OPEN C Platform

Neda Document Number: 103-100-14

Revision: source unspecified

Last Updated: Author unspecified

Tag: CVS tag unspecified

Mohsen Banan
Neda Communications, Inc.
17005 SE 31st Place
Bellevue, WA 98008

July 13, 1998

This document describes the Open C Platform Library, a utility which aids in the consistency of portable implementation of data communications software.

Published by Neda Communications, Inc.
3610 164th Place SE
Bellevue, WA 98008 USA

Contents

List of Tables

List of Figures

Foreword

Realization of global connectivity through the concept of Open Systems Interconnection (OSI) will be the reward of a significant collective effort. The development of OSI over the past ten years has involved the collective efforts of many experts from many countries.

A great deal of the development of OSI is in its final stages. Fruits of these efforts are beginning to appear in our daily lives as implementation of OSI protocols. The future of OSI is very exciting and its impact on our lives will be significant.

This book focuses on creating an environment (a model, and architecture and a platform) for software implementation of OSI protocols. It is assumed that the reader is familiar with the concepts of OSI Reference Model. This book establishes a reference model for portable software implementation of OSI in C.

Although the primary intent of this book is to establish a platform for implementation of International Standards Organization (ISO) specification of OSI protocols, a great deal of the contents of this book applies to layered software implementation in general.

The C programming language has proven itself as a popular portable and efficient language for implementation of data communications software. The implementation environment described in this book is centered around C. It is assumed that the reader has a working knowledge of C.

Software implementation of OSI protocols that adhere to the ”Open C Environment” (OCE) described in this book can be integrated to realize efficient real open systems. The environment presented in this book has been used for implementation of a number of ISO specification of OSI protocols. Any C compiler that conforms to the ANSI C specification can be used to port this implementation platform to the target environment.

It is the aim of this book to refine the abstract descriptions of the OSI model as they apply to software implementation in C. An implementation environment that provides for efficient integration of OSI protocols is created. Specification of OSI protocols, C language and the environment described in this book provide for portable and efficient realization of real open systems.

Implementors, system architects, system programmers, application programmers and anyone who is interested in understanding the implementation aspects of OSI can benefit from reading this book. This book can be used to teach the practical aspects of data communications software development in conjunction with other books that deal with the concepts of Open Systems Interconnections.

The structure of this book is as follows. Chapter 1 provides an introduction to the issues involved in implementation of OSI. Chapter 2 describes the Open C Environment. Chapter 3 outlines a number of conventions followed in defining the Open C Platform. Chapter 4 outlines the common features of the implementation layers. Chapter 5 deals with the architecture of network management. Chapter 6 describes the interface and service definition for Open C Platform. Chapter 7 provides an example usage of Open C Platform. Chapter 8 deals with implementation issues. Appendix A contains a set of UNIX style manual pages for facilities of Open C Platform.

Acknowledgement

Many of the implementation techniques outlined in this book are very similar to the ones practiced by NBS [1], Retix and a number of other implementors. Little, if any, of the credit for the implementation techniques should go to the author. My goal has been to combine these commonly practiced good techniques into an environment that may facilitate the realization of OSI.

I wish to thank my colleagues at Retix who inspired the ideas behind many of the concepts appearing in this book.

A number of individuals at several companies have contributed to the creation and expansion of sections of this manual as well as the software itself. Neda Communications, Inc. leads the effort of maintaining and documenting the majority of the software.

Significant contributions to this work resulted from the use of OCP by AT&T Wireless Services.

Finally, a number of individuals who have made contributions to the OCP library and this document deserve special recognition.

- Derrell Lipman - Visual Aviation derrell@vis-av.com
- Sanjay Bapat - PCSI, Inc. SPB@pcsi.cirrus.com
- Mohsen Banan - Neda Communications, Inc. mohsen@neda.com
- Steve Farowich - Neda Communications, Inc. steve@neda.com
- Kamran Ghane - Neda Communications, Inc. kamran@rostam.neda.com
- Fletch Holmquist - Neda Communications, Inc. fletch@neda.com
- Hugh Shane - Neda Communications, Inc. hshane@neda.com

We look forward to continued growth of the OCP library. Please send your feedback, including code contributions, to

Mohsen Banan - Neda Communications, Inc. mohsen@neda.com

About This Document

This manual is written in LaTeXinfo. LaTeXinfo is a documentation system that uses a single source file for both on–line documentation and a printed manual. See the LaTeXinfo Manual for more details [**?**].

The on–line documentation is in the form of an Info file. An on-line Info file is a file formatted so that the Info documentation reading program can operate on it. Info files are divided into pieces called *nodes*, each of which contains the discussion of one topic. M-x info in emacs and xinfo under X11 are two Info documentation reading programs that can be used for on-line manipulation of this manual.

