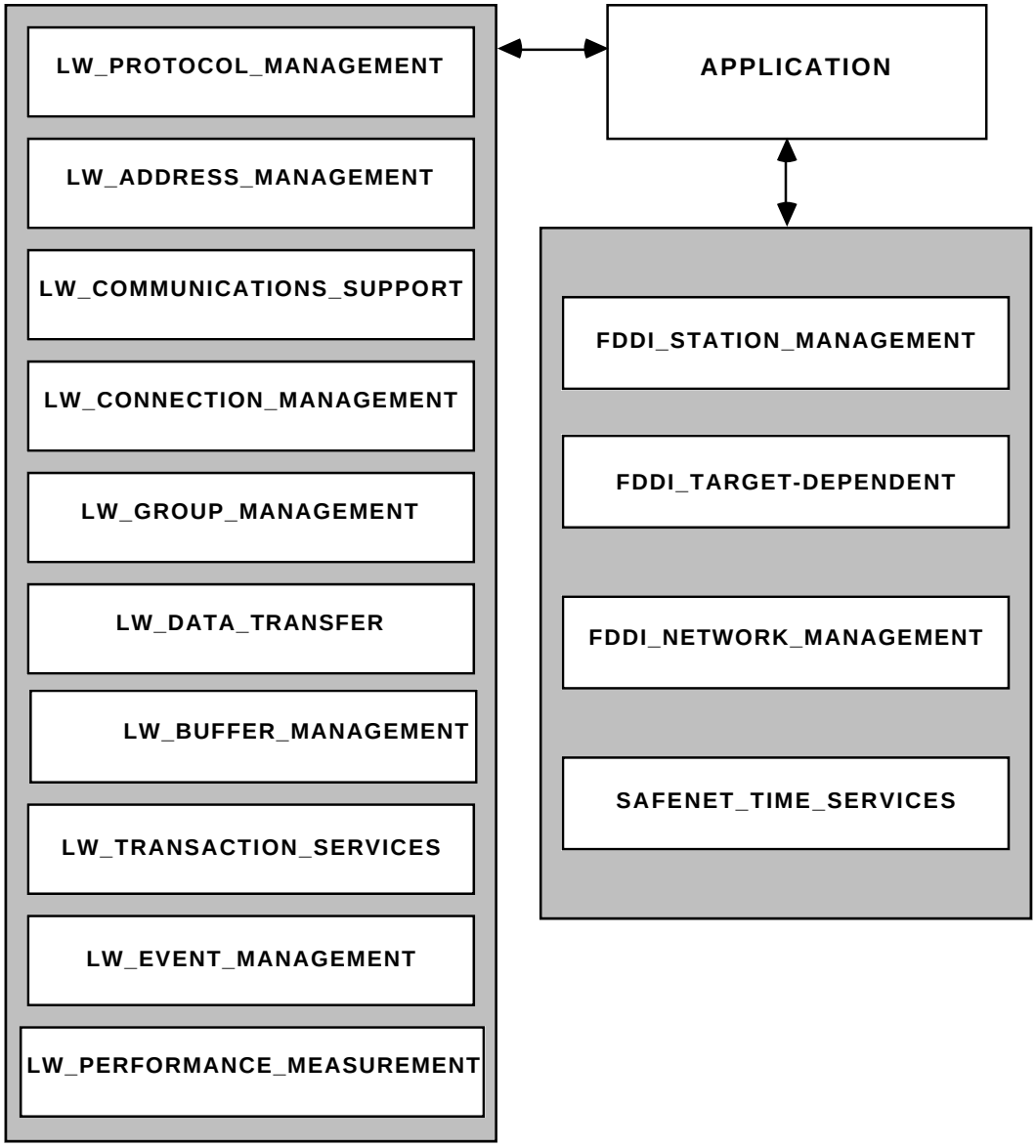


Figure 4-2: Application Relationship to API in FDDI Configuration

5. Protocol Management

This section discusses the protocol-dependent element of the Ada binding. The package specification appears in Appendix B on page 73.

5.1. SAFENET Specification

The SAFENET standard provides no specifications for the management of the lightweight protocol suite: such management is viewed as a local matter.

5.2. API Considerations

The package *LW_Protocol_Management* exports functionality related to the underlying protocol used in the lightweight suite. It is important to emphasize that this package is the only package in the Ada binding dealing with protocol-specific parameters and functionality. We believe that it would be fairly straightforward to modify this package to support other protocols.

The specification of this package contains implementation dependencies that are discussed below. In particular, the specific parameters associated with the protocol can be defined by an implementor. In the case of XTP, in addition to the local MAC and network addresses, reference [PE92] includes the following as default parameters:

- The initial assumption for the round trip time.
- The sender's initial maximum packet burst per context.[27]
- The sender's initial maximum data rate in bytes per second.
- The maximum size of output queue per context.
- The maximum size of input queue per context.
- The default connection timeout value.
- The default connection activity timer value.
- The initial value for the acknowledgement wait timer.
- The default allocation permitted by the receiver.
- The default allocation assumed by the sender.
- The retry count associated with a connection.
- The application defined message class (*btag*).

The above parameters are collected in a record that can used by the procedures defined in the package. An implementor is free to define the specific parameters related to an implementation. Each procedure that operates on the parameters is provided with the local MAC

[27] A context is the XTP terminology for the set of variables associated with an instance of XTP at a particular endpoint.

address as well as the network address. The functionality exported by this package includes the following:

- *Initialize_LW_Protocol:* Provides the parameters used by XTP.
- *Terminate_LW_Protocol:* Allows an application to terminate the lightweight protocol.
- *Update_Protocol_Parameters*: Allows an application to modify one or more parameters.
- *Return_Protocol_Parameters*: Allows an application to obtain the current values of the parameters.
- *Protocol_Parameters_Defined:* May be used to determine if the required protocol parameters are defined.

Recognition of error conditions is reported by a parameter of type *status*. If an application attempts to define a parameter that is not compatible with the implementation of XTP (such as a negative value for a timer) the value of *status* will be returned as *Invalid_Protocol_Parameter*. A value of *Invalid_Protocol_Operation* will be returned if there is an attempt to perform an operation that is illegal with respect to the current protocol state. For example, an attempt to terminate the protocol is invalid unless the current protocol state is initialized. Similarly, it is illegal to initialize the protocol if the current protocol state is *Initialized* (that is, the protocol must be terminated before it can be reinitialized).

5.3. Implementation Considerations

The following implementation dependencies apply to this package:

- An implementor has freedom in defining the elements of the record *LW_Protocol_Data*. The implementation must document the meaning of all parameters, their units, and acceptable range of values.
- An implementor can include additional subprograms to perform operations on the defined parameters. For example, an implementor can define a function to return the current values of each parameter.

5.4. Issues Considered

There are no significant issues associated with this package.

6. Address Management

This section discusses address management support in the Ada binding. The package specification appears in Appendix C on page 77.

6.1. SAFENET Specification

The SAFENET standard does not define services specific to address management. It is clear, however, that address management functionality is required by applications. We have therefore defined a package *LW_Address_Management* that is a part of a directory services function.

Addresses are referenced in the SAFENET standard and there are requirements that a language binding must satisfy. These requirements for the lightweight protocol are as follows:

- Physical addresses must be in the ISO standard format (see [ISO88]). These addresses must be unique throughout a system.
- Certain addresses are reserved for globally administered groups. Two examples of this are a broadcast address and an address reserved for the SAFENET time services.

6.2. API Considerations

It was believed warranted to define a separate package for the address management function for the lightweight protocol stack. The subprograms exported by this package are as follows:

- A procedure *Bind* establishes a mapping between the specified logical name and physical address. An identifier is returned to the application for use in subsequent data transfer operations.
- Procedures are defined to remove address bindings. *Unbind* will remove a logical name and physical address pairing. *Unbind_Logical_Name* will remove all pairings of the specified logical name and associated physical addresses. *Unbind_Physical_Address* will remove all address pairings containing the specified physical address.
- The functions *Logical_Name_Is_Bound* and *Physical_Address_Is_Bound* return *true* if and only if the specified name or physical address is bound. The function *Address_Binding_Exists* returns *true* if and only if a logical name is bound to a physical address.
- The function *Number_of_Addresses_Bound* returns the number of currently bound address pairs.

Recognition of an error is achieved by returning an element of the type *Status* that indicates the error that was recognized. The reasons for the possible errors are self-explanatory.

6.3. Issues Considered

The following issues were considered in the development of this package:

- The definition of a logical name must be visible. A logical name is defined in the package *LW_Communications_Support* as an Ada string type (except the blank character must not be permitted). We considered restricting the logical name to be composed of printable characters (such as the ASN.1 printable string), but felt the Ada character set was suitable.
- It is necessary to define a physical address. This was chosen to be an (unconstrained) array of unsigned bytes.
- The Ada binding does not assume that there is a unique mapping between a logical name and a physical address. For example, several logical names may be bound to the same physical address.
- A procedure was considered that would translate a physical address into an ASCII string. However, the representation of the resulting string was deemed to be application-dependent; hence, this functionality was not included.
- An *iterator* function was considered that would return address bindings on successive calls. The application would initialize the iterator and be able to terminate it before normal completion. However, the iterator was discarded because: (1) it presumes the application does not know logical addresses, and (2) it was not clear what the preferred mechanism for selecting the return from the iterator should be (for example, one could iterate over logical names in an alphabetical manner, or according to the time that the logical name was bound.)

The following are several open issues that should be considered:

- There may be a need to specifically define the primary and secondary MAC physical addresses in a dual-MAC configuration and to be able to operate on them (that is, get the physical address of the primary MAC). We have not included such functionality because the dual-MAC requirement has been removed from the SAFENET standard.
- Certain physical addresses in SAFENET are predefined, such as the address for the SAFENET time services. It may be desirable to also predefine a logical name for these services and then effect the binding to the required physical address upon package elaboration. Note that this would require SAFENET to specify a logical name not currently defined by the standard. The package specification does not assume that any physical address is reserved and the user is free to define a logical name for the different reserved addresses.
- The fact that a given logical name may be bound to multiple physical addresses adds complexity to an implementation. It may be appropriate to revisit this position and require a unique logical name to physical address mapping.
- An application may need to maintain timing information associated with physical addresses, such as the last time of reception from some physical address. It was believed that this, and similar functionality, should be provided by the application rather than by the interface binding.

7. Communications Support

7.1. SAFENET Specification

The SAFENET standard does not explicitly refer to the functionality provided by this package. The Ada package specification appears in Appendix D on page 81.

7.2. API Considerations

The package *LW_Communications_Support* contains type declarations and subprograms used by other packages in the Ada binding. The types declared serve the following purpose:

- A logical name is declared as an array of type *character*. The length of the name is an implementation-defined constant. Note that this is the single definition of a logical name used to declare connection identifiers, group names, and logical (address) names. For application portability, the following requirement is placed upon implementations of the API for case sensitivity of logical names: an implementation shall accept a logical name regardless of case; however, when a logical name is returned (as an Ada *out* parameter), the implementation shall return a name that is upper case.
- A data buffer is declared as an array of unsigned bytes. This is then used to define buffers for both connectionless and connection-oriented data transfers. In each case, the maximum buffer length is an implementation-defined constant.
- A message class is declared as a non-negative integer. The application can associate a message class with a particular type of data. Messages then can be distinguished based on their associated message class.
- A record is declared whose components are required for use in data transfers, such as connection-oriented transfer. The components of this record are defined as follows:
    - *Checksum* is a Boolean that is *true* if and only if the application requests a checksum.
    - *Error* is of type *Error_Control* and can assume the values *None*, *Reliable*, and *Aggressive*.[28]
    - *P* is of type *priority*, with values in the range 0 .. 255, with a larger value indicating a higher priority. The scheduling policy in the Ada binding is

[28]The reason for the inclusion of *Aggressive* error control is to permit applications access to the FASTNAK bit in XTP; see reference [PE92].

based on priority.[29]

- *Timeout* is of type *duration* and bounds the approximate amount of blocking that an application will accept. Timeout values are also used in support of asynchronous transfers.
- *Mode* is of type *Activity*, with values *Asynchronous* and *Synchronous*.
- *Suspend* is a Boolean that is *true* if and only if the calling task will be suspended during the performance of the activity.
- *Lifetime* is of type *duration* and represents the approximate lifetime of a message in the network.

The above parameters are collected in a record of type *Activity_Block*. A procedure is provided to create an activity block and return an *Activity_Block_ID*. Subprograms are provided to perform operations on components of an activity block. For example, there are procedures to get and set the priority. We have included this functionality to allow an application to reuse an activity block for different transfers.

This package also supports asynchronous operations by defining the following types and procedures:

- A type *Activity_Index* is used to identify an asynchronous activity.
- A type *Activity_State* contains the elements *Success*, *In_Progress*, *Failed*, or *Unknown*.
- A procedure *Get_Asynchronous_Activity_State* returns the activity state for a specified activity index.
- A procedure *Cancel_Asynchronous_Activity* allows an application to cancel an asynchronous activity.
- A procedure *Wait_On_Asynchronous_Activity* allows an application to wait on an asynchronous activity. A timeout can be specified to bound the amount of time that the application will wait.
- A procedure *Cancel_All_Asynchronous_Activities* allows an application to terminate all its asynchronous activities.
- A procedure *Terminate_All_Transfers* closes all connections of an application in an immediate or graceful manner depending on the parameter supplied. All the application's data transfers, including unicast, multicast and transaction will be terminated.

[29] Readers familiar with the details of XTP will recognize that the *sort* field in XTP permits a 32-bit integer that is interpreted as a priority. The range we have chosen is based on (1) the recognition that Futurebus+ provides 256 levels of priority, and (2) the realization that more than 8 bits of priority does not increase the schedulability by any appreciable amount. Moreover, there are systems based on deadline scheduling policies, where the representation of time prioritizes requests to the network. This type of policy is not explicitly supported in the Ada binding, although there is nothing to prevent an application from mapping timing values onto integer priorities.

7.3. Issues Considered

There are no significant issues relevant to this package.

8. Connection Management

This section discusses connection management capabilities of the Ada binding. The package specification appears in Appendix E on page 87.

8.1. SAFENET Specification

Connection-oriented data transfer is a well-known technique in distributed systems. As specified in the SAFENET standard, after a connection is established, users can exchange messages with the ‘‘assurance that all messages will be delivered correctly and in the order in which they were submitted.’’ An application can simultaneously maintain multiple connections with other applications.

The SAFENET standard specifies two services that relate to the management of connections. The first allows an application to establish a connection with a remote user. The service includes source and destination logical names as well as an identifier for the connection. The quality of service parameters include *selectable error control*, *throughput*, *maximum latency*, and *priority*. A request to establish a connection can also include an optional message to be transferred in the opening of the connection.

The second service is a *disconnect* service that can be used to terminate processing for a previously defined connection. A request to terminate a connection includes an identifier for the connection and an optional termination mode. The mode parameter can be either graceful or immediate. In the first case, any pending data transfers will be completed before the connection is terminated. In the second case, the connection will be terminated with no attempt made to complete any pending data transfers. The *disconnect* service can also include a priority. The response to a *disconnect* request includes the connection identifier, as well as a reason indicating why the connection is being closed. The SAFENET standard does not specify any reasons that are applicable to the closing of a connection.

8.2. API Considerations

The service definitions appearing in the SAFENET standard were followed closely in the development of the Ada binding. Additional functionality, believed useful to an application, is also included. The procedures that open and close connections can be invoked in either a synchronous or asynchronous manner. In each case, a timeout can be included to bound the blocking that an application can experience. This design approach was discussed in Section 3.5.1. The procedures declared that directly relate to the SAFENET standard are

- *Open_Connection*, which permits an application to establish a connection with a specified destination. When the connection has opened, a connection identifier, defined as a private type, is returned. This identifier is then used to refer to a particular connection, such as for data transfers. The application can specify the characteristics of the connection (such as flow control) by use of a parameter of the type *Connection_Block*.

- *Open_Connection_With_Data* is similar to *Open_Connection* except that an application can include a message to be transmitted as part of connection establishment. This procedure is overloaded to allow the application to specify a message class for the associated data.
- *Close_Connection* can be used to close a specified connection. Either a graceful or immediate close can be specified.
- *Close_All_Connections* can be used to close all existing connections in a specified manner (graceful or immediate).

The additional functionality exported by this package includes the following:

- A connection is modeled as a state machine and the function *Get_Connection_State* returns the state of a specified connection. The following states are defined:
  - *Unknown*
  - *Open_Request_Sent*
  - *Open_Response_Pending*[30]
  - *Open*
  - *Graceful_Close_Sent*
  - *Immediate_Close_Sent*
  - *Graceful_Close_Received*
  - *Immediate_Close_Received*
- A procedure *Accept_Connection* allows an application to accept a request to open a connection from a specified logical name.
- A function *Number_Of_Connections* returns the number of connections that are in a specified state. The function definition is overloaded to also return the total number of connections.
- *Connection_Is_Known* returns *true* if and only if a specified connection is known to the underlying implementation.
- A function *Buffer_Length_Is_Valid* returns *true* if and only if a specified value is acceptable to an implementation. This function may be used as an inquiry function if an error condition is recognized.

8.3. Issues Considered

The following issues were considered in the development of this package:

- Certain parameters are associated with a connection, such as whether or not checksums are to be enabled in a data transfer. These parameters are imported from the package *LW_Communications_Support*.
- Inclusion of a state *Closed* in the state model for a connection was considered.

[30] This state is entered as a result of a call to the procedure *Accept_Connection*; see below.

This was discarded for it would require that an implementation maintain information about a connection.

- Further refinement of the notion of the open state in the state model of a connection was considered. In particular, we considered replacing the open states with states *Idle*, *Send_in_progress*, *Receive_in_progress*, and *Blocked*. The *Blocked* state could also be refined to account for *blocked on receive*, or *blocked on send*. We did not include these however, because it was not clear how they would be used by an application in a timely manner. It is possible for an implementation to incorporate some of this information in an implementation-dependent manner. For example, an implementation could associate an event with blocking caused by flow control restrictions, and an application may respond as needed.
- It would have been possible to restrict the direction of data flow over a connection. For example, a parameter for opening a connection could be that the requested connection be *read only*. Although such an option could be useful to some applications, it was not included.

9. Group Management

Group management, a subset of directory services, provides support for an application in multicast operations. The group management functionality has been implemented as a separate package discussed below. The Ada package specification appears in Appendix F on page 93.

9.1. SAFENET Specification

The SAFENET standard specifies four functions associated with group management, which give an application program the ability to open or close a group or to join or leave an existing group.

In addition, when opening a group, an application can restrict the membership in the group by including a membership list. The use of a membership list is an option. When a membership list is included, an application will not be permitted to join a group unless its (logical) name appears in the membership list.

9.2. API Considerations

The API for group management was implemented in a straightforward manner. The following procedures have a direct relationship with the services defined in the SAFENET standard:

- *Open_Group:* Allows an application to create a multicast group by providing the name of the group, a membership list, and a permission list. The permission list allows the application to specify which members of the group have permission to close the group. If no such permission is granted, only the creator of the group can delete the group.
- *Join_Group:* Allows an application process to join a previously defined multicast group.
- *Leave_Group:* Allows an application process to remove itself from a previously defined group.
- *Close_Group:* Allows a specified application process to remove a previously defined group. A notification parameter is included; if *true*, other members of the group will be notified when the group is closed.[31]

In addition to the above, the following procedures and functions have been included in the package specification:

- *Remove_Member_From_Group*: Allows an application to remove a member from the group. The need for this procedure is as follows. The SAFENET stan-

[31] This is an approved change in SAFENET.

dard [NGCR92b] specifies that the *join_group* service allows an application to add *itself* to a group. Although not quite clear, we assume that the *leave_group* service allows an application to remove itself from a group. In the presence of failure, there exists the need to remove a group member that is no longer available; hence, the need for the service.[32]

- *Number_of_Group_Members:* Allows an application to determine the number of members in specified group.
- *Obtain_Group_Members:* Allows an application to determine the number of members and obtain a list of those members of a specified group.
- *Get_Permission_of_Member:* Can be used to determine if a specified member of a group has permission to close the group.
- *Grant_Permission_To_Close:* Allows a member to have permission to close a group.

All subprograms in this package include a priority that is interpreted as the relative degree of urgency of the request. Also, each subprogram can be used in a synchronous or asynchronous manner.

9.3. Issues Considered

The following issues were considered in the development of this package.

- There are two basic methods by which group management can be implemented. In the first case, one can employ a *localized* model. In this case, there is no need to generate network traffic in support of group management. The second implementation choice is that of a *distributed* model, which does require network traffic for support of the group management. This package does not assume a particular model for the underlying implementation. However, all subprograms defined in this package include a timeout parameter to limit the amount of time for a response after a given request has been made. The timeout parameter therefore bounds the blocking experienced by an application as a result of a group management request. Note that a schedulability analysis for group management requires knowledge of implementation details, such as the model chosen as well as any message-generated processing.
- The name of a group is of type *Logical_Name*, defined in package *LW_Communications_Support*. When a group is opened, a group identifier is returned that is specified as a private type. To reduce lookup time, the group identifier is thereafter used to refer to the group.
- It should be noted that the SAFENET standard does not restrict the members of a group to only be individual processes. This allows, for example, a call to create a group to include members who are themselves groups. There are potential issues with such an option that could lead to ambiguities. We recommend that this issue be examined in the SAFENET community.

[32] If this functionality were not present, it could cause excessive retransmissions and result in possible errors caused by quality of service considerations. That is, if a request is made to send to all members of a group and a node fails (thereby removing a member), the resulting service will fail. We recommend that the SAFENET community examine this question in greater detail.

- It is appropriate to define the maximum number of groups that can exist. This is specified as an implementation-dependent constant.[33]
- It is appropriate to define the maximum number of members that can belong to a group. This is specified as an implementation-dependent constant.
- Set-theoretic operations on a group were considered, such as the ability to create a group from the composition of two other groups. Since it can be implemented by an application, this functionality was discarded.
- There are several issues that deal with notification of group members that warrant consideration. For example, if a member is added or removed from a group, should the other members of the group be notified? We have included these as options and recommend that the SAFENET standard be modified accordingly.

[33]The SAFENET standard states that an application shall be able to recognize ''a minimum of 65,536 group addresses.'' A possible interpretation of this statement is that an implementation is required to support 65,536 different groups. If the SAFENET statement refers to the number of bits in a group address, then that is a different issue. We recommend that this point be clarified in the SAFENET standard.

10. Data Transfer Services

The section describes the data transfer capabilities of the Ada binding. The package specification appears in Appendix G on page 97.

10.1. Unicast

10.1.1. SAFENET Specification

The SAFENET standard specifies two services for unicast data transfer. The first allows transfers over previously established connections. In this case, the only relevant quality of service parameter is *priority*.

The second service relates to connectionless transfers and the standard specifies a source and destination logical name. The relevant quality of service parameters are those for *selectable error control*, *priority* and *transit delay*. The service user can also specify a request identifier when errors are detected. The user can specify the use of either XTP or the OSI connectionless transport protocol with XTP as the default.

Note that the SAFENET specified service parameter *special-data* is interpreted as a *message class* (see 3.4).

10.1.2. API Considerations

The following routines are provided in the Ada binding for unicast data transfers:

- *Send_Connection_Message* transfers a message over a previously established connection.
- *Get_Connection_Message* receives a message over a previously established connection.
- *Send_Connectionless_Message* sends a connectionless data transfer to a specified destination.
- *Get_Connectionless_Message* receives a message from a specified source that was transmitted in a connectionless manner.
- *Get_Any_Connectionless_Message* receives a message from any source that was transmitted in a connectionless manner.

In addition, the following functionality is provided by this package:

- *Connectionless_Buffer_Length_Valid* is an inquiry function that returns *true* if and only if a specified buffer length is acceptable for connectionless transfers.
- *Messages_Pending_On_Connection* returns the number of messages waiting for a *get* operation for a specified connection. A similar function returns the number of connectionless messages waiting for a *get* action for a specified source.
- *Purge_Messages_On_Connection* will remove a specified number of messages waiting for a *get* operation for a particular connection. The procedure declara-

tion is overloaded to also remove messages that were received before a specified time. A similar capability is provided for connectionless transfers.

10.1.3. Issues Considered

The following issues were considered in the development of this package:

- Inclusion of a parameter in the procedures that get a message that would indicate the time of receipt of the message was considered. Such time-tagging is frequently needed by applications. It was believed, however, that time-tagging of messages is better performed by the application. Even if a time-tag were applied by an implementation, it is not clear what the time tag would mean (that is, the mechanism for determining the time value is unspecified), or which clock to use.
- A connectionless data transfer can be received using either the XTP or OSI connectionless transfer protocols. We considered including a parameter in the procedures that get a message that would return the protocol stack over which the message was received. It was not clear that this would be useful to applications, and it was therefore discarded.
- The ability to purge messages has been defined for messages that have been received, but not had a *get* operation applied to them. It may be relevant to include a function to purge messages that are waiting for transmission.
- Procedures that perform *get* operations also return the priority at which the message was transmitted. This functionality has been included to allow applications to dynamically set their priority to that of the received message (a form of priority inheritance based on message priorities). It was believed that support of this design model would be useful. We believe it is better to have it directly supported in the Ada binding, rather than have applications extract priorities from message buffers.

10.2. Multicast

10.2.1. SAFENET Specification

There are two services defined in the SAFENET standard for multicast data transfer services. The first service includes the logical name of the requester and the name of the group that is to receive the message. The quality of service parameters that are relevant are *selectable error control*, *transit delay*, and *message priority*. A parameter that uniquely identifies the request is required. This parameter is used to inform the user (in an asynchronous manner) regarding status or error information.

The second specified multicast data transfer service is for multicast stream. This service allows an application to establish a connection with a group and use that connection to send data to the members of the group. The quality of service parameters that are relevant are *selectable error control*, *maximum latency*, and *message priority*.

The SAFENET standard specifies that XTP will be the default provider of the service. However, the user can specify that the OSI connectionless transport protocol be used. The standard further states that in this case, it is the responsibility of the user to ensure that the message will satisfy the constraints of the underlying protocol.

Note that the SAFENET specified service parameter *special-data* is interpreted as a *message class* (see 3.4).

10.2.2. API Considerations

The development of the Ada binding for multicast operations closely follows the SAFENET standard. The data transfers can be performed in a synchronous or asynchronous manner, including a timeout specification. The basic support provided is as follows:

- *Send_Multicast_Message* allows an application to send a message to the members of a specified group. The user can specify the protocol used and the priority of the request.
- *Get_Message_For_Group* allows an application to obtain a message for the specified group.
- *Get_Message_For_Any_Group* allows an application to obtain a message for any group that the application belongs to.
- *Send_Multicast_Stream_Message* allows an application to send a message to the members of a specified group through a connection.

Additional functionality provided for the multicast case is as follows:

- *Messages_Pending_For_Group* returns the number of messages for a specified group that are present but not yet processed by the application.
- *Purge_Messages_From_Group* allows an application to remove a specified number of messages that are waiting for a *get* operation for a specified group. The procedure definition is overloaded to allow an application to remove messages that were received before a specified time.

Support for broadcast is provided separately. The following subprograms are provided:

- The procedure *Enable_Broadcast* allows an application to accept broadcast messages.
- The procedure *Disable_Broadcast* can be used to withhold acceptance of messages transmitted in a broadcast mode. The function *Broadcast_Enabled* returns *true* if and only if broadcast is enabled.
- The procedure *Purge_All_Broadcast_Messages* will either remove all broadcast messages for an application or all broadcast messages received before the specified time.

10.2.3. Issues Considered

The following issues were considered in the development of this package (note that some of the issues discussed in regard to unicast data transfer also apply to the multicast case):

- The logical name of the initiator of a multicast message could be provided, but it is not clear how to implement this. Hence, if such a capability is desired, it must be implemented in the application software.
- The need to purge messages that have not yet been transmitted is an open issue.

10.3. Use of Buffer Management

There are applications based on the use of buffer management for data transfers; in particular, the use of pointers. The Ada binding includes a buffer management package that can be used in conjunction with data transfer operations. The functionality discussed above is duplicated through the use of the buffer management package. For example, one can send a message over a connection using the procedure *Send_Connection_Buffer*. This procedure includes parameters that allow the use of pointers in the buffer transfer. We did not, however, consider the option of an implementation-defined buffer management package.

11. Transaction Services

This section describes support for peer-to-peer transaction services provided by the Ada binding. The package specification appears in Appendix H on page 111.

11.1. SAFENET Specification

The SAFENET standard specifies a unicast transaction service with the following characteristics:

- The application must specify the source and destination logical names.
- A request message must be included, and a response message is optional.
- A request identifier is included to identify the requester of an asynchronous event.
- A transaction identifier is included that provides a mechanism for the application to associate responses with requests.

The permitted quality of service parameters are *selectable error control*, *maximum latency*, and *priority*.

The SAFENET standard also indicates that the *notify* service can be used to inform the user if there are any problems with delivering the message within the specified quality of service parameters. This type of functionality is discussed in Section 12.

Note that the SAFENET specified service parameter *special-data* is interpreted as a *message class* (see 3.4).

11.2. API Considerations

The Ada binding for transaction services is based on the model discussed in Section 3.5.1 and provides for synchronous and asynchronous operation.

The following procedures support the unicast transaction service:

- *Initiate_Unicast_Transaction*, which corresponds to the request side of the transaction. This procedure contains the logical name of the recipient, information regarding selectable error control, and the data. A priority is included that indicates the priority of the process performing the transaction. The procedure can be invoked in either a synchronous or asynchronous manner. A transaction identifier is returned to the application that can be used to determine the result of the transaction.
- *Get_Unicast_Response* allows an application to obtain the response from a transaction, indicated by the transaction identifier. A timeout can be included, which, if present, will cause the application to block pending the result of the transaction.

- *Accept_Transaction_Request* allows an application to accept a request for a transaction from a specified source. The request can be made in a synchronous or asynchronous manner, and a timeout can be included.
- *Accept_Any_Transaction_Request* allows an application to accept a request for a transaction from any source. The request can be made in a synchronous or asynchronous manner, and a timeout can be included.
- *Respond_to_Transaction_Request* can be used after an application has received a transaction request.

The following additional subprograms are provided in support of an application:

- *Get_Transaction_Response* determines if a transaction response is present for a specified transaction identifier. The function can contain a timeout which, if present, will cause the application to wait pending receipt of the response.
- *Number_of_Transactions* returns the number of transactions waiting for a *get* operation by the application.
- *Cancel_Transaction* terminates a specified transaction.
- *Cancel_All_Transactions* terminates all transactions.

11.3. Issues Considered

The following issues were considered in the development of this package:

- The only scheduling mechanism used in the Ada package is that of priority; we have not provided for maximum latency as a scheduling mechanism. Section 3.3 discussed the rationale for this choice.
- The Ada binding makes no assumption about either the contents of transaction data, nor the mechanism by which data is encoded or decoded from a buffer. Such matters are application-dependent; for example, there are several different techniques available for encoding and decoding data.
- More elaborate models for managing the transactions could have been employed. In the current design, it is assumed that when a *get* operation is performed after a transaction completes, all information associated with the transaction is removed. This need not be the case; for example, the procedures that return the result of a transaction could include an optional parameter that indicates whether or not the information can be deleted. The issue of transaction management by an application is worthy of further investigation. Indeed, it is not hard to envision a package devoted to transaction management.

12. Event Management

This section describes the event management facility exported by the Ada binding. The package specification appears in Appendix I on page 115.

12.1. SAFENET Specification

The relevant SAFENET specification for the event management capability provided by the Ada binding is the *notify* service. This service is intended to provide information concerning errors detected in the transmission of messages to external users. In the case of multicast transfers, the *notify* service can be used to provide information on partial delivery of data to a group of users. The SAFENET standard also states that the only quality of service applicable to the *notify* service is *priority*.

12.2. API Considerations

In this section we will discuss the considerations relevant to the design of the event management package. Some general remarks about the design are presented, followed by discussions of the specific events handled.

12.2.1. General Remarks

The rationale and philosophy for the existence of an event handling capability was discussed in Section 3.5.3. The design of the event management package is based on a hierarchical scheme. At the highest level is the event. An event has an *event class* that represents the operation associated with the event. The event classes are *connection*, *group*, *unicast*, *multicast*, and *transaction*. Each event class has an associated *subclass* that describes the possible nature of the event. For example, a connection event has associated subclasses of *closed*, *fail*, and *error*. Finally, for each class and subclass there is a set of reasons that the event was generated. In the case of a closed connection, the possible reasons are that the connection was closed in a graceful manner or the connection was closed in an immediate manner. The term *entity* refers to the instance of the object associated with an event. For the class of events dealing with connections, the corresponding entity would be the name of the connection.

The following are general remarks about the design of this package:

- When an application registers for an event, the priority can also be included.
- An application can process events in a prioritized (based on the event's priority) or a FIFO manner.[34] Furthermore, if a FIFO access method is used, an application can require either the oldest or the latest event to be returned. All three of these access mechanisms are provided.

[34]Note that using priorities when a queue is processed in a FIFO manner introduces priority inversion.

- Procedures are supplied to register or deregister for an event. This can be done on the basis of a class; a class and subclass; or a class, subclass and entity.
- Functions are provided to return the number of events pending for either a class; class and subclass; or class, subclass and entity. A timeout can also be specified, indicating that the application can wait for a specified time interval for the presence of the event.
- Functions are provided to return the number of events pending in a class; class and subclass; or class, subclass and entity.
- The data associated with an event is maintained in variant records. The data always contains the time of the event, followed by event-specific information. Procedures are available to obtain the data for an associated event based on class; class and subclass; or class, subclass and entity. A parameter can be supplied by the application to delete the event after delivery to the application. A timeout can also be included, indicating the amount of time an application can wait pending receipt of the event data.

The following subsections provide a brief overview of each event class:

12.2.2. Events Related to Connections

An application registers and receives information regarding a connection. The following subclasses are defined: (1) a connection has closed, (2) a connection has failed, and (3) an error has occurred.

The reasons associated with the closing of a connection are (1) the connection closed in a graceful manner, and (2) the connection closed in an immediate manner.

The reasons associated with a connection failure are (1) the connection timed out, and (2) some other reason.

The reasons associated with a connection error are (1) the connection has been reset, (2) the *read* side of the connection has failed, (3) the *write* side of the connection has failed, (4) a request to establish a connection has been refused, (5) the reason for the connection failure is unknown, and (6) some other reason.

Note the distinction between a connection *fail* and a connection *error*. In one sense, a failure of a connection is, of course, an error. However, when a connection fails it is represented as a *hard* failure, whereas the subclass for errors can be viewed as a *soft* failure. The distinction is made because error recovery can be a function of the event sub-class.

12.2.3. Events Related to Groups

The processing for group-related events is similar to that discussed above for connection-related events. An application can register a group in one of the following subclasses: (1) group-related, (2) member-related, and (3) error-related.

The subclass of group-related events includes events that affect the group as a whole. The

possible reasons associated with a group-related event are (1) the group has closed, (2) the group has no members, and (3) the group is full.

The subclass of events related to a particular member of a group are (1) a member has been added to the group, (2) a member has been deleted from the group, and (3) the privilege (to delete a group) of a member has changed.

The subclass of events related to a group error are (1) a group name is invalid, (2) a member name is invalid, (3) the group does not exist, (4) insufficient permission exists to remove a member of the group, (5) insufficient permission exists to join a group, (6) the system is unable to add a member to a group, and (7) some other reason.

12.2.4. Events Related to Unicast Data Transfers

The events associated with a unicast data transfer are based on the subclasses related to connection-oriented transfers, connectionless transfers, and general errors.

The reasons associated with connection-oriented transfer are (1) invalid service parameter, (2) unknown connection, (3) invalid buffer length, and (4) some other reason.

The reasons associated with connectionless transfers are (1) an unknown name, (2) invalid protocol suite, and (3) some other reason.

The reasons associated with errors are (1) an invalid name, and (2) some other error.

12.2.5. Events Related to Multicast Data Transfers

The events associated with multicast data transfer are based on the three subclasses (1) number delivery, (2) named delivery, and (3) other errors. The first two classes provide information about the recipients of a multicast operation.

Note that number delivery and named delivery do not have reasons associated with them. Rather, associated with number delivery is the number of members of a group that received a multicast message, while associated with named delivery are the members of the group that did not receive the multicast data transfer.

The reasons associated with an error are (1) invalid buffer length, (2) invalid group name, (3) unknown group, (4) invalid protocol, and (5) some other error.

12.2.6. Events Related to Transactions

The events related to transactions have only the error-related subclass.

12.3. Implementation Considerations

The implementation dependencies of this package are as follows.

- Different implementations can define different error conditions. Hence, an implementation can modify the package specification accordingly. The following provisions apply to modification of the package specification:
    - An implementor cannot change the subclass for a set of events. For example, the *Connection_Subclass* must include only the elements *closed*, *fail*, and *error*.
    - An implementor can change the elements of the enumerated type identifying the reasons associated with a particular subclass. For example, the Ada binding defines a type *Connection_Fail_Reason* as including the members *Timeout* and *Other*. An implementor can modify the members of the enumerated type to include other reasons. A natural consideration is to include more specific reasons for the generated connection failure event.
    - An implementor may not change the event-related data returned to an application.

  An implementor is free to include a method by which the recognition of multiple events is made available to an application.[35]

12.4. General Issues Considered

Several general issues applying to the packages discussed above are as follows:

- The packages do not contain processing for an *unregistered* event. A natural question is: What should an implementation do if it recognizes an unregistered event? The choice made here is that the implementation can disregard the event. Other design choices are possible. An implementation could provide an option for the application to enable or disable the processing of unregistered events. Another possibility is for the implementation to raise an exception when an unregistered event occurs.
- An open issue is if a provision should be made to dynamically change the priority of an event. We can envision the need for such a facility, for example, in the case of a mode change.

[35] One technique to accomplish this is to have a subclass that includes *Multiple_Event*. A procedure, say, *Get_Multiple_Connection_Event* can be called to return an indication of which events were generated simultaneously. The actual manner in which the data are returned could be either a stack, or some bit encoding in a record type.

- Another open issue is the level of detail specified when an error is detected.[36] This issue requires further study. The result may depend on the envisioned manner in which an application would respond to detailed error information.
- The Ada binding incorporates into one package the event management for each event class. It may be worthwhile to have multiple packages, one for each class. It did not seem worthwhile to include the event management for a given class within the Ada package for that class (for example, including event management for connections in the connection management package).

[36]For example, some of the reasons specified in reference [ISO87a] for a connection error include: a disconnection, a delineation of a connection rejection because of either an unknown network service access point (NSAP) address, the NSAP is unreachable, quality of service is not available, or an unspecified reason. These reasons are overloaded to also indicate a transient or permanent condition. Real-time mission-critical systems can require knowledge of transient, as opposed to permanent, errors. That is, the response to an error can depend on whether the error is transient or permanent.

13. Buffer Management

This section describes the support for buffer management operations in the Ada binding. The package specification appears in Appendix J on page 127.

13.1. SAFENET Specification

The SAFENET standard does not contain any specification relevant to buffer management operations.

13.2. API Considerations

It is recognized that there are applications that require buffer management operations; in particular, the ability to pass data through the use of pointers. This package is designed to provide such a capability. Its use is optional; that is, messages can be transferred using a standard Ada mechanism or by the use of buffer management.

The following model was adopted for buffer management operations. An application can create and delete a *pool* of a specified size. An actual buffer, such as used in message transfer, is then allocated from a specified buffer pool. When a buffer is allocated, the application is returned a pointer to the buffer. In addition, an application can specify that a buffer be allocated on either a byte, 16-bit, 32-bit, or 64-bit address boundary.

The following subprograms are included in this package:

- *Initialize_Buffer_Management* must be called by an application before any buffers can be allocated.[37]
- *Allocate_Pool* creates a pool of a specified size (in bytes).
- *Deallocate_Pool* removes a previously allocated pool. A pool can be released in either a graceful or immediate manner. In the first case, the release will not be performed if there are any outstanding buffers allocated from the pool. In the second case, the pool will be released regardless of the allocation state of any buffers in the pool. These release mechanisms are provided in the event of mode changes.
- *Increase_Pool_Size* increases the storage allocation associated with a specified pool.
- *Decrease_Pool_Size* decreases the storage allocation associated with a specified pool.
- *Buffer_Pool_Size* returns the current size of a specified pool (in bytes).
- *Buffer_Space_Allocated* returns the amount of storage currently allocated from a specified pool.

[37] The reason for requiring that the procedure be called is discussed in Section 13.3, below.

- *Buffer_Space_Available* returns the amount of storage that has not been allocated from a specified pool. A pointer to the buffer is returned to the application.
- *Get_Buffer* allocates a buffer of a specified size and boundary alignment from a specified pool.
- *Release_Buffer* frees a previously allocated buffer.

13.3. Implementation Considerations

This package contains one implementation-dependent operation. An implementation may modify the procedure *Initialize_Buffer_Management* by including parameters in the procedure definition. This option is provided for cases in which specific addresses are required from which buffers will be allocated.[38]

An implementation is required to allocate buffers in a physically contiguous manner.