Release Information

Module openCPlatform
Was released as Revision: on by The name tag for this release is The file capturing this release is

[1] now NIST.

Caveats

This manual is mostly complete and has gone through a few drafts. However, it is far from being flawless.

This document should be fully correct in what it does say; and it is therefore open to criticism on anything it does include — from specific examples and descriptive text, to the ordering of chapters and sections. If something is confusing or incorrect, then perhaps the document should be fixed.

As you use this document, please mark pages with corrections so you can later look them up and send them in. Please reference any comments to the chapter name and section name, since page numbers and chapter and section numbers will change.

Comments

Comments concerning this document should be addressed to:


```
E-Mail:                 mohsen@neda.com

Postal Address:
                        Mohsen Banan
                        Neda Communications, Inc.
                        17005 SE 31st Place
                        Bellevue, WA 98008

Fax:                    (425) 562-9591
```

Chapter 1

INTRODUCTION

International Standards Organization (ISO) specification of OSI model and protocols are based on a number of abstract descriptions which specify essential requirements but make no mention of how these requirements should be met by an implementation.

Typically each OSI layer has two sets of specifications: One for the services provided by the protocol and one for the protocol. Service specifications for each layer are normally described in a high level of abstraction. The interfaces for the service definition is left at the level of primitive. Furthermore, the ISO specifications do not deal with any details concerning the mechanism used to exchange primitives across a layer interface.

Lack of a standard interface or a standard mechanism for exchange of primitives across a layer interface can cause severe problems in implementation of machine independent OSI software.

OSI Reference Model offers a number of features that are well aligned with today's software implementation methodology. To name a few:

- OSI model subdivides the communications functions into logically separate modules.
- Exclusive peer-to-peer interaction.
- A well defined service definition expected from each layer.
- Independence of protocol specification from the services expected from the protocol.

A well defined model, architecture and interface provides for integration of independent implementations of OSI protocols. It is the aim of this book to refine the abstract descriptions of the OSI model and create a software implementation environment (A model, an architecture and a platform) that facilitates the implementation of OSI.

Given today's state of technology a number of observations are made.

- The specification of the bottom five layers of OSI have been adopted as ISO standards and are well understood.
- C language has been recognized as a standard for portable and efficient implementation of data communication software.
- A correct, efficient and environment independent implementation of OSI protocols is very desirable.

The author therefore feels that the time is right for creation of a software environment that can facilitate the realization of OSI. Such a platform should be:

- Operating Environment (Operating System/ CPU/ Hardware) independent.
- Efficient.
- Consistent.
- Expandable.

This has been the motivation behind the creation of ”Open C Environment” (OCE) for implementation of OSI. OCE is a set of architectural guide lines, conventions and a platform for portable implementation of OSI software. Independent implementations of OSI protocols that adhere to OCE can easily be integrated to realize real open systems.

A platform of facilities that can ease the implementation of OSI protocols is provided. One of the primary goals of this book is to formally define the interface to this platform of facilities.

1.1 Lower Layers

The nature of the different services expected of the different layers results into particular services expected from the implementation environment. The lower layers (Transport, Network, Data Link and Physical) are responsible for transport of unstructured data. The upper layers (Application, Presentation and Session) deal with the structure, syntax and semantics of the communication.

The upper layers expect more services form their implementation environment than the lower layers. The implementation environment expected by the upper layers can be considered as an extension of the lower layers implementation environment.

It is recognized that at this time only the bottom five layers have matured. This book's primary focus is to create an implementation environment for the lower layers.

1.2 Target Environment

One of the goals for the creation of Open C Environment for implementation of OSI is portability and environment independence. Two basic categories of target environments are identified.

- Hosted
- Un-Hosted

The Hosted-Environment is an environment in which the existance of an operating system and multitasking capabilities may be assumed. The availability of facilities such as scheduling, timers, synchronization and dynamic memory allocation are often available in Hosted-Environments. UNIX, VMS, MS-DOS and VRTX are examples of hosted environments.

The Unhosted-Environment is an environment in which the existance of no facilities may be assumed. The underlying environment should at least provide:

- CPU
- Adequate amount of memory
- Periodic interrupt

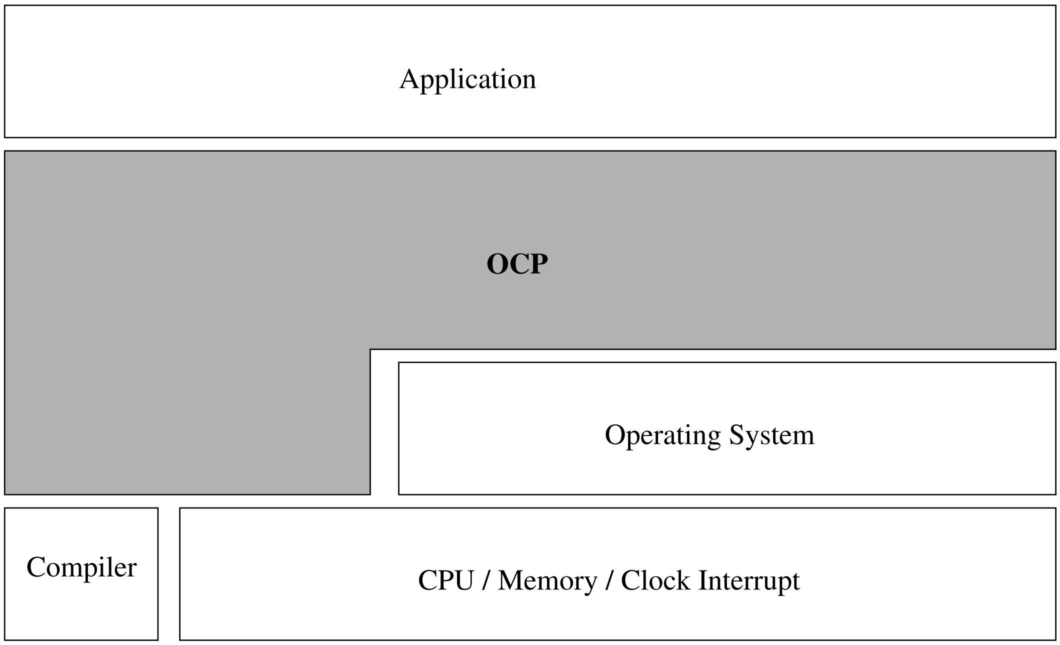

Figure 1.1: Open C Environment

Bare, intelligent, front-end processors are examples of Unhosted-Environments.

Open C Environment is designed to exist in Hosted and Unhosted environments. In Hosted-Environments, the interface to host facilities should be mapped on to the Open C Platform interface definitions. Design of OCE does consider well-behaved existance in Hosted-Environments.

The type of environment specific facilities defined in OCE are basic and simple. These facilities can easily be implemented from scratch in Un-Hosted environments.

1.3 Development Environment

A C compiler, an assembler, a linker, and a loader are expected of a development environment necessary for use of OCE.

The primary development environment used by the author is UNIX.

1.4 The Open C Environment

This section defines the relationship between OCP and the various components in it's Environment. In the following discussion refer to the diagram in Figure 1.1

The diagram as a whole represents a particular implementation of an Open C Layer. The Application, represented as the highest level component, is the body of software that is unique to a given application or service. The CPU/Memory combination, at the lowest level, represents the minimum computing functionality that OCP requires.

The Operating System, which may or may not exist on a given computing platform, provides some set of services to OCP. Finally, a particular C Compiler is used to generate machine executable code.

For example, in the case of a Unix workstation, the CPU might be a 64-bit processor with gigabytes of virtual memory. The operating system would provide a wide range of services including disk file I/O, interprocess communication, and network services.

On the other hand, a portable messaging device might use a 16-bit processor with 64 kilobytes of memory while the operating system might be nothing more than a simple task scheduler.

OCP isolates the Application from all of this underlying complexity and variability. Once OCP is ported to a given Open C Environment (CPU/Memory, Operating System, and Compiler) any Application developed under one Open C Environment can easily be ported to another.

1.5 Definition of Terms

Definitions, notation, abbreviation and terminology used in this book are consistent with the terminology and principles established by ISO for Open Systems interconnection.

The OSI reference model is based on a number of 'abstract' descriptions. In our implementation model, these abstract descriptions are refined to define the precise context they take as they apply to this implementation architecture.

Facility To evade the question of whether an operation is implemented as a function or a preprocessor macro, the word facility is used. What is of interest is the facility's interface description. Most of the facilities are described as if they were functions. But when efficiency justifies the equivalent may be implemented as macro. With this understanding, 'facility' and 'function' are used interchangeably.

Module Typically a set of related facilities and data abstractions are combined into a module. The facility is typically realized by a link module (e.g. a library) and a declaration module. The correct way to use a facility is to have, at the beginning of the user program, a preprocessor #include command to include the relevant facility declarations.

Open C Environment OCE is a set of architectural guide lines, conventions and a well defined platform interface that can ease portable implementation of OSI software.

Open C Platform OCP is a collection of modules that provide the basic facilities expected by lower layers. The interface to these facilities is described in this book.

Open C Layer An OCL is an implementation of an OSI layer that adheres to the architecture of Open C Environment.

Service Provider A module responsible for providing some well defined set of services through a well defined set of interfaces.

Service User A module that uses the services of a service provider.

Primitive A facility expressing an interaction between a service user and a service provider.

Parameter A facility argument to a primitive.

Primitive Action A facility invocation into the service provider module. Primitive actions are provided in the service provider module and invoked by the service user. Primitive action is the collective name for requests and responses.