13.4. Issues Considered

The following issues were considered in the development of this package:

- We considered including a Boolean parameter to indicate if buffers should be allocated in a physically contiguous manner. However, our experience in dealing with real-time systems indicates that this should be a requirement of an implementor (especially for use with direct memory access (DMA)).
- The utility of the procedures to increase and decrease pool sizes is questionable. There are other techniques to accomplish the same result (for example, rather than increase a pool size, an application could open a new pool). These procedures have been included, however, for those applications that require such functionality, and can have utility in limited memory management systems (for which opening a new pool would not be an acceptable option).
- Other possible alignments for buffer allocation, such as page alignment, were considered.

[38]We would not expect, in general, that the options would be available to the application software. Such information would be handled by an implementation rather than the application.

14. Performance_Measurement

This section describes the performance measurement capabilities specified by the Ada binding. The Ada package specification appears in Appendix K on page 131.

14.1. SAFENET Specification

The SAFENET standard does not refer to the functionality provided by this package.

14.2. API Considerations

This package is based on two classes of activities. The first class is related to data transfer and includes as members: broadcast, connectionless transfer, connection-oriented transfer, multicast, and transaction. Within this class there is a *Mode* that can assume the values *send*, *receive*, or *both*. Using the transfer class and mode, an application can enable the recording of information associated with receipt of multicast transfers, for example. The subprograms defined for this class are as follows:

- *Enable_Measurement* enables logging of information for a specified transfer class and mode.
- *Disable_Measurement* disables logging of information for a specified transfer class and mode.
- *Measurement_Enabled* returns *true* if logging is enabled for a specified transfer class and mode.

The second class of activities are those related to the events defined in the package *LW_Event_Management*. The elements of this class are: connection, group, unicast, multicast, and transaction. The following subprograms support the logging of information for this class.

- *Enable_Events* enables the logging of information for a specified event class.
- *Disable_Events* disables the logging of information for a specified event class.
- *Events_Enabled* returns *true* if and only if logging is enabled for the specified event class.

Note that the performance measurement operations are defined on a global basis. That is, if performance measurement is enabled for connection-oriented transfer, then data will be logged for *all* connections (for that application).

14.3. Implementation Dependencies

The following implementation dependencies apply to this package:

- An implementation can specify what data is recorded for a specified class and mode combination.
- An implementation is free to define the mechanism by which the data is recorded, such as an internal buffer or external file.
- An implementation is free to define additional subprograms in the package specification for the manipulation of recorded data.

14.4. Issues Considered

The following issues were considered in the development of this package:

- We considered specifying a more detailed interface to the performance measurement function. For example, it is possible to specify that performance measurement be enabled for a particular connection, as opposed to all connections. We left this as an implementation option.
- It is intended that this package will be used more during application development rather than as part of delivered software.

The Ada binding does not specify guidelines for what information is recorded as part of performance measurement. It would be worthwhile to develop such guidelines, recognizing that they would be of assistance to other application programs and developers.

Application developers must recognize that the implementation of this package will affect performance. For those applications that require non-invasive performance measurement, it is recommended that hardware procedures be considered.

15. Summary

This document provides a draft Ada binding to the SAFENET lightweight application services. The document contains a rationale that explains the choices that were made, as well as issues that were addressed. A motivating factor throughout the development of the Ada binding is that the resulting binding should enable applications to develop systems that are schedulable. Hence, the Ada binding provides a mechanism to bound the blocking that an application can experience. In addition, support is provided for both synchronous and asynchronous transfers. The Ada package specifications for the interface to the lightweight application services are included in the report.

References

[ANSI91] American National Standards Institute.
FDDI Station Management, Draft ANSI Standard X3T9.5/Rev 6.2
1991.

[COS88] Corporation for Open Systems.
Manufacturing Automation Protocol Specification, Version 3.0
1988.

[IEEE85] Institute of Electrical and Electronics Engineers.
Logical Link Control, ANSI/IEEE Standard 802.2, ISO/DIS 8802-2-1985.
New York, New York: August, 1985.

[IEEE89] Institute of Electrical and Electronics Engineers.
Futurebus+: System Layer Specifications, P896.2/D3.0.
New York, New York: June 15, 1989.

[IEEE90] Institute of Electrical and Electronics Engineers.
Distributed Queue Dual Bus (DQDB) Subnetwork of a Metropolitan Area Network (DRAFT), IEEE P802.6/D15.
New York, New York: October, 1990.

[IEEE91a] Institute of Electrical and Electronics Engineers.
POSIX: IEEE Standard Portable Operating Systems Interface for Computer Environments
New York, New York: March, 1991.

[ISO86] International Organization for Standardization.
Information processing systems - Open Systems Interconnection - Connection oriented transport protocol specification
Switzerland: ISO/IES Copyright Office, 1986.

[ISO87a] International Organization for Standardization.
Information Processing Systems - Data Communications - Network Service Definition, ISO 8348.
Switzerland: ISO/IES Copyright Office, 1987.

[ISO87b] International Organization for Standardization.
Information Processing Systems - Open Systems Interconnection - Protocol for providing the connectionless-mode transport service
Switzerland: ISO/IES Copyright Office, December 1987.

[ISO88] International Organization for Standardization.
Information Processing Systems - Data Communications - Network Service Definition- Addendum 2: Network Layer Addressing
Switzerland: ISO/IES Copyright Office, March 1988.

[ISO89a] International Organization for Standardization.
Information Processing Systems - Fiber Distributed Data Interface (FDDI) - Part 1: Token Ring Physical Layer Protocol (PHY); ISO 9314-1
Switzerland: ISO/IES Copyright Office, 1989.

[ISO89b] International Organization for Standardization.
Information Processing Systems - Fiber Distributed Data Interface (FDDI) - Part 2: Token Ring Media Access Control (MAC); ISO 9314-2
Switzerland: ISO/IES Copyright Office, 1989.

[ISO89c] International Organization for Standardization.
Information Processing Systems - Fiber Distributed Data Interchange (FDDI) - Part 3: Token Ring Physical Layer, Medium Dependent (PMD); ISO 9314-3
Switzerland: ISO/IES Copyright Office, 1989.

[ISO89d] International Organization for Standardization.
Information processing systems - Open Systems Interconnection - Connection oriented transport protocol specification - Addendum 2: Class 4 operation over connectionless-mode network service
Switzerland: ISO/IES Copyright Office, September 15, 1989.

[NGCR92a] Navy NGCR Program Office.
MIL-STD-2204 (DRAFT): Survivable Adaptable Fiber Optic Embedded Network (SAFENET)
1992.

[NGCR92b] Navy NGCR Program Office.
MIL-HDBK-818-1 (DRAFT): Survivable Adaptable Fiber Optic Embedded Network (SAFENET)
1992.

[PE92] Protocol Engines Inc.
XTP Protocol Definition, Revision 3.6
1992.

Appendix A: Application Interface Preamble

```
------------------------------------------------------------------
--
-- MODULE: LIGHTWEIGHT PROTOCOL SUITE
--
-- Purpose: This Module provides the application interface to
--      the SAFENET lightweight application services. The
--      following packages are provided:
--
--
--        o  LW_ADDRESS_MANAGEMENT: Provides management of
--              logical and physical addresses.
--
--        o  LW_BUFFER_MANAGEMENT: Provides a basic buffer
--              management capability.  Storage allocation
--              of buffers is done within buffer pools that can
--              be created and deleted by an application.
--
--        o  LW_COMMUNICATIONS_SUPPORT: Exports types used by
--              the connection management and data transfer
--              packages.  This package also provides
--              asynchronous operations.
--
--        o  LW_CONNECTION_MANAGEMENT: Provides for management
--              of connections used in data transfers.
--
--        o  LW_DATA_TRANSFER_SERVICES: Provides support for the
--              communication between distributed applications.
--              The services provided are unicast, multicast,
--              and broadcast data transfers. Unicast operations
--              can be either connection-oriented or
--              connectionless.
--
--        o  LW_EVENT_EVENT_MANAGEMENT: Allows an application to
--              register and subsequently receive an event. The
--              event management facility also supports error-
--              handling capabilities.
--
--        o  LW_GROUP_MANAGEMENT: Provides for management of
--              groups used in multicast data transfers.
--              Functions are provided to create and delete a
--              group, or to add and remove a member of a group.
--
--        o  LW_PERFORMANCE_MEASUREMENT: Allows an application
--              to obtain performance data from the underlying
--              implementation.
--
--        o  LW_PROTOCOL_MANAGEMENT: Provides the protocol-
--              dependent operations for the lightweight stack,
```

```
--              particularly those related to XTP.
--
--        o  LW_TRANSACTION_SERVICES: Allows an application
--              to initiate a request/response mechanism with an
--              external user.  An example of this is a remote
--              procedure call.
--
--        The design and specification of this module is based on
--      schedulability concerns. A single representation of
--      priority is used as a scheduling mechanism.  Activities
--      such as data transfers can be performed in a synchronous
--      or asynchronous manner. An application can specify a
--      timeout that bounds the blocking incurred in the
--      synchronous case, or is interpreted as a deadline in the
--      asynchronous case.
--
-------------------------------------------------------------------
```

Appendix B: Protocol Management

```
------------------------------------------------------------------
-- Package: LW_PROTOCOL_MANAGEMENT
--
-- Purpose: This package is a part of the Ada binding to the
--   SAFENET lightweight application services. This particular
--   package provides the implementation-dependent
--   protocol management capabilities, including
--
--      o Initialization of protocol suite: Before an application
--        can request communication services over the lightweight
--        protocol suite, initialization is required.
--
--      o  Termination of the protocol suite.
--
--      o  Procedures and functions that permit an application to
--        determine and change the values of certain parameters.
--
--  Note: The definition of parameters in LW_PROTOCOL_DATA is
--  implementation-dependent.  The values appearing below are
--  based on the XTP Specification, Version 3.6.
--
------------------------------------------------------------------

with    LW_ADDRESS_MANAGEMENT;
package LW_PROTOCOL_MANAGEMENT is

   package LWAM renames LW_ADDRESS_MANAGEMENT;

------------------------------------------------------------------
--    The LW_PROTOCOL_DATA structure encapsulates parameters that
--  are required for a particular implementation of XTP.  The
--  declaration of parameters is implementation-dependent.
------------------------------------------------------------------
   type LW_PROTOCOL_DATA is
      record
          BURST                      : positive;
          CONNECTION_RETRY_COUNT     : positive;
          MAX_INPUT_CONTEXT_QUEUE    : positive;
          MAX_OUTPUT_CONTEXT_QUEUE   : positive;
          RECEIVER_ALLOCATION_QUEUE  : positive;
          SENDER_DEFAULT_ALLOCATION  : positive;
          ACKNOWLEDGEMENT_WAIT_TIMER : duration;
          CONNECTION_ACTIVITY_TIMER  : duration;
          CONNECTION_TIMEOUT         : duration;
          MAX_DATA_RATE              : duration;
          ROUND_TRIP_TIME            : duration;
      end record;

   type STATUS is (OK, INVALID_PROTOCOL_PARAMETER,
```

```
        INVALID_PROTOCOL_OPERATION);

-----------------------------------------------------------------
--   The procedure INITIALIZE_LW_PROTOCOL performs initialization
--  of the lightweight protocol suite.  The input parameters are
--  the MAC physical address, the network physical address, and
--  the initialization record.
-----------------------------------------------------------------
   procedure INITIALIZE_LW_PROTOCOL
        ( MAC      : in     LWAM.Physical_Address;
          NET      : in     LWAM.Physical_Address;
          START_UP : in     LW_PROTOCOL_DATA;
          RESULT   :    out STATUS);

-----------------------------------------------------------------
--    The procedure TERMINATE_LW_PROTOCOL terminates the
--  lightweight protocol for a specified MAC and network physical
--  address.
-----------------------------------------------------------------
   procedure TERMINATE_LW_PROTOCOL
        ( MAC    : in     LWAM.Physical_Address;
          NET    : in     LWAM.Physical_Address;
          RESULT :    out STATUS);

-----------------------------------------------------------------
--   The procedure UPDATE_PROTOCOL_PARAMETERS changes one or more
-- of the parameters.
-----------------------------------------------------------------
   procedure UPDATE_PROTOCOL_PARAMETERS
        ( MAC   : in     LWAM.Physical_Address;
          NET   : in     LWAM.Physical_Address;
          PARMS : in     LW_PROTOCOL_DATA;
          RESULT:    out STATUS);

-----------------------------------------------------------------
--   The procedure RETURN_PROTOCOL_PARAMETERS provides the values
--  of the parameters for the specified MAC and network physical
--  addresses.
-----------------------------------------------------------------
   procedure RETURN_PROTOCOL_PARAMETERS
        ( MAC     : in     LWAM.Physical_Address;
          NET     : in     LWAM.Physical_Address;
          LW_PARMS:    out LW_PROTOCOL_DATA;
          RESULT  :    out STATUS);

-----------------------------------------------------------------
--   The procedure PROTOCOL_PARAMETERS_DEFINED can be used to
-- determine if the relevant protocol parameters are defined
-- for XTP. This can be used to test for the presence of default
-- parameters.
-----------------------------------------------------------------
```

```
   procedure PROTOCOL_PARAMETERS_DEFINED
        ( MAC     : in     LWAM.Physical_Address;
          NET     : in     LWAM.Physical_Address;
          RESULT  :    out STATUS);

-----------------------------------------------------------------

private

-- implementation-defined

end LW_PROTOCOL_MANAGEMENT;
```

Appendix C: Address Management

```
------------------------------------------------------------------
--
-- Package: LW_ADDRESS_MANAGEMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services.  The purpose of
--   of this package is to provide support for the management
--   of the logical and physical name space.  The following
--   functionality is provided:
--
--      o  Create binding between logical and physical addresses.
--
--      o  Remove logical-physical binding.
--
--      o  Certain query functions are supported, e.g., to find
--         if an address is bound, and to return the total number
--         of bound address mappings.
--
--     This package does not assume that the address bindings are
--  unique.  For example, several logical names may be bound to
--  the same physical address.
--
------------------------------------------------------------------
with LW_COMMUNICATIONS_SUPPORT;
package LW_ADDRESS_MANAGEMENT is

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;

   type ADDRESS_ID is private;

   type PHYSICAL_ADDRESS is new LWCS.BYTE_BUFFER;
   pragma pack (PHYSICAL_ADDRESS);

   type STATUS is (OK, UNABLE_TO_ALLOCATE,
      INVALID_LOGICAL_NAME, INVALID_PHYSICAL_ADDRESS,
      LOGICAL_NAME_ALREADY_BOUND, NONEXISTENT_BINDING);

------------------------------------------------------------------
--    The procedure BIND creates a logical name to
--  physical address binding.  An identifier is returned for use
--  in data transfer operations.
------------------------------------------------------------------
   procedure BIND
         (L_NAME: in     LWCS.Logical_Name;
          P_NAME: in     PHYSICAL_ADDRESS;
          ID    :    out ADDRESS_ID;
          RESULT:    out STATUS);

------------------------------------------------------------------
```

```
--    The procedure UNBIND will remove the specified
--  logical name to physical address mapping.
-------------------------------------------------------------------
   procedure UNBIND
         (L_NAME: in     LWCS.Logical_Name;
          P_NAME: in     PHYSICAL_ADDRESS;
          RESULT:    out STATUS);

   procedure UNBIND
         (ID     : in     ADDRESS_ID;
          RESULT :    out STATUS);

-------------------------------------------------------------------
--    The procedure UNBIND_LOGICAL_NAME will remove the
--  specified logical name (and all associated physical
--  addresses).
-------------------------------------------------------------------
   procedure UNBIND_LOGICAL_NAME
               (L_NAME: in     LWCS.Logical_Name;
                RESULT:    out STATUS);

-------------------------------------------------------------------
--    The procedure UNBIND_PHYSICAL_ADDRESS will remove all
--  bindings to the specified physical address (and all the
--  associated logical names).
-------------------------------------------------------------------
   procedure UNBIND_PHYSICAL_ADDRESS
               (P_NAME: in     PHYSICAL_ADDRESS;
                RESULT:    out STATUS);

-------------------------------------------------------------------
--    The function ADDRESS_BINDING_EXISTS returns true if and
--  only if the specified logical name is bound to the specified
--  physical address.
-------------------------------------------------------------------
   function ADDRESS_BINDING_EXISTS
        (L_NAME: in     LWCS.Logical_Name;
         P_NAME: in     PHYSICAL_ADDRESS)
                return boolean;

-------------------------------------------------------------------
--    The function LOGICAL_NAME_IS_BOUND returns true if and only
--  if the specified logical name is bound.
-------------------------------------------------------------------
   function LOGICAL_NAME_IS_BOUND
        (L_NAME: in LWCS.Logical_Name)
                return boolean;

-------------------------------------------------------------------
--   The function PHYSICAL_ADDRESS_IS_BOUND returns true if and
--  only if the specified physical address is bound.
```

```
-------------------------------------------------------------------
   function PHYSICAL_ADDRESS_IS_BOUND
        (P_NAME: in PHYSICAL_ADDRESS)
                return boolean;

-------------------------------------------------------------------
--   The function NUMBER_OF_LOGICAL_NAME_BINDINGS returns the
--  number of physical address that are bound to a specified
--  logical name.
-------------------------------------------------------------------
   function NUMBER_OF_LOGICAL_NAME_BINDINGS
        (L_NAME: in LWCS.Logical_Name)
                return natural;

-------------------------------------------------------------------
--    The function NUMBER_OF_PHYSICAL_NAME_BINDINGS returns the
--  number of logical names that are bound to a specified
--  physical address.
-------------------------------------------------------------------
   function NUMBER_OF_PHYSICAL_NAME_BINDINGS
        (P_NAME : in PHYSICAL_ADDRESS)
                return natural;

-------------------------------------------------------------------
--    The function NUMBER_OF_ADDRESSES_BOUND returns the number
--  of address bindings that exist for the application.
-------------------------------------------------------------------
   function NUMBER_OF_ADDRESSES_BOUND
                return natural;


private

-- implementation-defined

end LW_ADDRESS_MANAGEMENT;
```