Primitive Event A facility invocation outside of the service provider module. Typically in the service user module. Primitive events are provided in the service user module and invoked by the service provider. Primitive event is the collective name for indications and confirmations. The set of primitive events to be used are communicated to the service provider during the creation of the service access point.

Request A primitive issued by the service user to express a request for a service from the service provider.

Indication A primitive issued by the service provider to indicate that a request has been issued by the service user at the peer service access point or to communicate a local event to the service user.

Response A primitive issued by the service user in response to some indication previously invoked by the service provider.

Confirm A primitive issued by the service provider in response to some response previously issued by the service user.

Service Access Point A service access point (SAP) is defined as the interface between a service user and a service provider. Each SAP is identified by the service provider through a SAP-Address-Selector.

Connection End Point An (N) connection is an association established by the (N) layer between two or more (N+1) entities for the transfer of data. An (N) connection end point (CEP) is a terminator at one end of an (N) connection within an (N)-SAP. An (N) connection end point identifier (CEP-ID) is a unique representation of a CEP within the scope of the (N) service

sequence-of Borrowed from ASN.1. sequence-of type: A structured type, defined by referencing a single existing type; each value in the new type is an ordered list of zero, one or more values of existing type.

set-of Borrowed from ASN.1. set-of type: A structured type, defined by referencing a single existing type; each value in the new type is an unordered list of zero, one or more values of the existing type.

Chapter 2

OPEN C ENVIRONMENT

Open C Environment is a set of architectural guidelines, conventions, and a common platform that allow for efficient implementation and integration of OSI protocols in C. Open C Environment defines the architecture and the mechanism used for exchange of primitives across layer interfaces. Open C Platform defines the interfaces to a set of common facilities that may be used by Open C layers.

An Open C Layer is the implementation of an OSI protocol that conforms to Open C Environment architecture and guidelines. Direct function calls are used for exchange of primitives across Open C layers. This simple and efficient primitive exchange mechanism allows for integration of Open C Layers into multi-layered communication entities. Other methods for exchange of service primitives (such as interprocess or interprocessor communication facilities) often result into performance degradation across layer boundaries.

Once the upper and the lower interfaces of a Layer are defined (based on the service definitions), independent implementors can implement Open C Layers. Since the upper interface of implementation of (N) layer protocols match the lower interface of implementation of (N+1) layer protocols, several Open C Layers can be integrated into a multi-layered communication software entity.

2.1 Open C Layer Architecture

An Open C Layer is a cohesive piece of software that provides a well defined service and has a set of well defined interfaces. An OSI Open C Layer implements an OSI protocol. Open C Layers have a consistent design architecture. Each implementation of an OSI protocol, provides to its user the set of services defined for that protocol at its upper interface. It assumes the services of the layer below it to be present at its lower interface.

A typical software layer (N) appears to its user as a link module with four defined interfaces. The figure below (Figure 2.1) presents an interface model for the (N) software module.

2.1.1 Upper Interface

The (N) Open C Layer is expected to provide the services defined for the (N) layer at its upper interface. The (N) Open C Layer's upper interface is a series of function calls (primitives). Each function call accepts a group of arguments (parameters). Each function call is non-blocking.

(N) request and responses, collectively referred to as (N) action primitives, are function calls into the (N) Open C Layer. (N) action primitives are invoked by the (N) service user. The code for (N) action primitives is inside the (N)

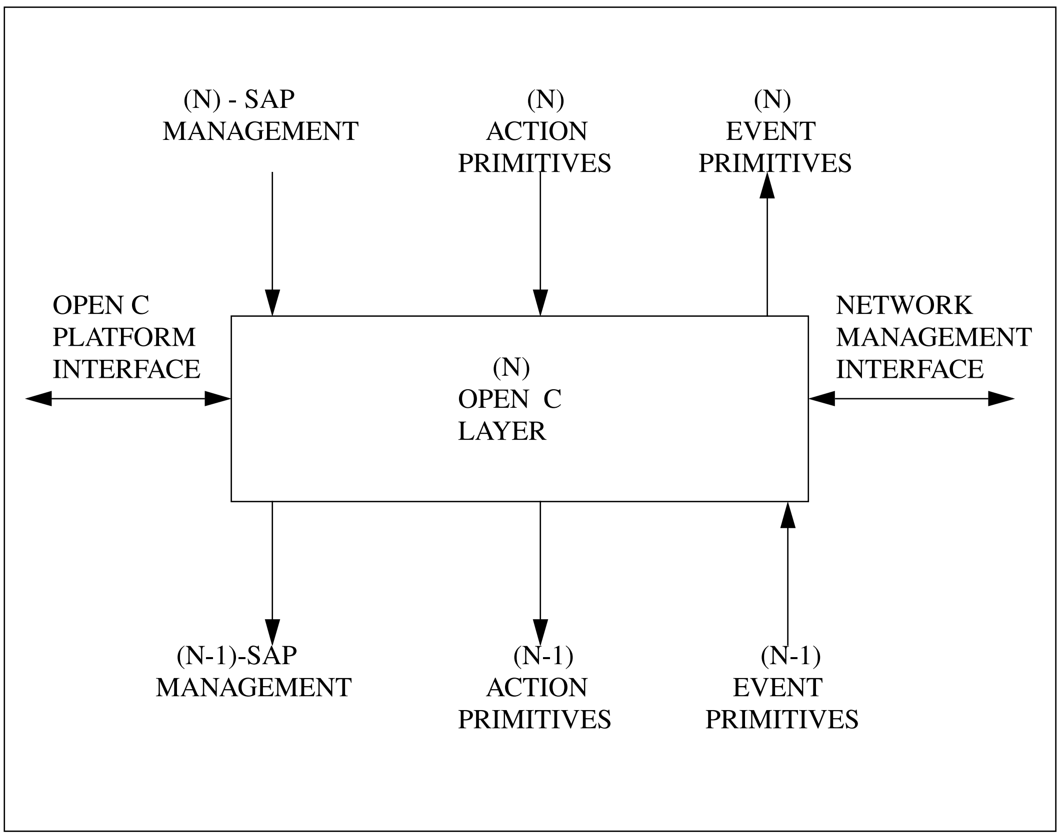

Figure 2.1: Architecture of a Layer

Open C Layer.

(N) indication and confirmations, collectively referred to as (N) event primitives, are function calls outside of the (N) software module. (N) event primitives are invoked by the (N) software module. The code for (N) event primitives is expected to be provided by the (N) service user. The entry point for (N) event primitives is conveyed to the (N) layer during the creation of the service access point by (N) service user.

2.1.2 Lower Interface

The (N) Open C Layer may use the services defined for the (N-1) layer at its lower interface. The lower interface of the (N) software module matches the upper interface of (N-1) software module.

2.1.3 Network Management Interface

Operation of an OSI layer may be monitored and manipulated by the Network Management Administrator. To provide for this each Open C Layer is responsible for providing an interface that allows for network management administration.

Within each Open C Layer, often a Layer Management Entity (LME) is responsible for defining and providing the network administration interface.

2.1.4 Open C Platform Interface

An Open C Layer can rely on the services provided by the Open C Platform. The interface and the description of the services provided by the Open C Platform is described in this book.

2.1.5 Significant Events

The following is a list of significant events that result into execution of code inside of (N) Open C layer.

- (N) Initialization.
- (N) SAP Management.
- (N) Action Primitives.
- (N-1) Event Primitives.
- Timer Expiration Events (See TMR_ module).
- (N) Re-Scheduling Request.

Chapter 3

CONVENTIONS

A set of programming conventions and restrictions are followed to improve portability and readability. These conventions are visible to the software modules that interface with Open C Platform. For sake of consistency it is suggested that the following simple conventions be adapted when interfacing with other elements of Open C Environment.

Good programming conventions can result in improvements in consistency, portability, readability and writability. The development environment and the nature of the software developed have a large influence on the styles and conventions that are adapted. As the development environment and software engineering methodology evolve, the stylistic notations and conventions will also evolve.

The identifier naming conventions described here are consistent with ISO's conventions in specification of ASN1 definitions. These naming conventions are also consistent with the conventions followed in the C bindings produced by MAP and TOP groups. There are some differences but these are minor.

3.1 Identifier Naming

The identifier naming conventions include:

- Variables, functions, parameters and struct fields beginning with lower-case, then upper and lower-case mixed. N_sapCreate and sapAddr are examples of function and parameter identifiers in the following example. "N_" in "N_sapCreate" is a module prefix and is not subject to this convention.

```
N_SapDesc
N_sapCreate(sapAddr)
N_SapAddr *sapAddr;
```

- Typedefs, struct/Union/enum tags named with upper-case groups. Beginning with upper case, then lower and upper-case mixed:

```
String, struct N_SapAddr
```

- #define and enumerated constants named with upper-case letters:

```
FALSE, MAXLENGTH
```

- Identifiers exported by a module prefix followed by an '_'. The module prefix can be 1 to 4 letters long.

```
QU_init, TMR_Desc, N_SAPLENGTH
```

The layout of expressions and statements follow those in the book written by Kernighan and Ritchie entitled "The C Programming Language".

The few simple conventions mentioned above are adequate for describing the basic guide lines. The following section describe in more detail the naming conventions followed and the rationale behind it.

3.2 Identifier Naming Details

The naming conventions recommended in this section are specific to the C programming language. These conventions take full advantage of the identifier naming facilities offered by the standard definition of C language. Lower and upper case letters as well as '_' are used in this naming convention. All identifiers are expected to be unique with in the first 24 characters.