Appendix D: Communications Support

```
------------------------------------------------------------
-- Package: LW_COMMUNICATIONS_SUPPORT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services. This package
--   contains the following types and subprograms used by other
--   packages.
--
--      o  Logical names
--
--      o  Data buffers (for both connection-oriented and
--         connectionless transfers).
--
--      o  Information relating to scheduling, including
--         priority, timeout, and request mode (synchronous or
--         asynchronous).
--
--      o  A record containing parameters used in data
--         transfers.
--
--      o  Types and procedures in support of asynchronous
--         communication.
--
------------------------------------------------------------

package LW_COMMUNICATIONS_SUPPORT is

   type ACTIVITY_BLOCK_ID is private;

------------------------------------------------------------
--    Declare a logical name as an array of characters.
--  The length of the logical name is an implementation-defined
--  constant.
------------------------------------------------------------
   LOGICAL_NAME_LENGTH : constant := implementation-defined;
   subtype LOGICAL_NAME is string (1 .. LOGICAL_NAME_LENGTH);

------------------------------------------------------------
--    Declare a data buffer as an array of unsigned bytes.
------------------------------------------------------------
   type UNSIGNED_BYTE is range 0 .. 255;
   for  UNSIGNED_BYTE'size use 8;

   type BYTE_BUFFER is array (Integer range <>)
           of UNSIGNED_BYTE;
   pragma pack (BYTE_BUFFER);

------------------------------------------------------------
```

```
--    Define a buffer for connection-oriented data transfers.

--  The maximum size of the buffer is implementation-defined.
-------------------------------------------------------------
   MAX_CONNECTION_BUFFER_SIZE : constant :=
        implementation-defined;

   type CONNECTION_BUFFER_SIZE is range
            1 .. MAX_CONNECTION_BUFFER_SIZE;
   type CONNECTION_BUFFER is array
           (CONNECTION_BUFFER_SIZE range <>) of
           UNSIGNED_BYTE;
   pragma pack (CONNECTION_BUFFER);

   type CONNECTION_DATA
           (SIZE : CONNECTION_BUFFER_SIZE) is
     record
        DATA : CONNECTION_BUFFER ( 1 .. SIZE);
     end record;

-------------------------------------------------------------
--    Define a buffer for use in connectionless data transfers.
--  The maximum length of the buffer is implementation-
--  defined.
-------------------------------------------------------------
   MAX_CONNECTIONLESS_BUFFER_SIZE : constant :=
        implementation-defined;

   type CONNECTIONLESS_BUFFER_SIZE is range
          1 .. MAX_CONNECTIONLESS_BUFFER_SIZE;
   type CONNECTIONLESS_BUFFER is array
        (CONNECTIONLESS_BUFFER_SIZE range <>) of
        UNSIGNED_BYTE;
   pragma pack (CONNECTIONLESS_BUFFER);

   type CONNECTIONLESS_DATA
          (SIZE : CONNECTIONLESS_BUFFER_SIZE) is
     record
        DATA : CONNECTIONLESS_BUFFER (1 .. SIZE);
     end record;

-------------------------------------------------------------
--    Declare a priority and timeout.  The scheduling of
--  activities is based only on priority. A timeout parameter
--  can be used to limit the amount of time that an application
--  will wait for completion of a requested activity.
-------------------------------------------------------------
   type PRIORITY is range 0 .. 255;
   subtype TIMEOUT is duration;

   type TERMINATE_MODE is (GRACEFUL, IMMEDIATE);
```

```
------------------------------------------------------------
--    Declare types relevant to asynchronous processing.  These
--  include an activity type, activity state, and an activity
--  index used to uniquely identify an asynchronous operation.
------------------------------------------------------------

   type ACTIVITY is (SYNCHRONOUS, ASYNCHRONOUS);

   subtype ACTIVITY_INDEX is integer range 0 .. integer'last;

   type ACTIVITY_STATE is (SUCCESS, IN_PROGRESS,
        UNKNOWN, FAILED, ERROR);

------------------------------------------------------------
--    Define the error control method to be used.
------------------------------------------------------------
   type ERROR_CONTROL is (NONE, RELIABLE, AGGRESSIVE);

------------------------------------------------------------
--    Define type MESSAGE_CLASS
------------------------------------------------------------

   subtype MESSAGE_CLASS is integer range 0 .. integer'last;

------------------------------------------------------------
--    Define a type ACTIVITY_BLOCK that encapsulates items used
--  in communications.  An object of this type is always passed
--  as an 'in' parameter.
------------------------------------------------------------
   type ACTIVITY_BLOCK is
     record
        CHECKSUM    : boolean;
        ERROR       : ERROR_CONTROL;
        LIFETIME    : duration;
        MODE        : ACTIVITY;
        P           : PRIORITY;
        SUSPEND     : boolean;
        TIMEOUT     : duration;
      end record;

------------------------------------------------------------
--    Define procedures that create and operate on elements of
--  an activity block.
------------------------------------------------------------
   procedure CREATE_ACTIVITY_BLOCK
        (DATA : in     ACTIVITY_BLOCK;
         ID   :    out ACTIVITY_BLOCK_ID );

   procedure DELETE_ACTIVITY_BLOCK
        (ID : in ACTIVITY_BLOCK_ID);
```

```
procedure DELETE_ALL_ACTIVITY_BLOCKS;

procedure ENABLE_CHECKSUM
     (ID : in ACTIVITY_BLOCK_ID);

procedure DISABLE_CHECKSUM
     (ID : in ACTIVITY_BLOCK_ID);

function CHECKSUM_ENABLED
     (ID : in ACTIVITY_BLOCK_ID) return boolean;

procedure SET_ERROR_CONTROL
     (ID  : in ACTIVITY_BLOCK_ID;
      ERR : in ERROR_CONTROL);

procedure GET_ERROR_CONTROL
     (ID  : in     ACTIVITY_BLOCK_ID;
      ERR :    out ERROR_CONTROL);

procedure SET_MESSAGE_LIFETIME
     (ID   : in ACTIVITY_BLOCK_ID;
      TIME : in duration);

procedure GET_MESSAGE_LIFETIME
     (ID   : in     ACTIVITY_BLOCK_ID;
      TIME :    out duration);

procedure SET_ACTIVITY_MODE
     (ID : in ACTIVITY_BLOCK_ID;
      MODE : in ACTIVITY);

procedure GET_ACTIVITY_MODE
     (ID : in ACTIVITY_BLOCK_ID;
      MODE : out ACTIVITY);

procedure SET_PRIORITY
     (ID : in ACTIVITY_BLOCK_ID;
      P  : in PRIORITY);

procedure GET_PRIORITY
     (ID : in     ACTIVITY_BLOCK_ID;
      P  :    out PRIORITY);

procedure ENABLE_CALLER_SUSPENSION
     (ID : in ACTIVITY_BLOCK_ID);

procedure DISABLE_CALLER_SUSPENSION
     (ID : in ACTIVITY_BLOCK_ID);

function CALLER_SUSPENSION_ENABLED
     (ID : in ACTIVITY_BLOCK_ID) return boolean;
```

```
   procedure SET_TIMEOUT
        (ID : in ACTIVITY_BLOCK_ID;
         T  : in duration);

   procedure GET_TIMEOUT
        (ID : in     ACTIVITY_BLOCK_ID;
         T  :    out duration);

-----------------------------------------------------------------
--    Define subprograms for use in asynchronous transfer. They
--  can be used to determine the state of an activity, to wait
--  on an asynchronous activity, or to cancel previously
--  requested asynchronous activities that are pending
--  completion.
-----------------------------------------------------------------
   function GET_ASYNCHRONOUS_ACTIVITY_STATE
        (ID : in ACTIVITY_INDEX)
            return ACTIVITY_STATE;

   procedure WAIT_ON_ASYNCHRONOUS_ACTIVITY
        (ID : in ACTIVITY_INDEX;
         T  : in duration);

   procedure CANCEL_ASYNCHRONOUS_ACTIVITY
        (ID : in ACTIVITY_INDEX);

   procedure CANCEL_ALL_ASYNCHRONOUS_ACTIVITIES;

-----------------------------------------------------------------
--    The procedure TERMINATE_ALL_TRANSFERS will terminate all
--  transfers.  This includes closing all connections (in a
--  graceful manner, and terminating all unicast, multicast
--  and transactions that may be in progress.
-----------------------------------------------------------------
   procedure TERMINATE_ALL_TRANSFERS (MODE : TERMINATE_MODE);

-----------------------------------------------------------------

private

-- implementation-defined

end  LW_COMMUNICATIONS_SUPPORT;
```

Appendix E: Connection Management

```
------------------------------------------------------------

-- Package: LW_CONNECTION_MANAGEMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services. This package
--   exports operations associated with connections, providing
--   the ability to
--
--      o  Open a connection
--
--      o  Open a connection with data transfer
--
--      o  Close a connection
--
--      These operations can be performed in a synchronous or
--  asynchronous manner.  This package also provides
--  subprograms to return the state of a connection.
--
------------------------------------------------------------

with LW_COMMUNICATIONS_SUPPORT;

package LW_CONNECTION_MANAGEMENT is

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;

   CONNECTION_ERROR: exception;

   type CONNECTION_ID is private;

   type CONNECTION_BLOCK_ID is private;

   type STATE is
        ( UNKNOWN, OPEN_REQUEST_SENT, OPEN_RESPONSE_PENDING,
          OPEN, GRACEFUL_CLOSE_SENT, IMMEDIATE_CLOSE_SENT,
          GRACEFUL_CLOSE_RECEIVED, IMMEDIATE_CLOSE_RECEIVED);

------------------------------------------------------------
--    Define a type used to specify if flow control is to be
--  enabled.
------------------------------------------------------------

   type FLOW_CONTROL is (ENABLED, DISABLED);

------------------------------------------------------------
--    Define a type used to specify the acknowledgement policy.
------------------------------------------------------------
```

```
   type ACKNOWLEDGEMENT_GRANULARITY is (OCTET, PACKET, MESSAGE,
        ONCE);

   type ACKNOWLEDGEMENT_POLICY is
     record
       GRANULARITY  : ACKNOWLEDGEMENT_GRANULARITY;
       NUMBER_UNITS : natural;
     end record;

   type CONNECTION_BLOCK is
     record
       FLOW_CTL    : FLOW_CONTROL;
       RATE_CTL    : natural;
       POLICY      : ACKNOWLEDGEMENT_POLICY;
       RESERVATION : boolean;
     end record;

-------------------------------------------------------------
--    The following procedures create and operate on the
--  elements of the connection block.
-------------------------------------------------------------

  procedure CREATE_CONNECTION_BLOCK
       (DATA : in     CONNECTION_BLOCK;
        ID   :    out CONNECTION_BLOCK_ID);

  procedure DELETE_CONNECTION_BLOCK
       (ID   : in CONNECTION_BLOCK_ID);

  procedure DELETE_ALL_CONNECTION_BLOCKS;

  procedure SET_FLOW_CONTROL
       (ID : in CONNECTION_BLOCK_ID;
        FC : in FLOW_CONTROL);

  procedure GET_FLOW_CONTROL
       (ID : in     CONNECTION_BLOCK_ID;
        FC :    out FLOW_CONTROL);

  procedure SET_RATE_CONTROL
       (ID : in CONNECTION_BLOCK_ID;
        RC : in natural);

  procedure GET_RATE_CONTROL
       (ID     : in     CONNECTION_BLOCK_ID;
        RATE   :    out natural);

  procedure SET_POLICY
       (ID     : in CONNECTION_BLOCK_ID;
        POLICY : in ACKNOWLEDGEMENT_POLICY);
```

```
  procedure GET_POLICY

       (ID     : in     CONNECTION_BLOCK_ID;
        POLICY :    out ACKNOWLEDGEMENT_POLICY);

  procedure SET_RESERVATION
       (ID   : in CONNECTION_BLOCK_ID;
        RES  : in boolean);

  procedure GET_RESERVATION
       (ID   : in     CONNECTION_BLOCK_ID;
        RES  :    out boolean);


------------------------------------------------------------------
--    The procedure OPEN_CONNECTION allows an application to
--  establish a connection with another user.
------------------------------------------------------------------
   procedure OPEN_CONNECTION
        ( DESTINATION : in     LWCS.Logical_Name;
          PARAMETERS  : in     LWCS.Activity_Block_ID;
          CONN_PARAMS : in     Connection_Block_ID;
          CONNECTION  :    out Connection_ID;
          INDEX       :    out LWCS.Activity_Index );

------------------------------------------------------------------
--    The procedure OPEN_CONNECTION_WITH_DATA allows an
--  application to open a connection with another user and
--  include a data buffer as part of the process. The
--  procedure is overloaded to allow the application to specify
--  a message class.
------------------------------------------------------------------
   procedure OPEN_CONNECTION_WITH_DATA
        ( DESTINATION   : in     LWCS.Logical_Name;
          PARAMETERS    : in     LWCS.Activity_Block_ID;
          CONN_PARAMS   : in     Connection_Block_ID;
          DATA          : in     LWCS.Connection_Data;
          CONNECTION    :    out CONNECTION_ID;
          INDEX         :    out LWCS.Activity_Index );

   procedure OPEN_CONNECTION_WITH_DATA
        ( DESTINATION   : in     LWCS.Logical_Name;
          PARAMETERS    : in     LWCS.Activity_Block_ID;
          CONN_PARAMS   : in     Connection_Block_ID;
          DATA          : in     LWCS.Connection_Data;
          MSG_CLASS     : in     LWCS.MESSAGE_CLASS;
          CONNECTION    :    out CONNECTION_ID;
          INDEX         :    out LWCS.Activity_Index );

------------------------------------------------------------------
--    The procedure CLOSE_CONNECTION allows an application to
```

```
--  to close a previously opened connection.

-------------------------------------------------------------
   procedure CLOSE_CONNECTION
        ( CONNECTION : in     Connection_ID;
          MODE       : in     LWCS.TERMINATE_MODE;
          CONTROL    : in     LWCS.Activity_Block_ID;
          INDEX      :    out LWCS.Activity_Index );

-------------------------------------------------------------
--    The procedure CLOSE_ALL_CONNECTIONS closes all of an
--  application's connections, regardless of state.
-------------------------------------------------------------
   procedure CLOSE_ALL_CONNECTIONS
        (MODE      : in     LWCS.TERMINATE_MODE;
         CONTROL   : in     LWCS.Activity_Block_ID;
         INDEX     :    out LWCS.Activity_Index );

-------------------------------------------------------------
--    The procedure ACCEPT_CONNECTION allows the specified
--  logical name to establish a connection.
-------------------------------------------------------------
    procedure ACCEPT_CONNECTION
        ( FROM       : in     LWCS.Logical_Name;
          CONNECTION :    out CONNECTION_ID);

-------------------------------------------------------------
--    The function GET_CONNECTION_STATE returns the state of
--  state of the specified connection.
-------------------------------------------------------------
    function GET_CONNECTION_STATE
        ( CONNECTION : in     Connection_ID)
                return STATE;

-------------------------------------------------------------
--    The function NUMBER_OF_CONNECTIONS returns the number of
--  connections in a specified state.  The function definition
--  is overloaded to also return the total number of
--  connections.
-------------------------------------------------------------
   function NUMBER_OF_CONNECTIONS
        ( requested_state : in STATE)
           return natural;

   function NUMBER_OF_CONNECTIONS
        return natural;

-------------------------------------------------------------
--    The function CONNECTION_IS_KNOWN returns true if and only
--  if the specified connection exists.
-------------------------------------------------------------
```

```
   function CONNECTION_IS_KNOWN
        ( CONNECTION : in LWCS.Logical_Name)
                return boolean;

-----------------------------------------------------------------
--    The function BUFFER_LENGTH_IS_VALID returns true if and
--  only if the specified value is valid (i.e., acceptable to
--  the underlying implementation);
-----------------------------------------------------------------
   function BUFFER_LENGTH_IS_VALID
        ( LENGTH : in positive)
            return boolean;
private

   type CONNECTION_ID is implementation-defined

   type CONNECTION_BLOCK_ID is implementation-defined

end  LW_CONNECTION_MANAGEMENT;
```

Appendix F: Group Management

```
------------------------------------------------------------
-- Package: LW_GROUP_MANAGEMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services.  The package
--   defines subprograms in support of group management.
--   This includes operations for:
--
--      o  Opening a group
--
--      o  Closing a group
--
--      o  Joining a group
--
--      o  Leaving a group
--
--   This package does not assume a particular model for the
--  implementation of group management, e.g., localized or
--  distributed.  Support is provided for synchronous and
--  asynchronous operations. In addition, several functions
--  are provided to support an application, for example, by
--  returning the number of members of a group.
--
------------------------------------------------------------

with LW_COMMUNICATIONS_SUPPORT;
with LW_CONNECTION_MANAGEMENT;

package LW_GROUP_MANAGEMENT is

  package LWCS renames LW_COMMUNICATIONS_SUPPORT;
  package LWCM renames LW_CONNECTION_MANAGEMENT;

  type GROUP_ID is private;

  GROUP_ERROR: exception;

  MAX_GROUPS : constant := implementation-defined;
  MAX_MEMBERS : constant := implementation-defined;
  subtype MEMBERSHIP_INDEX is integer range
        1 .. MAX_MEMBERS;
  type MEMBERSHIP_LIST is array ( MEMBERSHIP_INDEX range <> )
        of LWCS.Logical_Name;
  type PERMISSION_LIST is array ( MEMBERSHIP_INDEX range <> )
        of boolean;

------------------------------------------------------------
--    The procedure OPEN_GROUP is a request to open a group
```

```
--  with the specified group_name. A list of the members of the
--  group can also be specified.
-------------------------------------------------------------
  procedure OPEN_GROUP
        ( GROUP      : in     LWCS.Logical_Name;
          MEMBERS    : in     MEMBERSHIP_LIST;
          PERMISSION : in     PERMISSION_LIST;
          CONTROL    : in     LWCS.Activity_Block_ID;
          PARAMS     : in     LWCM.Connection_Block_ID;
          NOTIFY     : in     boolean;
          INDEX      :    out LWCS.Activity_Index;
          ID         :    out GROUP_ID);
-------------------------------------------------------------
--    The procedure JOIN_GROUP is a request to add the
--  specified logical name to a group. If NOTIFY is true, the
--  other members of the group will be informed of the new
--  member.
-------------------------------------------------------------
  procedure JOIN_GROUP
        ( MEMBER    : in     LWCS.Logical_Name;
          GROUP     : in     GROUP_ID;
          PRIV      : in     boolean;
          NOTIFY    : in     boolean;
          CONTROL   : in     LWCS.Activity_Block_ID;
          INDEX     :    out LWCS.Activity_Index);

-------------------------------------------------------------
--    The procedure LEAVE_GROUP allows an application to
--  remove itself from a multicast group.  If NOTIFY is true
--  the other members of the group will be informed of the
--  deleted member.
-------------------------------------------------------------
  procedure LEAVE_GROUP
        ( MEMBER    : in     LWCS.Logical_Name;
          GROUP     : in     GROUP_ID;
          NOTIFY    : in     boolean;
          CONTROL   : in     LWCS.Activity_Block_ID;
          INDEX     :    out LWCS.Activity_Index);

-------------------------------------------------------------
--    The procedure REMOVE_MEMBER_FROM_GROUP allows an
--  application to remove another application from a group.
-------------------------------------------------------------
  procedure REMOVE_MEMBER_FROM_GROUP
        ( GROUP     : in     GROUP_ID;
          MEMBER    : in     LWCS.Logical_Name;
          CONTROL   : in     LWCS.Activity_Block_ID;
          NOTIFY    : in     boolean;
          INDEX     :    out LWCS.Activity_Index );

-------------------------------------------------------------
```

```
--    The procedure NUMBER_OF_GROUP_MEMBERS returns the number
--  of members of a specified group.
-------------------------------------------------------------
  procedure NUMBER_OF_GROUP_MEMBERS
        ( GROUP     : in     GROUP_ID;
          CONTROL   : in     LWCS.Activity_Block_ID;
          INDEX     :    out LWCS.Activity_Index;
          NUMBER    :    out natural);

-------------------------------------------------------------
--    The procedure OBTAIN_GROUP_MEMBERS returns the number of
--  members, logical names, and permission values for a given
--  group.
-------------------------------------------------------------
   procedure OBTAIN_GROUP_MEMBERS
        ( GROUP     : in     GROUP_ID;
          CONTROL   : in     LWCS.Activity_Block_ID;
          INDEX     :    out LWCS.Activity_Index;
          NUMBER    :    out natural;
          MEMBERS   :    out MEMBERSHIP_LIST;
          PRIV      :    out PERMISSION_LIST);

-------------------------------------------------------------
--   The procedure GET_PERMISSION_OF_MEMBER will return an
--  indication whether a member of a group has permission to
--  close the group. The procedure GRANT_PERMISSION_TO_CLOSE
--  allows a member to give close group permission to another
--  member of the group.
-------------------------------------------------------------
   procedure GET_PERMISSION_OF_MEMBER
        ( MEMBER      : in     LWCS.Logical_Name;
          GROUP       : in     GROUP_ID;
          CONTROL     : in     LWCS.Activity_Block_ID;
          INDEX       :    out LWCS.Activity_Index;
          VALUE       :    out boolean);

   procedure GRANT_PERMISSION_TO_CLOSE
        ( MEMBER      : in     LWCS.Logical_Name;
          GROUP       : in     GROUP_ID;
          CONTROL     : in     LWCS.Activity_Block_ID;
          INDEX       :    out LWCS.Activity_Index);

-------------------------------------------------------------

private

  type GROUP_ID is new integer;
-- implementation-defined

end LW_GROUP_MANAGEMENT;
```