Concept of a module and hiding in C is limited to the source file. The scope of an identifier outside of a source file is global. Explicit importing and exporting of identifiers is not supported in standard C. This naming convention tries to address this known deficiency in C.

All identifiers are composed of two elements: module prefix and qualifier. Each module is identified by a module prefix. A module prefix is a short name (normally 1 to 4 characters long) followed by an '_'. The case of the module prefix specifies the scope of the identifier with regard to that module. Identifiers exported by a module, have an all upper-case module prefix. An all lower case module prefix signifies that although the identifier is needed by more than one source file within the implementation of the module, it is purely private to the module and need not be exposed to any users of the module. Purely local identifiers need not have the module prefix component and consist of a pure qualifier. In addition to conveying the scope of the identifier, this convention results in prevention of naming collisions across independent modules.

The qualifier component of an identifier mainly conveys the semantic attributes of the identifier. The qualifier component may consist of many words. With the exception of the first word, all following words start with upper-case. The first character of the qualifier conveys some type information. Variables, functions, parameters and structure fields begin with lower-case, the then upper and lower-case mixed. Typedefs, struct/union/enum tags named with upper-case groups. Although not clearly specified in the language definition, typedefs name space is different from struct/union/enum tags name space. Name space overloading is encouraged.

This convention results into clear usage of the same name for a number of related identifiers. This is demonstrated in the following example.

```
typedef struct SapSelector {
        Int len;
        Byte addr[NSAPSZ];
} SapSelector;

SapSelector sapSelector;
```

Same descriptive name "sapSelector" is used for the structure tag identifier, the type definition identifier and the variable identifier for a generic instance.

3.2.1 Naming GuideLines

A set of recommendations are proposed for qualifier naming.

Grouping/Classing A concept is often expressed through a set of related identifiers. Within a module, related identifiers can be grouped in a variety of ways. Grouping with regard to the data class often works fine. Take the case of SAP management in "N_" module, N_sapCreate, N_sapDelete, n_SapInfo, N_SapDesc would be considered natural choices.

Procedures A verb followed by noun is often a good choice.

Labels Same as Procedures.

Booleans A phrase expressing the TRUE case of the variable.

3.2.2 Abbreviations

A number of abbreviations are commonly used to express well defined concepts with in the scope of OSI implementation. These abbreviations are commonly used for identifier naming.

Req A request primitive.

Rsp A response primitive.

Ind An indication primitive.

Cnf A confirmation primitive.

Sap Service Access Point.

Cep Connection End Point.

Du Data Unit.

Pdu Protocol Data Unit.

Sdu Service Data Unit.

Pci Protocol Control Information.

Buf Buffer.

Addr Address.

Sel Selector.

Src Source.

Dst Destination.

Loc Local. Matches with Rem.

Rem Remote. Matches with Loc.

Seq Sequence. Head of a linked list of ordered elements.

Set (Noun) Head of a linked list of unordered elements.

Elem Element. An element in a sequence, queue, set.

Next Matches with prev. A valid value.

Prev Previous. Matches with next. A valid value.

Last Matches with first. A valid value.

First Matches with last. A valid value.

Lim Limit. Not a valid value.

Min Minimum. A valid value. Compile or link time constant.

Max Maximum. Not a valid value. Compile or link time constant.

Info Information. Often used to provide an internal representation of a resource.

Desc Descriptor. Often used to provide a public handle to a resource.

Ref Reference. Often used as a prefix to provide a handle to a resource.

Prov Provider. To identify roles in a service provider/user situation.

User To identify roles in a service provider/user situation.

Cur Current.

Val Value.

Init Initialize.

Term Terminate.

Get Retrieve an information object from an information base.

Set (Verb) Set an information object in an information base.

3.2.3 Rationale

The primary intentions of the identifier naming convention mentioned above is to convey important information about identifiers while keeping the names short and natural.

The following is a partial list of the intentions of this naming convention.

- Maximize conveying important identifier attributes while keeping the names short and natural.
- Emphasis on semantic information. An identifier's primary responsibility is to convey the natural aspect of what it represents.

- Emphasize module definition and highlight scope recognition. Definition of C language is limited in its ability with regard to modularity and hiding to one source files. This naming convention extends the module and scope concepts beyond one source module.
- It is recognized that complete type information about all identifiers is accessible through their definition and context. Proper tools may be used to perform syntax type checking improve reliability. Some information regarding the type of the identifier is conveyed in this naming convention. The conventions with regard to type allow for name space over loading and minimize name selection.

3.3 Source File Names

Recognizing the restrictions that may be imposed by the development environments that may be used, the following conventions are recommended.

- Base of the file name is limited to 8 characters. The extension of a file name is limited to 3 characters.
- Only lower case letters and '_' may be used in file name specification.
- C header files have a ".h" extension.
- C source files have a ".c" extension.