Appendix G: Data Transfer Services

```
------------------------------------------------------------
-- Package: LW_DATA_TRANSFER
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services.  This package
--   provides support for data transfer in distributed
--   applications.  Unicast transfers can be either
--   connection-oriented or connectionless. In the latter case,
--   the user can specify either XTP or the OSI connectionless
--   protocol. This package also provides support for multicast
--   transfers.
--
--     The procedures specified in this package can be used in
--   either a synchronous or asynchronous manner.  In the
--   synchronous case, the user-specified timeout bounds the
--   blocking permitted. In the asynchronous case the user-
--   supplied timeout is a deadline.
--
------------------------------------------------------------

with    LW_ADDRESS_MANAGEMENT;
with    LW_BUFFER_MANAGEMENT;
with    LW_COMMUNICATIONS_SUPPORT;
with    LW_CONNECTION_MANAGEMENT;
with    LW_GROUP_MANAGEMENT;
with    CALENDAR;
with    SYSTEM;
package LW_DATA_TRANSFER is

   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWBM renames LW_BUFFER_MANAGEMENT;
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWGM renames LW_GROUP_MANAGEMENT;

------------------------------------------------------------
--    The following exceptions are raised if an error is found
--  in connection with a data transfer operation of the
--  indicated type.
------------------------------------------------------------
   CONNECTION_TRANSFER_ERROR     : exception;
   CONNECTIONLESS_TRANSFER_ERROR : exception;
   MULTICAST_TRANSFER_ERROR      : exception;

------------------------------------------------------------
--    The type PROTOCOL_SUITE can be used to select a protocol
--  to be used in connectionless transfers.
------------------------------------------------------------
   type PROTOCOL_SUITE is (XTP, OSI_CONNECTIONLESS_TRANSPORT,
```

```
          OPTIONAL);


-------------------------------------------------------------
--    The type RECEPTION_CRITERIA is used in multicast data
--  transfers to indicate required reception quality for a
--  member of the group.  RECEPTION_LIST is an array of
--  RECEPTION_CRITERIA.
-------------------------------------------------------------
   type RECEPTION_CRITERIA is (REQUIRED, OPTIONAL);
   type RECEPTION_LIST is array (LWGM.Membership_Index range <>)
          of RECEPTION_CRITERIA;

-------------------------------------------------------------
--    The type STATUS is returned by query procedures to
--  indicate that a message has been received.
-------------------------------------------------------------
   type STATUS is (MESSAGE_PRESENT, MESSAGE_NOT_PRESENT);

-------------------------------------------------------------
--    The type PURGE_ORDER allows an application to remove
--  messages in the specified order.
-------------------------------------------------------------
   type PURGE_ORDER is (OLDEST_FIRST, NEWEST_FIRST);

-------------------------------------------------------------
--    The procedure SEND_CONNECTION_MESSAGE allows an
--  application to send a message over a previously established
--  connection. The procedure is overloaded to allow the
--  application to specify a message class.
-------------------------------------------------------------
   procedure SEND_CONNECTION_MESSAGE
        ( CONNECTION    : in     LWCM.Connection_ID;
          PARMS         : in     LWCS.Activity_Block_ID;
          DATA          : in     LWCS.Connection_Data;
          INDEX         :    out LWCS.Activity_Index );

   procedure SEND_CONNECTION_MESSAGE
        ( CONNECTION    : in     LWCM.Connection_ID;
          PARMS         : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          : in     LWCS.Connection_Data;
          INDEX         :    out LWCS.Activity_Index );

-------------------------------------------------------------
--    The procedure GET_CONNECTION_MESSAGE can be used to
--  receive a message for a specified connection or wait a
--  specified time for receipt of the message.  The procedure
--  is overloaded to allow the application to specify a message
--   class.
-------------------------------------------------------------
```

```
   procedure GET_CONNECTION_MESSAGE
        ( CONNECTION    : in     LWCM.Connection_ID;
          PARMS         : in     LWCS.Activity_Block_ID;
          DATA          :    out LWCS.Connection_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          MSG_RCLASS    :    out LWCS.Message_Class;
          RESULT        :    out STATUS);

   procedure GET_CONNECTION_MESSAGE
        ( CONNECTION    : in     LWCM.Connection_ID;
          PARMS         : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          :    out LWCS.Connection_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          RESULT        :    out STATUS);


-------------------------------------------------------------
--    The procedure SEND_CONNECTIONLESS_MESSAGE allows an
--  application to send a message to another application
--  without establishing a connection. Either XTP or the
--  OSI connectionless transport may be specified.  The
--  procedure is overloaded to allow the application to specify
--  a message class.
-------------------------------------------------------------
   procedure SEND_CONNECTIONLESS_MESSAGE
        ( DESTINATION   : in     LWAM.Address_ID;
          PROTOCOL      : in     PROTOCOL_SUITE;
          PARMS         : in     LWCS.Activity_Block_ID;
          DATA          : in     LWCS.Connectionless_Data;
          INDEX         :    out LWCS.Activity_index);

   procedure SEND_CONNECTIONLESS_MESSAGE
        ( DESTINATION   : in     LWAM.Address_ID;
          PROTOCOL      : in     PROTOCOL_SUITE;
          PARMS         : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          : in     LWCS.Connectionless_Data;
          INDEX         :    out LWCS.Activity_index);


-------------------------------------------------------------
--    The procedure GET_CONNECTIONLESS_MESSAGE can be used to
--  receive a message from a specified source.
--    The procedure GET_ANY_CONNECTIONLESS_MESSAGE can be used
--  to receive a message from any source.
--  These procedures are overloaded to allow the application to
--  specify a message class.
-------------------------------------------------------------
   procedure GET_CONNECTIONLESS_MESSAGE
        ( SOURCE        : in     LWAM.Address_ID;
```

```
          CONTROL       : in     LWCS.Activity_Block_ID;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          MSG_RCLASS    :    out LWCS.Message_Class;
          RESULT        :    out STATUS );

   procedure GET_ANY_CONNECTIONLESS_MESSAGE
        ( CONTROL       : in     LWCS.Activity_Block_ID;
          SOURCE        :    out LWAM.Address_ID;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          MSG_RCLASS    :    out LWCS.Message_Class;
          RESULT        :    out STATUS );

   procedure GET_CONNECTIONLESS_MESSAGE
        ( SOURCE        : in     LWAM.Address_ID;
          CONTROL       : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          RESULT        :    out STATUS );

   procedure GET_ANY_CONNECTIONLESS_MESSAGE
        ( CONTROL       : in     LWCS.Activity_Block_ID;
          SOURCE        :    out LWAM.Address_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_PRIORITY  :    out LWCS.Priority;
          RESULT        :    out STATUS );

-------------------------------------------------------------
--    The procedure SEND_MULTICAST_MESSAGE allows an
--  application to send a message to a group of other
--  applications. The user can select the protocol to be used.
--  The procedure is overloaded to allow the application to
--  specify a message class.
-------------------------------------------------------------
   procedure SEND_MULTICAST_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         CONDITION      : in     RECEPTION_LIST;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         DATA           : in     LWCS.Connectionless_Data;
         INDEX          :    out LWCS.Activity_Index );

   procedure SEND_MULTICAST_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         RECEIVERS      : in     natural;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         DATA           : in     LWCS.Connectionless_Data;
```

```
        INDEX          :    out LWCS.Activity_Index );


   procedure SEND_MULTICAST_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         CONDITION      : in     RECEPTION_LIST;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         MSG_CLASS      : in     LWCS.Message_Class;
         DATA           : in     LWCS.Connectionless_Data;
         INDEX          :    out LWCS.Activity_Index );

   procedure SEND_MULTICAST_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         RECEIVERS      : in     natural;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         MSG_CLASS      : in     LWCS.Message_Class;
         DATA           : in     LWCS.Connectionless_Data;
         INDEX          :    out LWCS.Activity_Index );

-----------------------------------------------------------------
--    The procedure SEND_MULTICAST_STREAM_MESSAGE allows an
--  application to send a message to a group of other
--  applications via a connection. The user can select the
--  protocol to be used. The procedure is overloaded to allow
--  the application to specify a message class.
-----------------------------------------------------------------
   procedure SEND_MULTICAST_STREAM_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         DATA           : in     LWCS.Connection_Data;
         PARAMS         : in     LWCM.Connection_Block_ID;
         INDEX          :    out LWCS.Activity_Index );

   procedure SEND_MULTICAST_STREAM_MESSAGE
        (GROUP          : in     LWGM.Group_ID;
         PROTOCOL       : in     PROTOCOL_SUITE;
         CONTROL        : in     LWCS.Activity_Block_ID;
         MSG_CLASS      : in     LWCS.Message_Class;
         DATA           : in     LWCS.Connection_Data;
         PARAMS         : in     LWCM.Connection_Block_ID;
         INDEX          :    out LWCS.Activity_Index );

-----------------------------------------------------------------
--    The procedure GET_MESSAGE_FOR_GROUP allows an application
--  to receive a message sent to the specified group.
--  GET_MESSAGE_FOR_ANY_GROUP allows an application
--  to receive a message sent to any group to which the
--  application belongs.
```

```
--  These procedures are overloaded to allow the application to
--  specify a message class.
-------------------------------------------------------------
   procedure GET_MESSAGE_FOR_GROUP
        ( GROUP         : in     LWGM.Group_ID;
          CONTROL       : in     LWCS.Activity_Block_ID;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_RCLASS    :    out LWCS.Message_Class;
          RESULT        :    out STATUS );

   procedure GET_MESSAGE_FOR_ANY_GROUP
        ( CONTROL       : in     LWCS.Activity_Block_ID;
          SOURCE        :    out LWGM.Group_ID;
          DATA          :    out LWCS.Connectionless_Data;
          MSG_RCLASS    :    out LWCS.Message_Class;
          RESULT        :    out STATUS );

   procedure GET_MESSAGE_FOR_GROUP
        ( GROUP         : in     LWGM.Group_ID;
          CONTROL       : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          DATA          :    out LWCS.Connectionless_Data;
          RESULT        :    out STATUS );

   procedure GET_MESSAGE_FOR_ANY_GROUP
        ( CONTROL       : in     LWCS.Activity_Block_ID;
          MSG_CLASS     : in     LWCS.Message_Class;
          SOURCE        :    out LWGM.Group_ID;
          DATA          :    out LWCS.Connectionless_Data;
          RESULT        :    out STATUS );

-------------------------------------------------------------
--    The following procedures are provided in support of
--  broadcast.  An application can enable or disable receipt
--  of broadcast messages and get or send broadcast messages.
-------------------------------------------------------------
   procedure ENABLE_BROADCAST;

   procedure DISABLE_BROADCAST;

   function BROADCAST_ENABLED return boolean;

   procedure BROADCAST_MESSAGE
        ( PARMS : in     LWCS.Activity_Block_ID;
          DATA  : in     LWCS.Connectionless_data;
          INDEX :    out LWCS.Activity_Index);

   procedure GET_BROADCAST_MESSAGE
        ( PARMS        : in     LWCS.Activity_Block_ID;
          DATA         :    out LWCS.Connectionless_data;
          MSG_PRIORITY :    out LWCS.Priority;
```

```
          MSG_RCLASS   :     out LWCS.Message_Class;

          RESULT       :     out STATUS);

   procedure BROADCAST_MESSAGE
        ( PARMS     : in     LWCS.Activity_Block_ID;
          MSG_CLASS : in     LWCS.Message_Class;
          DATA      : in     LWCS.Connectionless_data;
          INDEX     :    out LWCS.Activity_Index);

   procedure GET_BROADCAST_MESSAGE
        ( PARMS        : in     LWCS.Activity_Block_ID;
          MSG_CLASS    : in     LWCS.Message_Class;
          DATA         :    out LWCS.Connectionless_data;
          MSG_PRIORITY :    out LWCS.Priority;
          RESULT       :    out STATUS);

-----------------------------------------------------------------
--    The following procedures duplicate the functionality
--  defined above except that the buffer management package
--  is used to allocate messages.
-----------------------------------------------------------------
   procedure SEND_CONNECTION_BUFFER
        ( CONNECTION : in     LWCM.Connection_ID;
          PARMS      : in     LWCS.Activity_Block_ID;
          POINTER    : in     SYSTEM.Address;
          LENGTH     : in     natural;
          INDEX      :    out LWCS.Activity_Index );

   procedure SEND_CONNECTION_BUFFER
        ( CONNECTION : in     LWCM.Connection_ID;
          PARMS      : in     LWCS.Activity_Block_ID;
          MSG_CLASS  : in     LWCS.Message_Class;
          POINTER    : in     SYSTEM.Address;
          LENGTH     : in     natural;
          INDEX      :    out LWCS.Activity_Index );

   procedure GET_CONNECTION_BUFFER
        ( CONNECTION   : in     LWCM.Connection_ID;
          PARMS        : in     LWCS.Activity_Block_ID;
          POINTER      : in     SYSTEM.Address;
          LENGTH       :    out natural;
          MSG_PRIORITY :    out LWCS.Priority;
          MSG_RCLASS   :    out LWCS.Message_Class;
          INDEX        :    out LWCS.Activity_Index);


   procedure GET_CONNECTION_BUFFER
        ( CONNECTION   : in     LWCM.Connection_ID;
          PARMS        : in     LWCS.Activity_Block_ID;
          MSG_CLASS    : in     LWCS.Message_Class;
```

```
        POINTER      : in     SYSTEM.Address;
        LENGTH       :    out natural;
        MSG_PRIORITY :    out LWCS.Priority;
        INDEX        :    out LWCS.Activity_Index);

procedure SEND_CONNECTIONLESS_BUFFER
     ( DESTINATION : in     LWAM.Address_ID;
       PROTOCOL    : in     PROTOCOL_SUITE;
       PARMS       : in     LWCS.Activity_Block_ID;
       POINTER     : in     System.Address;
       LENGTH      : in     natural;
       INDEX       :    out LWCS.Activity_Index);

procedure SEND_CONNECTIONLESS_BUFFER
     ( DESTINATION : in     LWAM.Address_ID;
       PROTOCOL    : in     PROTOCOL_SUITE;
       PARMS       : in     LWCS.Activity_Block_ID;
       MSG_CLASS   : in     LWCS.Message_Class;
       POINTER     : in     System.Address;
       LENGTH      : in     natural;
       INDEX       :    out LWCS.Activity_Index);

procedure GET_CONNECTIONLESS_BUFFER
     ( SOURCE       : in     LWAM.Address_ID;
       CONTROL      : in     LWCS.Activity_Block_ID;
       POINTER      : in     SYSTEM.Address;
       LENGTH       :    out natural;
       MSG_PRIORITY :    out LWCS.Priority;
       MSG_RCLASS   :    out LWCS.Message_Class;
       INDEX        :    out LWCS.Activity_Index);

procedure GET_CONNECTIONLESS_BUFFER
     ( SOURCE       : in     LWAM.Address_ID;
       CONTROL      : in     LWCS.Activity_Block_ID;
       MSG_CLASS    : in     LWCS.Message_Class;
       POINTER      : in     SYSTEM.Address;
       LENGTH       :    out natural;
       MSG_PRIORITY :    out LWCS.Priority;
       INDEX        :    out LWCS.Activity_Index);

procedure SEND_MULTICAST_BUFFER
     ( GROUP     : in     LWGM.Group_ID;
       PROTOCOL  : in     PROTOCOL_SUITE;
       CONTROL   : in     LWCS.Activity_Block_ID;
       POINTER   : in     SYSTEM.Address;
       LENGTH    : in     natural;
       INDEX     :    out LWCS.Activity_Index);

procedure SEND_MULTICAST_BUFFER
     ( GROUP     : in     LWGM.Group_ID;
       PROTOCOL  : in     PROTOCOL_SUITE;
```

```
        CONTROL   : in     LWCS.Activity_Block_ID;
        MSG_CLASS : in     LWCS.Message_Class;
        POINTER   : in     SYSTEM.Address;
        LENGTH    : in     natural;
        INDEX     :    out LWCS.Activity_Index);

   procedure SEND_MULTICAST_STREAM_BUFFER
        ( GROUP     : in     LWGM.Group_ID;
        PROTOCOL  : in     PROTOCOL_SUITE;
        CONTROL   : in     LWCS.Activity_Block_ID;
        POINTER   : in     SYSTEM.Address;
        LENGTH    : in     natural;
        PARAMS    : in     LWCM.Connection_Block_ID;
        INDEX     :    out LWCS.Activity_Index);

   procedure SEND_MULTICAST_STREAM_BUFFER
        ( GROUP     : in     LWGM.Group_ID;
        PROTOCOL  : in     PROTOCOL_SUITE;
        CONTROL   : in     LWCS.Activity_Block_ID;
        MSG_CLASS : in     LWCS.Message_Class;
        POINTER   : in     SYSTEM.Address;
        LENGTH    : in     natural;
        PARAMS    : in     LWCM.Connection_Block_ID;
        INDEX     :    out LWCS.Activity_Index);

   procedure GET_BUFFER_FOR_GROUP
        ( GROUP        : in     LWGM.Group_ID;
        CONTROL      : in     LWCS.Activity_Block_ID;
        POINTER      : in     SYSTEM.Address;
        LENGTH       :    out natural;
        MSG_PRIORITY :    out LWCS.Priority;
        MSG_RCLASS   :    out LWCS.Message_Class;
        RESULT       :    out STATUS);

   procedure GET_BUFFER_FOR_GROUP
        ( GROUP        : in     LWGM.Group_ID;
        CONTROL      : in     LWCS.Activity_Block_ID;
        MSG_CLASS    : in     LWCS.Message_Class;
        POINTER      : in     SYSTEM.Address;
        LENGTH       :    out natural;
        MSG_PRIORITY :    out LWCS.Priority;
        RESULT       :    out STATUS);

   procedure BROADCAST_BUFFER
        ( PARMS     : in     LWCS.Activity_Block_ID;
        MSG_CLASS : in     LWCS.Message_Class;
        POINTER   : in     SYSTEM.Address;
        LENGTH    : in     natural;
        INDEX     :    out LWCS.Activity_Index);

   procedure BROADCAST_BUFFER
```

```
        ( PARMS   : in     LWCS.Activity_Block_ID;
          POINTER : in     SYSTEM.Address;
          LENGTH  : in     natural;
          INDEX   :    out LWCS.Activity_Index);

   procedure GET_BROADCAST_BUFFER
        ( PARMS        : in     LWCS.Activity_Block_ID;
          MSG_PRIORITY : in     LWCS.Priority;
          POINTER      : in     SYSTEM.Address;
          LENGTH       :    out natural;
          MSG_RCLASS   :    out LWCS.Message_Class;
          RESULT       :    out STATUS );

   procedure GET_BROADCAST_BUFFER
        ( PARMS        : in     LWCS.Activity_Block_ID;
          MSG_PRIORITY : in     LWCS.Priority;
          MSG_CLASS    : in     LWCS.Message_Class;
          POINTER      : in     SYSTEM.Address;
          LENGTH       :    out natural;
          RESULT       :    out STATUS );

-------------------------------------------------------------
--    The function CONNECTIONLESS_BUFFER_LENGTH_VALID returns
--  true if and only if the specified buffer size is valid.
-------------------------------------------------------------
   function CONNECTIONLESS_BUFFER_LENGTH_VALID
        (length : in positive)
           return boolean;

-------------------------------------------------------------
--    The function MESSAGES_PENDING_ON_CONNECTION returns the
--  number of  messages that are pending a GET_MESSAGE request
--  for a specified connection.  Similar functions are defined
--  for connectionless receptions and group receptions.
-------------------------------------------------------------
   function MESSAGES_PENDING_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID )
             return natural;

   function MESSAGES_PENDING_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID;
          MSG_CLASS  : in LWCS.Message_Class)
             return natural;

   function MESSAGES_PENDING_FROM_SOURCE
        ( SOURCE : in LWAM.Address_ID )
             return natural;

   function MESSAGES_PENDING_FROM_SOURCE
        ( SOURCE    : in LWAM.Address_ID;
          MSG_CLASS : in LWCS.Message_Class)
```

```
            return natural;

   function MESSAGES_PENDING_FOR_GROUP
        ( GROUP : in LWGM.Group_ID )
            return natural;

   function MESSAGES_PENDING_FOR_GROUP
        ( GROUP     : in LWGM.Group_ID;
          MSG_CLASS : in LWCS.Message_Class)
            return natural;

   function MESSAGES_PENDING return natural;

   function MESSAGES_PENDING (MSG_CLASS : in LWCS.Message_Class)
                             return natural;

-------------------------------------------------------------

--    The procedure PURGE_MESSAGE_ON_CONNECTION will remove
--  a specified number of messages received on a particular
--  connection.  If ORDER is OLDEST_FIRST the oldest messages
--  will be deleted oldest first. Similar procedures are
--  defined for connectionless receipt and receipt from a
--  group. The procedure PURGE_ALL_MESSAGES will remove all
--  messages for an application from all sources.  The
--  procedure definitions are overloaded to permit the
--  deletion of messages received before the specified time.
-------------------------------------------------------------
   procedure PURGE_MESSAGES_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID;
          NUMBER     : in positive;
          ORDER      : in PURGE_ORDER);

   procedure PURGE_MESSAGES_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID;
          MSG_CLASS  : in LWCS.Message_Class;
          NUMBER     : in positive;
          ORDER      : in PURGE_ORDER);

   procedure PURGE_MESSAGES_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID;
          BEFORE     : in CALENDAR.time );

   procedure PURGE_MESSAGES_ON_CONNECTION
        ( CONNECTION : in LWCM.Connection_ID;
          MSG_CLASS  : in LWCS.Message_Class;
          BEFORE     : in CALENDAR.time );

   procedure PURGE_MESSAGES_FROM_SOURCE
        ( SOURCE     : in LWAM.Address_ID;
          NUMBER     : in positive;
```

```
       ORDER      : in PURGE_ORDER );

  procedure PURGE_MESSAGES_FROM_SOURCE
       ( SOURCE     : in LWAM.Address_ID;
         MSG_CLASS  : in LWCS.Message_Class;
         NUMBER     : in positive;
         ORDER      : in PURGE_ORDER );

  procedure PURGE_MESSAGES_FROM_SOURCE
       ( SOURCE     : in LWAM.Address_ID;
         BEFORE     : in CALENDAR.time );

  procedure PURGE_MESSAGES_FROM_SOURCE
       ( SOURCE     : in LWAM.Address_ID;
         MSG_CLASS  : in LWCS.Message_Class;
         BEFORE     : in CALENDAR.time );

  procedure PURGE_MESSAGES_FROM_GROUP
       ( GROUP      : in LWGM.Group_ID;
         NUMBER     : in positive;
         ORDER      : in PURGE_ORDER );

  procedure PURGE_MESSAGES_FROM_GROUP
       ( GROUP      : in LWGM.Group_ID;
         MSG_CLASS  : in LWCS.Message_Class;
         NUMBER     : in positive;
         ORDER      : in PURGE_ORDER );

  procedure PURGE_MESSAGES_FROM_GROUP
       ( GROUP      : in LWGM.Group_ID;
         BEFORE     : in CALENDAR.time );

  procedure PURGE_MESSAGES_FROM_GROUP
       ( GROUP      : in LWGM.Group_ID;
         MSG_CLASS  : in LWCS.Message_Class;
         BEFORE     : in CALENDAR.time );

  procedure PURGE_ALL_MESSAGES;

  procedure PURGE_ALL_MESSAGES (MSG_CLASS : in LWCS.Message_Class);

  procedure PURGE_ALL_MESSAGES
       ( BEFORE : in CALENDAR.time );

  procedure PURGE_ALL_MESSAGES
       ( BEFORE    : in CALENDAR.time;
         MSG_CLASS : in LWCS.Message_Class);

  procedure PURGE_ALL_BROADCAST_MESSAGES;

  procedure PURGE_ALL_BROADCAST_MESSAGES
```

```
        (MSG_CLASS : in LWCS.Message_Class);

   procedure PURGE_ALL_BROADCAST_MESSAGES
        ( BEFORE : in CALENDAR.time );

   procedure PURGE_ALL_BROADCAST_MESSAGES
        ( BEFORE    : in CALENDAR.time;
          MSG_CLASS : in LWCS.Message_Class);

private

-- implementation-defined

end LW_DATA_TRANSFER;
```