With these restrictions even MS-DOS development environment can be supported.

3.4 Implementation of Sequences and Sets

The concepts of ASN.1 sequences-of and sets-of are not inherent in C. Arrays are adequate for static usage. Dynamic usage is often implemented through the usage of linked lists. For implementation purposes set-of and sequence-of are considered the same. Relevance of order is the user's view and not an implementation issue. Only the implementation of sequence-of is described here.

ASN.1 sequence concept maps to struct in C. With this understanding, for design purposes a shorthand notation is adapted to make the concept of sequence-of more natural to C. The following keyword extension are made:

```
sequence
sequenceof
```

Their usage is similar to struct and union. sequence specifies an element of a list. sequenceof specifies the head of a list. These are based on the QU_ and SEQ_ module. sequence and sequenceof can be used for data or instance declaration.

Consider the following example:

```
typedef sequence SomeInfo {
        Int someField;
} SomeInfo;
```

Which is equivalent to:

```
typedef struct SomeInfo {
        struct SomeInfo *next;
        struct SomeInfo *prev;
        Int someField;
} SomeInfo;
```

For a sequenceof instance declaration, consider:

```
sequenceof SomeInfo someInfoSeq;
```

Which is equivalent to:

```
struct {
        SomeInfo *first;
        SomeInfo *last;
} someInfoSeq;
```

For a sequenceof type declaration, consider:

```
typedef sequenceof SomeInfo {
        Int nuOfElems;
} SomeInfoSeq;
SomeInfoSeq someInfoSeq;
```

Which is equivalent to:

```
typedef struct {
        SomeInfo *first;
        SomeInfo *last;
        Int nuOfElems;
} SomeInfoSeq;
SomeInfoSeq someInfoSeq;
```

Implementation of sets-of can be done as sequences-of. The user may convey the irrelevance of order through the usage of:

```
set
setof
```

reserved word extensions.

3.5 Restrictions on Usage of Features in C

Open C Environment conforms to the ANSI C standard. At present, a large number of existing C compilers have not implemented all the new features in ANSI C. New features of C, (function prototypes, structure assignment and new reserved words) may be used only if backwards compatibility is maintained. This can often be accomplished through conditional compilation.

Pre-processor identifier *ANSIC* is reserved for this purpose. Only environments that conform to the standard should have it defined. The following example demonstrates the proper usage of the new features.

```
#ifdef ANSIC
typedef void * Ptr;
#else
typedef unsigned char * Ptr;
#endif

#ifdef ANSIC
SuccFail N_sapCreate(Int sapSel);
#else
SuccFail N_sapCreate();
#endif
```

Structure assignment and use of structures as function arguments is also discouraged unless compatibility with older compilers is maintained.

3.6 Compile Time Configuration

Environment specific facilities may be implemented in a variety of ways. To support more than one target environment through the same source code, conditional compilation features are often used. Three basic elements of the operating environment are recognized as:

- The operating system.
- The CPU type.
- The C compiler.

Conventional compile time identifiers that identify the operating environment are placed in a file called ”oe.h”. When porting the Open C Platform ”oe.h” must be properly configured to reflect the target environment.

When developing on a PC-AT running XENIX, my oe.h is configured as:

```
#ifndef _OE_H_ /*{*/
#define _OE_H_

/*
 * CPU.
 * Used for byte order representation of integers.
 * Supported CPUs are:
 *      INTEL for 8086 family (8088, 80186, 80188, 80286)
 *      VAX    for DEC VAX mini-computers.
 *      M68K  for Motorola MC68000 family processors
 *      NS16   for National Semiconductor NS16000 series processors.
 */
#define INTEL

/*
 * Operating System.
 * Used for OS specific interprocess communication and
```

```
 * other OS specific facilities.
 * Supported Operating Systems are:
 *      SIMU   No operating system at all.
 *      SYSV   System V Unix release {2,3}.
 *      BSD    Berkeley Unix release 4.{2,3}.
 *      MSDOS
 *      VMS
 */
#define SYSV

/*
 * Compiler.
 * Used to identify the compiler.
 * This can be used to work around abnormalities of the C compiler.
 * Supported Compilers are:
 *      KANDR Traditional compilers.
 *      ANSIC Standard conformant.
 */
#define ANSIC

#endif /*}*/
```

Figure 3.1: Operating Environment Configuration

3.7 Global Standard Definitions (estd)

All Open C Environment source modules include the global portable standard definitions definitions file estd.h. Documentation on estd.h is provided in appendix A. Basic portable data type definitions are defined in estd.h.