Appendix H: Transaction Services

```
------------------------------------------------------------
-- Package LW_TRANSACTION_SERVICES
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services.  The purpose of
--   this package is to provide functionality used in a request
--   response model of a transaction (such as an RPC).
--
------------------------------------------------------------
with LW_ADDRESS_MANAGEMENT;
with LW_COMMUNICATIONS_SUPPORT;
with LW_DATA_TRANSFER;

package LW_TRANSACTION_SERVICES is

   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;

   UNICAST_ERROR   : exception;

   type TRANSACTION_STATE is (IN_PROGRESS, COMPLETED);

------------------------------------------------------------
--    Define a buffer for use in transaction processing. The
--  length of the buffer is implementation-defined.
------------------------------------------------------------
   MAX_TRANSACTION_BUFFER_SIZE : CONSTANT := implementation-defined;
   type TRANSACTION_BUFFER_SIZE is range
          1 .. MAX_TRANSACTION_BUFFER_SIZE;
   type TRANSACTION_BUFFER is array
          (TRANSACTION_BUFFER_SIZE range <>) of
           LWCS.Unsigned_Byte;
   pragma pack (TRANSACTION_BUFFER);

   type TRANSACTION_DATA (SIZE : TRANSACTION_BUFFER_SIZE) is
     record
        DATA : TRANSACTION_BUFFER(1 .. SIZE);
     end record;

------------------------------------------------------------
--   The procedure INITIATE_UNICAST_TRANSACTION allows an
--  application to begin a transaction with another user in a
--  peer-to-peer manner.
------------------------------------------------------------
     procedure INITIATE_UNICAST_TRANSACTION
        ( DESTINATION    : in     LWAM.Address_ID;
          CONTROL        : in     LWCS.Activity_Block_ID;
          DATA           : in     TRANSACTION_DATA;
```

```
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

     procedure INITIATE_UNICAST_TRANSACTION
        ( DESTINATION    : in     LWAM.Address_ID;
          CONTROL        : in     LWCS.Activity_Block_ID;
          MSG_CLASS      : in     LWCS.Message_Class;
          DATA           : in     TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

-------------------------------------------------------------
--   The procedure GET_UNICAST_RESPONSE allows an application
-- to receive a response associated with a particular
-- identifier. A timeout parameter is included in the activity
-- block that specifies how long the application will wait for
-- the response.
-------------------------------------------------------------
     procedure GET_UNICAST_RESPONSE
        ( TRANSACTION_ID : in     LWCS.ACTIVITY_INDEX;
          CONTROL        : in     LWCS.Activity_Block_ID;
          DATA           :    out TRANSACTION_DATA;
          RESULT         :    out TRANSACTION_STATE);

-------------------------------------------------------------
--    The procedure ACCEPT_TRANSACTION_REQUEST allows an
--  application to receive a request to initiate a transaction
--  from an external source. A timeout parameter is included in
--  the activity block that specifies how long the application
--  will wait for the response.
-------------------------------------------------------------
    procedure ACCEPT_TRANSACTION_REQUEST
        ( SOURCE         : in     LWAM.Address_ID;
          CONTROL        : in     LWCS.Activity_Block_ID;
          DATA           :    out TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

    procedure ACCEPT_TRANSACTION_REQUEST
        ( SOURCE         : in     LWAM.Address_ID;
          CONTROL        : in     LWCS.Activity_Block_ID;
          MSG_CLASS      : in     LWCS.Message_Class;
          DATA           :    out TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

-------------------------------------------------------------
--    The procedure ACCEPT_ANY_TRANSACTION_REQUEST allows an
--  application to receive a request to initiate a transaction
--  from any source. A timeout parameter is included in the
--  activity block that specifies how long the application will
```

```
--  wait for the response.
-------------------------------------------------------------
    procedure ACCEPT_ANY_TRANSACTION_REQUEST
        ( CONTROL        : in     LWCS.Activity_Block_ID;
          SOURCE         :    out LWAM.Address_ID;
          DATA           :    out TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

    procedure ACCEPT_ANY_TRANSACTION_REQUEST
        ( CONTROL        : in     LWCS.Activity_Block_ID;
          MSG_CLASS      : in     LWCS.Message_Class;
          SOURCE         :    out LWAM.Address_ID;
          DATA           :    out TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

    procedure ACCEPT_ANY_TRANSACTION_REQUEST
        ( CONTROL        : in     LWCS.Activity_Block_ID;
          RMSG_CLASS     :    out LWCS.Message_Class;
          SOURCE         :    out LWAM.Address_ID;
          DATA           :    out TRANSACTION_DATA;
          TRANSACTION_ID :    out LWCS.Activity_Index;
          RESULT         :    out TRANSACTION_STATE);

-------------------------------------------------------------
--    The procedure RESPOND_TO_TRANSACTION_REQUEST can be used
--  by an application to respond to a previously received
--  transaction request.
-------------------------------------------------------------
    procedure RESPOND_TO_TRANSACTION_REQUEST
        ( TRANSACTION_ID : in     LWCS.Activity_Index;
          CONTROL        : in     LWCS.Activity_Block_ID;
          DATA           : in     TRANSACTION_DATA;
          RESULT         :    out TRANSACTION_STATE );

-------------------------------------------------------------
--   The function TRANSACTION_RESPONSE_PRESENT allows an
--  application to determine if a transaction response is
--  available.  A timeout can also be specified, in which
--  case the application will block pending receipt of the
--  response.
-------------------------------------------------------------
     function TRANSACTION_RESPONSE_PRESENT
        ( TRANSACTION_ID : in LWCS.Activity_Index;
          TIMEOUT        : in duration)
             return boolean;

-------------------------------------------------------------
--   The function NUMBER_OF_TRANSACTIONS returns the number of
--  transactions that are in progress.
```

```
-------------------------------------------------------------
     function NUMBER_OF_TRANSACTIONS
        return natural;

     function NUMBER_OF_TRANSACTIONS
        (MSG_CLASS : in LWCS.Message_Class)
        return natural;

-------------------------------------------------------------
--   The procedure CANCEL_TRANSACTION will cancel a specified
--  transaction. The procedure CANCEL_ALL_TRANSACTIONS deletes
--  all transactions.
-------------------------------------------------------------
     procedure CANCEL_TRANSACTION
        (TRANSACTION_ID : in LWCS.Activity_Index);

     procedure CANCEL_ALL_TRANSACTIONS;

     procedure CANCEL_ALL_TRANSACTIONS
        (MSG_CLASS : in LWCS.Message_Class);
-------------------------------------------------------------

private

--implementation-defined

end LW_TRANSACTION_SERVICES;
```

Appendix I: Event Management

```
------------------------------------------------------------
-- Package: LW_EVENT_MANAGEMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services. The purpose of
--   this package is to provide support for event management.
--   Subprograms are provided to register for an event,
--   deregister for an event, obtain the number of events
--   present, and to get the data for an event. It is through
--   this package that the user can configure the error
--   detection capabilities provided by the Ada binding.
--
------------------------------------------------------------
with LW_ADDRESS_MANAGEMENT;
with LW_COMMUNICATIONS_SUPPORT;
with LW_CONNECTION_MANAGEMENT;
with LW_GROUP_MANAGEMENT;
with CALENDAR;

  package LW_EVENT_MANAGEMENT is

    package LWAM renames LW_ADDRESS_MANAGEMENT;
    package LWCS renames LW_COMMUNICATIONS_SUPPORT;
    package LWCM renames LW_CONNECTION_MANAGEMENT;
    package LWGM renames LW_GROUP_MANAGEMENT;

------------------------------------------------------------
--   There are two exceptions that can be exported from this
--  package. INVALID_PARAMETER is raised if user data
--  is erroneous (such as an invalid priority).  The
--  exception INVALID_OPERATION is raised if the user requests
--  an operation that cannot be performed (such as making a
--  GET_EVENT_DATA request for a connection which has not
--  been previously registered).
------------------------------------------------------------
  INVALID_PARAMETER : exception;
  INVALID_OPERATION : exception;

------------------------------------------------------------
--    The following type defines the classes for which an
--  event may be registered.
------------------------------------------------------------
   type EVENT_CLASS is (CONNECTION, GROUP, UNICAST, MULTICAST,
          TRANSACTION);

------------------------------------------------------------
--    The type EVENT_ACTION is used when requesting events; an
--  object of this type may be registered or deregistered.
------------------------------------------------------------
```

```
   type EVENT_ACTION is (REGISTER, DEREGISTER);

-------------------------------------------------------------
--    The following type is used when an application wants to
--  get the data associated with an event.  Mechanisms are
--  provided to return an event in order of priority, oldest,
--  or newest event occurrence.
-------------------------------------------------------------
   type ORDER is (PRIORITIZED, OLDEST, NEWEST);

-------------------------------------------------------------
--    The type EVENT_STATUS is used to indicate whether or not
--  an event is present.  It is used by procedures that get
--  event data.
-------------------------------------------------------------
   type EVENT_STATUS is (AVAILABLE, NOT_AVAILABLE);

-------------------------------------------------------------
--    The following types are declared for the class of events
--  related to a CONNECTION.
-------------------------------------------------------------
   type CONNECTION_SUBCLASS is (CLOSED, FAIL, ERROR);

   type CONNECTION_CLOSE_REASON is (GRACEFUL, IMMEDIATE);

   type CONNECTION_FAIL_REASON is (TIMEOUT,
           CONNECTION_FAILED_OTHER);

   type CONNECTION_ERROR_REASON is (RESET, READ_SIDE_FAIL,
           WRITE_SIDE_FAIL, REFUSED, CONNECTION_ERROR_UNKNOWN,
           CONNECTION_ERROR_OTHER);
-------------------------------------------------------------
--    The type CONNECTION_EVENT_DATA is used to return
--  information about an event. It contains the time of the
--  event, the name of the connection, and the reason the event
--  was generated.
-------------------------------------------------------------

   type CONNECTION_EVENT_DATA (C: CONNECTION_SUBCLASS) is
     record
        EVENT_TIME : CALENDAR.time;
        CONNECTION : LWCM.Connection_ID;
        -- implementation-defined data
          case C is
            when CLOSED =>
                A : CONNECTION_CLOSE_REASON;
            when FAIL   =>
                F : CONNECTION_FAIL_REASON;
            when ERROR  =>
                E : CONNECTION_ERROR_REASON;
          end case;
```

```
     end record;


------------------------------------------------------------
--    The following types are used to declare event information
--  for the group event class.
------------------------------------------------------------
   type GROUP_SUBCLASS is (GROUP_RELATED, MEMBER_RELATED,
           ERROR_RELATED, OTHER_GROUP_RELATED);

   type GROUP_RELATED_REASON is (GROUP_CLOSED, GROUP_EMPTY,
           GROUP_FULL, GROUP_UNKNOWN);

   type MEMBER_RELATED_REASON is (MEMBER_ADDED, MEMBER_REMOVED,
           CLOSE_PERMISSION_CHANGE, OTHER_MEMBER_RELATED_REASON);

   type ERROR_RELATED_REASON is (INVALID_GROUP_NAME,
           INVALID_MEMBER_NAME, UNKNOWN_GROUP,
           NO_PERMISSION_TO_JOIN_GROUP,
           GROUP_ERROR_OTHER);

   type OTHER_GROUP_RELATED_REASON is implementation-defined;

   type GROUP_EVENT_DATA (G : GROUP_SUBCLASS) is
     record
        EVENT_TIME : CALENDAR.time;
        GROUP      : LWGM.Group_ID;
        -- implementation-defined data
           case G is
             when GROUP_RELATED  =>
                NAME : LWCS.Logical_Name;
                R    : GROUP_RELATED_REASON;
             when MEMBER_RELATED =>
                ELEMENT : LWCS.Logical_Name;
                M       : MEMBER_RELATED_REASON;
             when ERROR_RELATED  =>
                err : ERROR_RELATED_REASON;
             when OTHER_GROUP_RELATED =>
                implementation-defined;
           end case;
     end record;

------------------------------------------------------------
--    The following types are used to declare event information
--  for the unicast event class.
------------------------------------------------------------
   type UNICAST_SUBCLASS is (CONNECTION, CONNECTIONLESS,
           ERROR);

   type CONNECTION_REASON is (INVALID_SERVICE_PARAMETER,
          UNKNOWN_CONNECTION, INVALID_BUFFER_LENGTH,
```

```
         CONNECTION_OTHER);

   type CONNECTIONLESS_REASON is (UNKNOWN_NAME,
          INVALID_PROTOCOL_SUITE, CONNECTIONLESS_OTHER);

   type ERROR_REASON is (INVALID_NAME,
                         UNICAST_ERROR_OTHER);

   type UNICAST_EVENT_DATA (U : UNICAST_SUBCLASS) is
     record
        EVENT_TIME : CALENDAR.time;
        -- implementation-defined data
         case U is
           when CONNECTION     =>
             CN_NAME : LWCM.Connection_ID;
             U1      : CONNECTION_REASON;

           when CONNECTIONLESS =>
             CL_NAME : LWCS.Logical_Name;
             U2      : CONNECTIONLESS_REASON;
           when ERROR          =>
             E : ERROR_REASON;
         end case;
     end record;

-----------------------------------------------------------------
--    The following types are used to declare event information
--  for the multicast event class.
-----------------------------------------------------------------
   type MULTICAST_SUBCLASS is ( NUMBER_DELIVERY_RELATED,
        NAMED_DELIVERY_RELATED, ERROR_RELATED,
        CONNECTION_CLOSED, CONNECTION_FAILED,
        CONNECTION_ERROR);

   type MULTICAST_ERROR_REASON is
          (MULTICAST_INVALID_BUFFER_LENGTH,
           MULTICAST_INVALID_GROUP_NAME,
           MULTICAST_UNKNOWN_GROUP,
           MULTICAST_INVALID_PROTOCOL,
           MULTICAST_ERROR_OTHER);

   type RECEPTION_STATE is array (1 .. LWGM.MAX_MEMBERS)
          of boolean;

   type MULTICAST_EVENT_DATA (M: MULTICAST_SUBCLASS) is
     record
        EVENT_TIME : CALENDAR.time;
        GROUP      : LWGM.Group_ID;
        -- implementation-defined data
          case M is
             when NUMBER_DELIVERY_RELATED =>
```

```
               NBR_RECEIVED     : natural;
               NBR_NOT_RECEIVED : natural;
            when NAMED_DELIVERY_RELATED =>
               NAME_RECEIVED : natural;
               NAME_NOT_RECEIVED : positive;
               WHO : RECEPTION_STATE;
            when ERROR_RELATED =>
              ERR:  MULTICAST_ERROR_REASON;
            when CONNECTION_CLOSED =>
              A : CONNECTION_CLOSE_REASON;
            when CONNECTION_FAILED =>
              B : CONNECTION_FAIL_REASON;
            when CONNECTION_ERROR =>
              C : CONNECTION_ERROR_REASON;
          end case;
     end record;

-------------------------------------------------------------
--    The following types are used to declare event information
--  for the transaction event class.
-------------------------------------------------------------
   type TRANSACTION_SUBCLASS is (ERROR_RELATED,
          TRANSACTION_OTHER);

   type TRANSACTION_EVENT_DATA (T: TRANSACTION_SUBCLASS) is
     record
        EVENT_TIME : CALENDAR.time;
        -- implementation-defined data
     end record;

-------------------------------------------------------------
--    The procedure REQUEST_EVENT allows an application
--  to register or deregister events.  Events can be
--  specified for a class; a class and subclass; or a class,
--  subclass, and entity.  A priority is also included.
-------------------------------------------------------------
   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          PRIORITY : in LWCS.Priority );

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          CLASS    : in EVENT_CLASS;
          PRIORITY : in LWCS.Priority );

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in CONNECTION_SUBCLASS);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
```

```
          SUBCLASS : in CONNECTION_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in GROUP_SUBCLASS);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in GROUP_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in UNICAST_SUBCLASS);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in UNICAST_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in MULTICAST_SUBCLASS);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in MULTICAST_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in TRANSACTION_SUBCLASS);

   procedure REQUEST_EVENT
        ( ACTION   : in EVENT_ACTION;
          SUBCLASS : in TRANSACTION_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name);


------------------------------------------------------------
--    The function EVENT_PRESENT returns true if and only if
--  there is at least one event pending. The procedure is
--  overloaded to indicate the presence of an event associated
--  with a particular subclass and/or entity. A timeout
--  parameter is included to allow an application to wait for
--  an event to be registered.
------------------------------------------------------------
  function EVENT_PRESENT
        ( INTERVAL : in duration)
            return boolean;
```

```
function EVENT_PRESENT
      ( CLASS    : in EVENT_CLASS;
        INTERVAL : in duration)
          return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in CONNECTION_SUBCLASS;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in CONNECTION_SUBCLASS;
        ENTITY   : in LWCS.Logical_Name;
        INTERVAL : in duration)
           return boolean;
function EVENT_PRESENT
      ( SUBCLASS : in GROUP_SUBCLASS;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in GROUP_SUBCLASS;
        ENTITY   : in LWCS.Logical_Name;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in UNICAST_SUBCLASS;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in UNICAST_SUBCLASS;
        ENTITY   : in LWCS.Logical_Name;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in MULTICAST_SUBCLASS;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in MULTICAST_SUBCLASS;
        ENTITY   : in LWCS.Logical_Name;
        INTERVAL : in duration)
           return boolean;

function EVENT_PRESENT
      ( SUBCLASS : in TRANSACTION_SUBCLASS;
        INTERVAL : in duration)
```

```
            return boolean;

  function EVENT_PRESENT
        ( SUBCLASS : in TRANSACTION_SUBCLASS;
          ENTITY   : in LWCS.Logical_Name;
          INTERVAL : in duration)
             return boolean;


------------------------------------------------------------
--    The function NUMBER_OF_EVENTS_PRESENT returns the number
--  of pending events. The function is overloaded to also
--  return the number of events of a particular class; a
--  class and subclass; or a class, subclass, and entity.
------------------------------------------------------------
  function NUMBER_OF_EVENTS_PRESENT
        return natural;

  function NUMBER_OF_EVENTS_PRESENT
        (CLASS: in EVENT_CLASS)
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        (SUBCLASS : in CONNECTION_SUBCLASS)
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        (SUBCLASS : in CONNECTION_SUBCLASS;
          ELEMENT  : in LWCS.Logical_Name )
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        (SUBCLASS : in GROUP_SUBCLASS)
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in GROUP_SUBCLASS;
          ELEMENT  : in LWCS.Logical_Name )
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in UNICAST_SUBCLASS)
            return natural;


  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in UNICAST_SUBCLASS;
          ELEMENT  : in LWCS.Logical_Name )
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
```

```
        ( SUBCLASS : in MULTICAST_SUBCLASS)
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in MULTICAST_SUBCLASS;
          ELEMENT  : in LWCS.Logical_Name )
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in TRANSACTION_SUBCLASS)
            return natural;

  function NUMBER_OF_EVENTS_PRESENT
        ( SUBCLASS : in TRANSACTION_SUBCLASS;
          ELEMENT  : in LWCS.Logical_Name )
            return natural;

-----------------------------------------------------------------
-- The function GET_EVENT_CLASS returns the event class of any
-- event.
-----------------------------------------------------------------
  function GET_EVENT_CLASS
        ( INTERVAL : in duration)
            return EVENT_CLASS;

-----------------------------------------------------------------
--    The procedures GET_EVENT_DATA returns the data
--  associated with the specified class or subclass of events.
--  If REMOVE is true, the event will be deleted.
--  A timeout is included that allows an application to bound
--  the length of time to wait for the receipt of the event.
-----------------------------------------------------------------
   procedure GET_EVENT_DATA
        ( SUBCLASS : in     CONNECTION_SUBCLASS;
          TIMEOUT  : in     duration;
          DATA     :    out CONNECTION_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS);

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     CONNECTION_SUBCLASS;
          ENTITY   : in     LWCS.Logical_Name;
          TIMEOUT  : in     duration;
          DATA     :    out CONNECTION_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     GROUP_SUBCLASS;
          TIMEOUT  : in     duration;
          DATA     :    out GROUP_EVENT_DATA;
```

```
        REMOVE   : in     boolean;
        RESULT   :    out EVENT_STATUS );


   procedure GET_EVENT_DATA
        ( SUBCLASS : in     GROUP_SUBCLASS;
          ENTITY   : in     LWCS.Logical_Name;
          TIMEOUT  : in     duration;
          DATA     :    out GROUP_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     UNICAST_SUBCLASS;
          TIMEOUT  : in     duration;
          DATA     :    out UNICAST_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     UNICAST_SUBCLASS;
          ENTITY   : in     LWCS.Logical_Name;
          TIMEOUT  : in     duration;
          DATA     :    out UNICAST_EVENT_DATA;
          REMOVE   : in     boolean);

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     MULTICAST_SUBCLASS;
          TIMEOUT  : in     duration;
          DATA     :    out MULTICAST_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     MULTICAST_SUBCLASS;
          ENTITY   : in     LWCS.Logical_Name;
          TIMEOUT  : in     duration;
          DATA     :    out MULTICAST_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

   procedure GET_EVENT_DATA
        ( SUBCLASS : in     TRANSACTION_SUBCLASS;
          TIMEOUT  : in     duration;
          DATA     :    out TRANSACTION_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );


   procedure GET_EVENT_DATA
        ( SUBCLASS : in     TRANSACTION_SUBCLASS;
```

```
          ENTITY   : in     LWCS.Logical_Name;

          TIMEOUT  : in     duration;
          DATA     :    out TRANSACTION_EVENT_DATA;
          REMOVE   : in     boolean;
          RESULT   :    out EVENT_STATUS );

private

-- implementation-defined

end LW_EVENT_MANAGEMENT;
```

Appendix J: Buffer Management

```
------------------------------------------------------------------
-- Package: LW_BUFFER_MANAGEMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services. The package
--   contains operations related to buffer management for use
--   in data transfers.  The model is based on the following:
--
--      o  An application can create or delete a buffer pool.
--
--      o  For a given buffer pool, an application can allocate
--   or deallocate a buffer of a specified length (in bytes).
--
--     An application can request a particular alignment when
--   allocating a buffer, such as alignment on a 32-bit word
--   boundary.
--
------------------------------------------------------------------

with SYSTEM;
package LW_BUFFER_MANAGEMENT is

   type POOL_ID is private;

------------------------------------------------------------------
--    The following type specifies the buffer alignments
--  provided for an application.  Provision is made for byte,
--  16-bit, 32-bit, or 64-bit alignments.
------------------------------------------------------------------
   type ALIGNMENT is (BYTE, DOUBLE_BYTE, QUAD_BYTE, OCT_BYTE);

------------------------------------------------------------------
--    The following type specifies the manner in which a buffer
--  pool may be released.  In a graceful manner, a pool may not
--  be released if there are buffers still allocated.  In an
--  immediate manner, the pool will be released regardless of
--  state of buffers allocated from this pool.
------------------------------------------------------------------
   type RELEASE is (GRACEFUL, IMMEDIATE);

------------------------------------------------------------------
--    The following exceptions are raised for the indicated
--  reason.
------------------------------------------------------------------
   BUFFER_MANAGEMENT_NOT_INITIALIZED : exception;
   INVALID_POOL                      : exception;
   INVALID_BUFFER                    : exception;
   INVALID_OPERATION                 : exception;
```

```
------------------------------------------------------------
--    The procedure INITIALIZE_BUFFER_MANAGEMENT performs
--  initialization operations for buffer management.  This
--  procedure must be called before any buffer management
--  routines are called.
------------------------------------------------------------
   procedure INITIALIZE_BUFFER_MANAGEMENT;

------------------------------------------------------------
--    The following subprograms are provided for operations
--  on buffer pools.
------------------------------------------------------------
   procedure ALLOCATE_POOL
        ( SIZE     : in     positive;
          BOUNDARY : in     ALIGNMENT;
          ID       :    out POOL_ID);

   procedure DEALLOCATE_POOL
        ( ID    : in POOL_ID;
          MEANS : in RELEASE);

   procedure INCREASE_POOL_SIZE
        ( ID     : in POOL_ID;
          AMOUNT : in positive);

   procedure DECREASE_POOL_SIZE
        ( ID     : in POOL_ID;
          AMOUNT : in positive);

   function BUFFER_POOL_SIZE
        ( ID: in POOL_ID)
                return positive;

   function BUFFER_SPACE_ALLOCATED
        ( ID : in POOL_ID)
                return natural;

   function BUFFER_SPACE_AVAILABLE
        ( ID: in POOL_ID)
                return positive;

------------------------------------------------------------
--    The following subprograms allocate or deallocate buffers
--  from the specified pool.
------------------------------------------------------------
   procedure GET_BUFFER
        ( POOL     : in     POOL_ID;
          SIZE     : in     positive;
          BOUNDARY : in     ALIGNMENT;
          POINTER  :    out SYSTEM.Address);
```

```
   procedure RELEASE_BUFFER

         ( POOL    : in POOL_ID;
           POINTER : in SYSTEM.Address;
           MEANS   : in RELEASE);
private

  -- implementation-defined

end LW_BUFFER_MANAGEMENT;
```

Appendix K: Performance_Measurement

```
------------------------------------------------------------
-- Package: LW_PERFORMANCE_MEASUREMENT
--
-- Purpose: This package is part of the Ada binding to the
--   SAFENET lightweight application services.  The package
--   exports subprograms that provide support for the
--   performance measurement capability. This capability is
--   defined for two classes of activities.
--
--     o  Data transfer, such as connectionless transfer or
--       multicast. In addition, the user can specify a
--       mode, either input, output, or both.
--
--     o  Events defined in relation to the classes in package
--       LW_EVENT_MANAGEMENT.
--
--     An implementation can specify the information recorded
--  and the recording mechanism. An implementation can also add
--  implementation-dependent options to the package
--  specification.
--
--
------------------------------------------------------------

package LW_PERFORMANCE_MEASUREMENT is

   type TRANSFER_CLASS is (BROADCAST, CONNECTIONLESS_TRANSFER,
        CONNECTION_ORIENTED_TRANSFER, MULTICAST, TRANSACTION);

   type MODE is (SEND, RECEIVE, BOTH);

   type PERFORMANCE_CLASS is (CONNECTION, GROUP, UNICAST,
        MULTICAST, TRANSACTION);

------------------------------------------------------------
--    The procedures ENABLE_MEASUREMENT and DISABLE_MEASUREMENT
--  perform the indicated operation for a specified transfer
--  class and mode. The function MEASUREMENT_ENABLED will
--  return true if and only if  performance measurement is
--  enabled for the specified transfer class.
------------------------------------------------------------
   procedure ENABLE_MEASUREMENT
        ( C : in TRANSFER_CLASS;
          M : in MODE);

   procedure DISABLE_MEASUREMENT
        ( C : in TRANSFER_CLASS;
          M : in MODE);
```

```
   function MEASUREMENT_ENABLED
        ( C : in TRANSFER_CLASS;
          M : in MODE)
                return boolean;

-----------------------------------------------------------------
--    The procedures ENABLE_EVENTS and DISABLE_EVENTS will
--  perform the indicated operation for the specified event
--  class.  The function EVENTS_ENABLED will return true if and
--  only if events are enabled for the specified class.
-----------------------------------------------------------------
   procedure ENABLE_EVENTS (E : in PERFORMANCE_CLASS);

   procedure DISABLE_EVENTS (E : in PERFORMANCE_CLASS);

   function EVENTS_ENABLED (E : in PERFORMANCE_CLASS)
        return boolean;

end LW_PERFORMANCE_MEASUREMENT;
```

Appendix L: API Implementation Notes

In this appendix we discuss some of the issues involved in the implementation of the Ada binding in other languages, particularly the C language. This appendix is provided as a guideline to an implementor and will provide rationale for an Ada implementation, discussed in Section 3.5. We assume the presence of an Ada runtime system without requiring additional operating system support, such as POSIX. Three different implementation schemes are considered.

1. Full Ada implementation (i.e., package specification and body).
2. Ada package specification only.
3. Ada package specification and body, with the package body providing the interface to another language.

The discussion is presented by use of an example. Assume that there is an Ada package that exports a procedure *Check_Activity*. The purpose of the procedure is to determine if there has been any activity associated with a specified connection within a certain amount of time. A procedure such as this could be used to determine if an error condition exists for the specified connection.

A package specification for a full Ada implementation is presented in Figure L-1. The timeout condition is specified as an object of type *duration* and an exception *Timeout_Expired* will be raised if there has been no activity for the specified connection during the timeout period.

```
package CONNECTION_SUPPORT is

   TIMEOUT_EXPIRED : exception;

   type CONNECTION_STATE is
      (UNKNOWN, OPEN, CLOSED, ERROR);

   type CONNECTION_ID is range 0 .. 128;

   procedure CHECK_ACTIVITY
      (C       : in     CONNECTION_ID;
       TIMEOUT : in     duration;
       S       : in out CONNECTION_STATE);

end CONNECTION_SUPPORT;
```

Figure L-1: Basic Ada Implementation for Connection Timeout

In the second case, we assume an Ada package specification without an Ada package

body. Such an implementation is presented in Figure L-2, where the implementation is in the C language.

```
package C_SUPPORT is

   type C_TIMEOUT is range 0 .. 500;
   for  C_TIMEOUT'SIZE use 32;

end C_SUPPORT;

with C_SUPPORT;
package CONNECTION_SUPPORT is

   TIMEOUT_EXPIRED : exception;
   type CONNECTION_STATE is
      (UNKNOWN, OPEN, CLOSED, ERROR);
   type CONNECTION_ID is range 0 .. 128;

   procedure CHECK_ACTIVITY
      (C       : in     CONNECTION_ID;
       TIMEOUT : in     C_SUPPORT.C_TIMEOUT;
       S       : in out CONNECTION_STATE);

private

   for CONNECTION_STATE use
        (UNKNOWN => 0, OPEN => 1, CLOSED => 2, ERROR => 3);
   for CONNECTION_STATE'SIZE use 32;
   for CONNECTION_ID'SIZE use 32;

   pragma interface (C, CHECK_ACTIVITY);

end CONNECTION_SUPPORT;
```

Figure L-2: Ada Implementation for Connection Timeout Without Package Body

Several points are worth noting about the implementation presented in Figure L-2.

- A package *C_Support* defines types that support the Ada-to-C binding. The timeout parameter is declared and represented in 32 bits. This package is ''withed'' to provide the necessary support for the timeout parameter in the procedure *Check_Activity*.
- The private part of the specification declares an enumeration representation clause for the connection state, as well as a size specification indicating 32 bits.
- The private part of the package also defines the size of the connection identifier as 32 bits.

- The private part of the package specification includes a pragma interface, indicating that the procedure *Check_Activity* will be implemented in the C language.

Note that the use of the enumeration representation clause and the size attribute are required when the underlying language is other than Ada. This enforces a contractual arrangement between the two languages and defines data representations for the implementation language. This is a consequence of Ada's visibility rules and the use of packages. There are several issues associated with the binding specified in Figure L-2. Of the more major issues, the following are to be noted:[39]

- The user of the package must perform a conversion of an Ada type *duration* to the corresponding type used in the implementation, which is assumed to represent an integer number of milliseconds. This is required because the C language does not provide support for fixed-point types.
- The implementation language is visible in the package specification. This is due to the presence of the pragma interface, and the package that is ''withed'' to provide the necessary information for the implementation.
- It is not at all clear that the C language implementation of the procedure *Check_Activity* is capable of raising an exception. This requires that there is visibility into the Ada runtime system.

Some of the difficulties associated with the implementation shown in Figure L-2 may be removed by using an Ada package body. This option is presented in Figure L-3. In this case, Ada constructs are used in the body, such as the raising of the exception.

The following points are to be noted about the code shown in Figure L-3:

- The package specification carries no information about the implementation language. For example, either the C language or assembler language could be used for the implementation.
- The package specification provides the information about the enumeration representation clause and the size attributes. However, as noted above, this information is required when the implementation is in some language other than Ada.
- The user can still refer to an object of type *duration*. It is the responsibility of the implementation to perform the necessary conversions.[40]
- The exception is raised from the Ada code in the package body by using a Boolean variable, returned from the C procedure, which is then checked to determine if the timeout has expired.

When the Ada binding will be implemented in some language other than Ada, one is re-

[39]We do not discuss hardware-related issues, such as the mechanism that the implementation employs to obtain access to timers, which may conflict with the Ada runtime system!

[40]Note that the Ada type *duration* is implementation-defined.

```
package CONNECTION_SUPPORT is

   timeout_expired: EXCEPTION;
   type CONNECTION_STATE is
      (UNKNOWN, OPEN, CLOSED, ERROR);
   type CONNECTION_ID is range 0 .. 128;

   procedure CHECK_ACTIVITY
      (C       : in CONNECTION_ID;
       TIMEOUT : in DURATION;
       S       : in out CONNECTION_STATE);
private
   for CONNECTION_STATE use
        (UNKNOWN => 0, OPEN => 1, CLOSED => 2, ERROR => 3);
   for CONNECTION_STATE'SIZE use 32;
   for CONNECTION_ID'SIZE use 32;
end CONNECTION_SUPPORT;

package body CONNECTION_SUPPORT is
   type C_TIME is range 0 .. 500;
   C_VALUE : C_TIME;
   err : boolean := false;
   procedure EXTRACT_DURATION (TIMEOUT: in     duration;
                               C_VALUE:    out C_TIME);
   pragma interface (c, EXTRACT_DURATION);
   procedure C_CHECK (C      : in     CONNECTION_ID;
                      C_VALUE: in     C_TIME;
                      ERR    :    out boolean);
   pragma interface (C, C_CHECK);

   procedure CHECK_ACTIVITY
              (C      : in     CONNECTION_ID;
               TIMEOUT: in     duration;
               S      : in out CONNECTION_STATE) is
   begin
      EXTRACT_DURATION (TIMEOUT, C_VALUE);
      C_CHECK (C, C_VALUE, ERR);
      if ERR = true
      then
          raise TIMEOUT_EXPIRED;
      end if;
   end CHECK_ACTIVITY;

end CONNECTION_SUPPORT;
```

Figure L-3: Ada Implementation for Connection Timeout Using Package Body

quired to perform the implementation in a manner suggested by Figure L-2 or Figure L-3. In

the latter case the resulting implementation is believed somewhat more portable, and there is a cleaner interface to the underlying language. Additionally, the use of an Ada package body permits the use of Ada features that may not be available in the implementation language.

The result of the preceding is that an implementation in another language may be achieved by the existence of a package body that provides interface functions (to the other language). The use of a package body satisfies the requirements placed upon an implementor, discussed in Section 3.6.

Appendix M: Alternate Event Management Specifications

M.1. Introduction

In this appendix we discuss several alternate specifications for the event management package.

The goal of the original and the alternate event management specifications is to allow the application to choose which events to process. Each specification must therefore provide a method for the application to

- Identify the events that are important to the application.
- Determine when those events occur.
- Determine the number of pending events.
- Retrieve the information associated with those events.

M.2. Original Specification

The original specification contained in Appendix I meets its goal by defining

- Enumerated event types to partition events.
- Event data records to associate information with events.
- A procedural interface to allow the application to select important events and retrieve information about them.

M.2.1. Event Types

Events can be partitioned into classes based on the type of the operation that generated the event. The original specification accordingly defines *Event_Class* as an enumerated type containing the elements *Connection*, *Group*, *Unicast*, *Multicast*, and *Transaction*. Events in a given class can be further partitioned into subclasses based on the state of that event's associated operation at the time the event was generated. The specification accordingly defines appropriate *subclass* enumeration types. For example, for the event class *Connection*, the subclass *Connection_Subclass* contains the elements *Closed*, *Fail*, and *Error*. Finally, events in a particular class and subclass can be further partitioned based on the reason that the event was generated. The specification therefore defines various *reason* enumeration types. In our previous example, the type *Connection_Close_Reason*, which contains the values *Graceful* and *Immediate*, is defined for the *Event_Class Connection* and *Connection_Subclass Closed*.

M.3. Event Data Records

Each event has information associated with it, such as the time at which the event was generated. Some event information depends on the class and subclass of the event. For example, a *Connection* event has an associated connection identifier. The specification therefore defines a different type of event data record for each event class. Each type of event record is then discriminated by the corresponding event subclass and contains the event information relevant for that class and subclass of event.

M.3.1. Procedural Interface

The subprograms in the specification in Appendix I provide the application with the ability to

- register and deregister for,
- test for the presence of,
- determine the number of, and
- retrieve information about

an event, a class of events, or a class and subclass of events. This provides the application with the ability to process event data for

- all events,
- all events of a particular class, and
- all events of a particular class and subclass.

The procedural interface contained in Appendix I defines 47 subprograms. There are 12 for registering and deregistering for events, 12 for determining the presence of an event, 12 for determining the number of events present, 1 for determining the class of an event, and 10 for retrieving the information for an event.

M.4. Alternate Specifications

M.4.1. Introduction

The alternate specifications are based on different views of the event classes, subclasses, and associated reasons defined in the package *LW_Event_Management* (see Appendix I). In each alternate specification, only one type of event data record is defined. This reduces the size of the procedural interface from 47 subprograms in the original specification to 13 in each of the alternate specifications.

The first section discusses the procedural interface that is common to the three alternate specifications. The last section then discusses the type declarations of those alternate specifications.

M.5. Procedural Interfaces

The procedural interfaces associated with these alternate declarations of event data contain only 13 subprograms. Note that because there is only one type of event data record, there is no need for the *Get_Message_Class* function. The procedural interface for each of the alternate specification is as follows:

```
procedure REQUEST_EVENT
     ( ACTION   : in EVENT_ACTION;
       CLASS    : in EVENT_CLASS;
       PRIORITY : in LWCS.Priority);

procedure REQUEST_EVENT
     ( ACTION   : in EVENT_ACTION;
       E        : in EVENT);

procedure REQUEST_EVENT
     ( ACTION   : in EVENT_ACTION;
       E        : in EVENT;
       ENTITY   : in LWCS.Logical_Name);

function EVENT_PRESENT
      ( INTERVAL : in duration)
          return boolean;

function EVENT_PRESENT
      ( CLASS    : in EVENT_CLASS;
        INTERVAL : in duration)
          return boolean;

function EVENT_PRESENT
      ( E        : in EVENT;
        INTERVAL : in duration)
          return boolean;

function EVENT_PRESENT
      ( E        : in EVENT;
        ENTITY   : in LWCS.Logical_Name;
        INTERVAL : in duration)
          return boolean;

function NUMBER_OF_EVENTS_PRESENT
      return natural;

function NUMBER_OF_EVENTS_PRESENT
      ( CLASS    : in EVENT_CLASS)
          return natural;

function NUMBER_OF_EVENTS_PRESENT
      ( E        : in EVENT)
          return boolean;
```

```
function NUMBER_OF_EVENTS_PRESENT
      ( E      : in EVENT;
        ENTITY : in LWCS.Logical_Name)
          return natural;

procedure GET_EVENT_DATA
     ( E       : in     EVENT;
       TIMEOUT : in     duration;
       REMOVE  : in     boolean;
       DATA    :    out EVENT_DATA;
       RESULT  :    out EVENT_STATUS);

procedure GET_EVENT_DATA
     ( E       : in     EVENT;
       ENTITY  : in     LWCS.Logical_Name;
       TIMEOUT : in     duration;
       REMOVE  : in     boolean;
       DATA    :    out EVENT_DATA;
       RESULT  :    out EVENT_STATUS);
```

M.6. Type Declarations

This section discusses the different type declarations for the three alternate specifications for the *LW_Event_Management* package. Each specification contain a single declaration for the type *Event_Data*.

M.6.1. Alternate Specification 1

The first alternate specification for event management combines the *Event_Class* type declaration and the event *subclass* type declarations (such as *Connection_Subclass*, *Unicast_Subclass*) into a new type declaration named *Event*. This permits the definition of only one event data record type, *Event_Data*, which is discriminated by the type *Event*. While the *Event_Data* record type is quite complex, it replaces the five different *event data* record types from the original specification.

The type declarations for this alternate specification are

```
type EVENT_CLASS is (CONNECTION, GROUP, UNICAST, MULTICAST,
                     TRANSACTION);

type EVENT_ACTION is (REGISTER, UNREGISTER);

type ORDER is (PRIORITIZED, OLDEST, NEWEST);

type EVENT_STATUS is (AVAILABLE, NOT_AVAILABLE);

type CONNECTION_CLOSE_REASON is (GRACEFUL, IMMEDIATE);

type CONNECTION_FAIL_REASON is (TIMEOUT,
```

```
        CONNECTION_FAILED_OTHER);

type CONNECTION_ERROR_REASON is (RESET, READ_SIDE_FAIL,
        WRITE_SIDE_FAIL, REFUSED, CONNECTION_ERROR_UNKNOWN,
        CONNECTION_ERROR_OTHER);

type GROUP_RELATED_REASON is (GROUP_CLOSED, GROUP_EMPTY,
        GROUP_FULL, GROUP_UNKNOWN);

type MEMBER_RELATED_REASON is (MEMBER_ADDED, MEMBER_REMOVED,
        CLOSE_PERMISSION_CHANGE,
        OTHER_MEMBER_RELATED_REASON);

type ERROR_RELATED_REASON is (INVALID_GROUP_NAME,
        INVALID_MEMBER_NAME, UNKNOWN_GROUP,
        NO_PERMISSION_TO_JOIN_GROUP, GROUP_ERROR_OTHER);

type OTHER_GROUP_RELATED_REASON is implementation-defined;

type CONNECTION_REASON is (INVALID_SERVICE_PARAMETER,
        UNKNOWN_CONNECTION, INVALID_BUFFER_LENGTH,
        CONNECTION_OTHER);

type CONNECTIONLESS_REASON is (UNKNOWN_NAME,
        INVALID_PROTOCOL_SUITE, CONNECTIONLESS_OTHER);

type ERROR_REASON is (INVALID_NAME, UNICAST_ERROR_OTHER);

type MULTICAST_ERROR_REASON is (MULTICAST_INVALID_BUFFER_LENGTH,
        MULTICAST_INVALID_GROUP_NAME, MULTICAST_UNKNOWN_GROUP,
        MULTICAST_INVALID_PROTOCOL, MULTICAST_ERROR_OTHER);

type RECEPTION_STATE is array (1..2) of boolean;

type EVENT is (CONNECTION_CLOSED,
               CONNECTION_FAIL,
               CONNECTION_ERROR,
               GROUP_GROUP_RELATED,
               GROUP_MEMBER_RELATED,
               GROUP_ERROR_RELATED,
               GROUP_OTHER_RELATED,
               UNICAST_CONNECTION,
               UNICAST_CONNECTIONLESS,
               UNICAST_ERROR,
               MULTICAST_NUMBER_DELIVERY_RELATED,
               MULTICAST_NAMED_DELIVERY_RELATED,
               MULTICAST_ERROR_RELATED,
               MULTICAST_CONNECTION_CLOSED,
               MULTICAST_CONNECTION_FAILED,
               MULTICAST_CONNECTION_ERROR,
               TRANSACTION_ERROR_RELATED,
```

```
                TRANSACTION_OTHER);

type EVENT_DATA (E : EVENT) is
  record
    EVENT_TIME : CALENDAR.clock;
    case E is
      when CONNECTION_CLOSED =>
        CONNECTION1        : LWCM.Connection_ID;
        A1                 : CONNECTION_CLOSE_REASON;
      when CONNECTION_FAIL =>
        CONNECTION2        : LWCM.Connection_ID;
        A2                 : CONNECTION_FAIL_REASON;
      when CONNECTION_ERROR =>
        CONNECTION3        : LWCM.Connection_ID;
        A3                 : CONNECTION_ERROR_REASON;
      when GROUP_GROUP_RELATED =>
        GROUP1             : LWGM.Group_ID;
        ERR1               : GROUP_RELATED_REASON;
      when GROUP_MEMBER_RELATED =>
        GROUP2             : LWGM.Group_ID;
        ERR2               : MEMBER_RELATED_REASON;
      when GROUP_ERROR_RELATED =>
        GROUP3             : LWGM.Group_ID;
        ERR3               : ERROR_RELATED_REASON;
      when GROUP_OTHER_RELATED =>
        GROUP4             : LWGM.Group_ID;
        ERR4               : OTHER_GROUP_RELATED_REASON;
      when UNICAST_CONNECTION =>
        CN_NAME1           : LWCM.Connection_ID;
        U1                 : CONNECTION_REASON;
      when UNICAST_CONNECTIONLESS =>
        CL_NAME2           : LWCS.Logical_Name;
        U2                 : CONNECTIONLESS_REASON;
      when UNICAST_ERROR =>
        ERR5               : ERROR_REASON;
      when MULTICAST_NUMBER_DELIVERY_RELATED =>
        GROUP6             : LWGM.Group_ID;
        NBR_RECEIVED6      : natural;
        NBR_NOT_RECEIVED6  : natural;
      when MULTICAST_NAMED_DELIVERY_RELATED =>
        GROUP7             : LWGM.Group_ID;
        NAME_RECEIVED7     : natural;
        NAME_NOT_RECEIVED7 : positive;
        WHO7               : RECEPTION_STATE;
      when MULTICAST_ERROR_RELATED =>
        GROUP8             : LWGM.Group_ID;
        ERR8               : MULTICAST_ERROR_REASON;
      when MULTICAST_CONNECTION_CLOSED =>
        GROUP9             : LWGM.Group_ID;
        ERR9               : CONNECTION_CLOSE_REASON;
      when MULTICAST_CONNECTION_FAILED =>
```

```
        GROUP10            : LWGM.Group_ID;
        ERR10              : CONNECTION_FAIL_REASON;
      when MULTICAST_CONNECTION_ERROR =>
        GROUP11            : LWGM.Group_ID;
        ERR11              : CONNECTION_ERROR_REASON;
      when TRANSACTION_ERROR_RELATED =>
        implementation-defined
      when TRANSACTION_OTHER =>
        implementation-defined
    end case;
end record;
```

M.7. Alternate Specification 2

The second alternate specification also combines the *Event_Class* type declaration and the different subclass type declarations into the type *Event*. Additionally, all the *reason* type declarations (such as *Connection_Close_Reason*, *Connection_Fail_Reason*) are combined into a single enumerated type named *Reason*. This further simplifies the *Event_Data* record by moving the *Reason* field out of the variant part of the record. The *Event_Data* record is also simplified by combining like alternatives in the variant part.

The type declarations for this alternate specification are

```
type EVENT_CLASS is (CONNECTION, GROUP, UNICAST, MULTICAST,
                     TRANSACTION);

type EVENT_ACTION is (REGISTER, UNREGISTER);

type ORDER is (PRIORITIZED, OLDEST, NEWEST);

type EVENT_STATUS is (AVAILABLE, NOT_AVAILABLE);

type EVENT is (CONNECTION_CLOSED,
               CONNECTION_FAIL,
               CONNECTION_ERROR,
               GROUP_GROUP_RELATED,
               GROUP_MEMBER_RELATED,
               GROUP_ERROR_RELATED,
               GROUP_OTHER_RELATED,
               UNICAST_CONNECTION,
               UNICAST_CONNECTIONLESS,
               UNICAST_ERROR,
               MULTICAST_NUMBER_DELIVERY_RELATED,
               MULTICAST_NAMED_DELIVERY_RELATED,
               MULTICAST_ERROR_RELATED,
               MULTICAST_CONNECTION_CLOSED,
               MULTICAST_CONNECTION_FAILED,
               MULTICAST_CONNECTION_ERROR,
               TRANSACTION_ERROR_RELATED,
```

```
                TRANSACTION_OTHER);

type REASON is (GRACEFUL,
                IMMEDIATE,
                TIMEOUT,
                CONNECTION_FAILED_OTHER,
                RESET,
                READ_SIDE_FAIL,
                WRITE_SIDE_FAIL,
                REFUSED,
                CONNECTION_ERROR_UNKNOWN,
                CONNECTION_ERROR_OTHER,
                GROUP_CLOSED, GROUP_EMPTY,
                GROUP_FULL,
                GROUP_UNKNOWN,
                MEMBER_ADDED,
                MEMBER_REMOVED,
                CLOSE_PERMISSION_CHANGE,
                OTHER_MEMBER_RELATED_REASON,
                INVALID_GROUP_NAME,
                INVALID_MEMBER_NAME,
                UNKNOWN_GROUP,
                NO_PERMISSION_TO_JOIN_GROUP,
                GROUP_ERROR_OTHER,
                INVALID_SERVICE_PARAMETER,
                UNKNOWN_CONNECTION,
                INVALID_BUFFER_LENGTH,
                CONNECTION_OTHER,
                UNKNOWN_NAME,
                INVALID_PROTOCOL_SUITE,
                CONNECTIONLESS_OTHER,
                INVALID_NAME,
                UNICAST_ERROR_OTHER,
                MULTICAST_INVALID_BUFFER_LENGTH,
                MULTICAST_INVALID_GROUP_NAME,
                MULTICAST_UNKNOWN_GROUP,
                MULTICAST_INVALID_PROTOCOL,
                MULTICAST_ERROR_OTHER);

type RECEPTION_STATE is array (1..2) of boolean;


type EVENT_DATA (E : EVENT) is
  record
    EVENT_TIME         : CALENDAR.clock;
    EVENT_REASON       : REASON;
    case E is
      when CONNECTION_CLOSED | CONNECTION_FAIL |
           CONNECTION_ERROR =>
        CONNECTION            : LWCM.Connection_ID;
      when GROUP_GROUP_RELATED | GROUP_MEMBER_RELATED |
```

```
            GROUP_ERROR_RELATED | GROUP_OTHER_RELATED =>
          GROUP                 : LWGM.Group_ID;
          NAME                  : LWCS.Logical_Name;
        when UNICAST_CONNECTION | UNICAST_CONNECTIONLESS |
             UNICAST_ERROR =>
          UNICAST_CONNECTION_ID : LWCM.Connection_ID;
          UNICAST_NAME          : LWCS.Logical_Name;
        when MULTICAST_NUMBER_DELIVERY_RELATED |
             MULTICAST_NAMED_DELIVERY_RELATED |
             MULTICAST_ERROR_RELATED =>
          MULTICAST_GROUP       : LWGM.Group_ID;
          MULTICAST_NAME        : LWCS.Logical_Name;
          NBR_RECEIVED          : natural;
          NBR_NOT_RECEIVED      : natural;
          NAME_RECEIVED         : natural;
          NAME_NOT_RECEIVED     : positive;
          WHO                   : RECEPTION_STATE;
        when TRANSACTION_ERROR_RELATED =>
          implementation-defined
        when TRANSACTION_OTHER =>
          implementation-defined
      end case;
  end record;
```

M.8. Alternate Specification 3

The final alternate specification combines the original type declaration *Event_Class*, the *subclass* type declarations, and the *reason* type declarations into one type, *Event*. This further simplifies the *Event_Data* record, as the reason associated with an event is now included in the record's discriminant. Also, notice that the elements in the enumerated type *Event* are ordered to allow the use of ranges in the alternatives of the *Event_Data* record.

The type declarations for this alternate specification are

```
type EVENT_CLASS is (CONNECTION, GROUP, UNICAST, MULTICAST,
                     TRANSACTION);

type EVENT_ACTION is (REGISTER, UNREGISTER);

type ORDER is (PRIORITIZED, OLDEST, NEWEST);

type EVENT_STATUS is (AVAILABLE, NOT_AVAILABLE);

type RECEPTION_STATE is array (1..2) of boolean;

type EVENT is (CONNECTION_CLOSED_GRACEFUL,
               CONNECTION_CLOSED_IMMEDIATE,
               CONNECTION_FAILED_TIMEOUT,
               CONNECTION_FAILED_OTHER,
               CONNECTION_ERROR_RESET,
```

```
                CONNECTION_ERROR_READ_SIDE_FAILED,
                CONNECTION_ERROR_WRITE_SIDE_FAILED,
                CONNECTION_ERROR_REFUSED,
                CONNECTION_ERROR_UNKNOWN,
                CONNECTION_ERROR_OTHER,
                GROUP_GROUP_CLOSED,
                GROUP_GROUP_EMPTY,
                GROUP_GROUP_FULL,
                GROUP_GROUP_UNKNOWN,
                GROUP_MEMBER_MEMBER_ADDED,
                GROUP_MEMBER_MEMBER_REMOVED,
                GROUP_MEMBER_CLOSE_PERMISSION_CHANGE,
                GROUP_MEMBER_OTHER_MEMBER_RELATED,
                GROUP_ERROR_INVALID_GROUP_NAME,
                GROUP_ERROR_INVALID_MEMBER_NAME,
                GROUP_ERROR_UNKNOWN_GROUP,
                GROUP_ERROR_NO_PERMISSION_TO_JOIN_GROUP,
                GROUP_ERROR_OTHER, GROUP_OTHER_RELATED,
                UNICAST_CONNECTION_INVALID_SERVICE_PARAM,
                UNICAST_CONNECTION_UNKNOWN_CONNECTION,
                UNICAST_CONNECTION_INVALID_BUFFER_LENGTH,
                UNICAST_CONNECTION_OTHER,
                UNICAST_CONNECTIONLESS_UNKNOWN_NAME,
                UNICAST_CONNECTIONLESS_INVALID_PROTOCOL_SUITE,
                UNICAST_CONNECTIONLESS_OTHER,
                UNICAST_ERROR_INVALID_NAME,
                UNICAST_ERROR_OTHER,
                MULTICAST_NUMBER_DELIVERY,
                MULTICAST_NAMED_DELIVERY,
                MULTICAST_ERROR_INVALID_BUFFER_LENGTH,
                MULTICAST_ERROR_INVALID_GROUP_NAME,
                MULTICAST_ERROR_UNKNOWN_GROUP,
                MULTICAST_ERROR_INVALID_PROTOCOL,
                MULTICAST_ERROR_OTHER,
                MULTICAST_CONNECTION_CLOSED,
                MULTICAST_CONNECTION_FAILED,
                MULTICAST_CONNECTION_ERROR,
                TRANSACTION_ERROR_RELATED,
                TRANSACTION_OTHER);

  type EVENT_DATA (E : EVENT) is
    record
      EVENT_TIME : CALENDAR.clock;
      case E is
        when CONNECTION_CLOSED_GRACEFUL .. CONNECTION_ERROR_OTHER =>
          CONNECTION            : LWCM.Connection_ID;
        when GROUP_GROUP_CLOSED ..  GROUP_OTHER_RELATED =>
          GROUP                 : LWGM.Group_ID;
          NAME                  : LWCS.Logical_Name;
        when UNICAST_CONNECTION_INVALID_SERVICE_PARAM
             .. UNICAST_ERROR_OTHER =>
```

```
        UNICAST_CONNECTION_ID : LWCM.Connection_ID;
        UNICAST_NAME          : LWCS.Logical_Name;
      when MULTICAST_NUMBER_DELIVERY .. TRANSACTION_ERROR_OTHER =>
        implementation-defined
    end case;
end record;
```

Appendix N: Differences from Previous Ada Binding

The SAFENET handbook, reference [NGCR92b] has been changed. The resulting changes to the previous Ada binding[41] are listed by Ada package.

- The changes to the package *LW_Address_Management* are:
  - The function *Address_Binding_Exists* was added.
  - The procedures *Bind* and *Unbind* replace the previous procedures *Bind_Address* and *Unbind_Address*.
- The changes to the package *LW_Communication_Support* are:
  - The new state *Failed* was added to type *Activity_State*.
  - The type *Terminate_Mode* was moved to here from the package *LW_Connection_Management*.
  - The type *Message_Class* was added.
  - The procedure *Wait_On_Asynchronous_Activity* was added.
  - A parameter of type *Terminate_Mode* was added to procedure *Terminate_All_Transfers*.
- The changes to the package *LW_Connection_Management* are:
  - Definitions for types *Connection_Block* and *Connection_Block_ID* were added.
  - Procedures were added for use with types *Connection_Block* and *Connection_Block_ID*.
  - Procedure *Open_Connection_With_Data* was added that allows an application to specify a message class.
  - The type definition *Flow_Control* was moved to here from package *LW_Communications_Support*.
  - Procedures to get and set the *Flow_Control* field of an *Activity_Block_ID* were added.
- The changes to the package *LW_Group_Management* are:
  - A parameter of type *Connection_Block_ID* was added to procedure *Open_Group*.
  - A parameter *Member* was added to procedures *Join_Group* and *Leave_Group*.
  - A parameter *Notify* was added to procedure *Open_Group*.
- The changes to the package *LW_Data_Transfer* are:

[41] The original version of the binding is available via anonymous FTP at site ftp.sei.cmu.edu, in the directory pub/posix/docs.

- Most procedures and functions in package *LW_Data_Transfer* were overloaded to include a parameter of type *Message_Class*.
- The procedure *Get_Any_Connectionless_Message* was added.
- The procedure *Get_Message_For_Any_Group* was added.
- The procedure *Purge_All_Broadcast_Messages* was added.
- The parameter *Data* in procedure *Get_Broadcast_Message* was changed to be an out parameter.
- The *Before* parameters in package *LW_Data_Transfer* were changed from type *duration* to type *Calendar.time*.
- The *Send_Connection_Message* routine changed so that the *Data* parameter is of type *Connection_Data*.
- The routine *Send_Multicast_Stream_Message* was added.
- The routine *Send_Multicast_Stream_Buffer* was added.

- The changes to the package *LW_Transaction_Services* are:
  - Procedure *Initiate_Unicast_Transaction* was overloaded to include an *in* parameter of type *Message_Class*.
  - Procedure *Accept_Transaction_Request* was overloaded to include an *in* parameter of type *Message_Class*.
  - Procedure *Accept_Any_Transaction_Request* was overloaded to include an *in* parameter of type *Message_Class*.
  - Procedure *Accept_Any_Transaction_Request* was overloaded to include an *out* parameter of type *Message_Class*.
  - Function *Number_of_Transactions* was overloaded to include an *in* parameter of type *Message_Class*.
  - Function *Cancel_All_Transactions* was overloaded to include an *in* parameter of type *Message_Class*.

- The changes to the package *LW_Event_Management* are:
  - A *Request_Event* routine that allows applications to register and deregister for all events was added.
  - An *Get_Event_Class* routine that returns event class was added.
  - The *Event_Time* component of the event data records was changed from type *duration* to type *Calendar.time*.
  - Several of the names in the *reason* enumerated types were changed to make each reason's name unique within the package.
  - The types *Transaction_Subclass* and *Transaction_Event_Data* were changed because SAFENET dropped the multicast transaction.
  - The *class* parameter was eliminated from the routines *Request_Event*, *Event_Present*, and *Number_of_Events_Present* routines (except for the first occurrence of each of those routines).
  - The routines that get the event data were renamed to *Get_Event_Data*.

Appendix O: Acronyms

The following is the list of the acronyms used in this report:

ANSI	American National Standards Institute
API	application program interface
ASN.1	Abstract Syntax Notation One
DMA	direct memory access
FDDI	Fiber Distributed Data Interface
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
LLC	logical link control
MAC	media access control
MAP	manufacturing automation protocol
OSI	open systems interconnection
NSAP	network service access point
NGCR	Navy Next Generation Computer Resources Program
NIST	National Institute of Standards and Technology
PHY	FDDI physical layer protocol
PMD	FDDI physical layer, media dependent
POSIX	portable operating system interface
QOS	quality of service
SAFENET	Survivable Adaptable Fiber Optic Embedded Network
SEI	Software Engineering Institute
SLA	SAFENET lightweight application
SMT	station management
TCP	transmission control protocol
TP4	transport protocol, class 4
XTP	express transfer protocol

Index