Chapter 4

COMMON FEATURES OF A LAYER

4.1 A Layer's Interfaces

Each layer has a number of typical exposed interfaces. As an example let's consider a sample software layer called "Some Service Provider" (SSP_). (SSP_) is expected to provide some defined services to the layer above it (UPPER_). It can rely on the services offered by the layer below it (LOWER_). The usual interfaces of the SSP_ layer are listed below:

```
Initialization:
        SSP_init()
        SSP_term()

SAP Mangement Interface:
        SSP_sapCreate()
        SSP_sapDelete()

Upper Interface:
        SSP_actionPrim(), ...
        (*upperEventPrim)(), ...

Lower Interface:
        LOWER_ActionPrim(), ...
        ssp_lowerEventPrim(), ...

Layer Management Interface:
        SSP_lmInit()
        SSP_lmActionPrim(), ...
        (*lmEventPrim)(), ...

Open C Platform Interface:
        TM_, EH_, SCH_, DU_, TMR_, ...
```

Consistent with the naming conventions mentioned in the previous section, all exposed interfaces of this module are prefixed by SSP_. The indirect function invocation (*func)() notation is used to indicate the event primitive interaction with other layers.

4.1.1 Initialization

Each layer can expect to be initialized before providing any services. During the initialization, the module can obtain its required resources and become ready to provide its expected services. The entry point to initialize the SSP_ layer is SSP_init().

Each Open C Layer must have a mechanism to be terminated and re-initialized. After termination a layer is not expected to maintain any history of what had happened prior to termination. The module must be re-initialized before being used. During a reset the layer should release all the resources that it had previously obtained. The entry point to terminate the SSP_ layer is SSP_term().

All initialization functions should be idempotent, meaning multiple invocations of the initialization facility do not result in an error. The following code fragment illustrates a conventional implementation of initialization and termination facilities.

```
static Bool virgin = TRUE;

Void SSP_init()
{
    if (virgin) {
        virgin = FALSE;         /* Only first time counts */
        /*
         * - Obtain necessary resources.
         * - Provide your services.
         */
    }
}

Void SSP_term()
{
    virgin = TRUE;              /* Can be re-initialized */
    /*
     * - Terminate your services.
     * - Release resources.
     */
}
```

4.1.2 SAP Management

All service provider layers have a means to support multiple service users simultaneously. This is supported through the concept of Service Access Points.

A service access point (SAP) is defined as the interface between a service user and a service provider. Each SAP is identified by the service provider through a SAP address selector. Above the network layer SAP address selectors are derived from SAP address suffixes.

Each service user is expected to create a service access point before using any of the services of the provider. Each layer has a function to create a SAP and a function to delete a SAP. Upon SAP creation, the service provider, associates SAP address selector of a service user entity with the address of a number of function(s) within the service user entity. These functions will be used by the service provider to handle indications and confirms (primitive events) at the upper interface.

4.1.3 Upper Interface

Each Open C Layer is expected to provide its services at its upper interface. The set of action primitives provided by the layer must be known by the service user. Entry points for events primitives must have been conveyed to the layer during the creation of the service access point. The calling sequence for event primitive must be consistent with the service user's expectations.

4.1.4 Lower Interface

Open C Layers may rely on the services provided by the layers below them. The set of action primitives provided by the layer below must be known. Entry points for event primitives are conveyed to the layer below during the SAP creation.

4.1.5 Layer Management Interface

Each layer has a layer management entity (LME) responsible for administration of the layer. This subject is discussed in more detail in chapter 4.

4.2 Usage of Resources

OSI resources such as SAPs and CEPs are often dynamically used. Creation, usage and deletion of these resources often follow a common pattern. This common pattern is described in this section.

Usage of resources is often through a descriptor based scheme. Through a create operation a unique descriptor referencing an instance of the resource is obtained. This descriptor is used as a reference to the resource in all future transactions. Two basic variations on this scheme are described below. The one way resource usage model and the two way resource usage model. The one way resource usage is typically more appropriate for interfaces that only requires action primitives. The two way resource usage is typically more appropriate for interfaces that require action and event primitives.

In the one way model, a resource typically has:

- An address.
- A provider reference.

```
FILE fopen(char *filename, char *permissions);
```

in stdio module is an example of the one way model for resource usage. fileName is the address of the resource and FILE is the provider's reference.

In the two way model, a resource typically has:

- An address.

- A user reference.
- A provider reference.

Let's consider the case of a Connection End Point.

```
CepProvRef conReq(CepUserRef cepUserRef, DstAddr dstAddr, ...);
Void (*conCnf)(CepUserRef cepUserRef, ...);
Void dataReq(CepProvRef cepProvRef, ...);
```

conReq and dataReq are action primitives, conCnf is an event primitive. Prior to the creation of a CEP, the service user prepares its private model of the CEP. During the creation of the CEP (conReq), a reference to this (cepUserRef) is conveyed to the service provider module. The service provider module (conReq) in turn returns a provider reference. All future action primitives dealing with the CEP will use the provider reference. All future event primitives dealing with the CEP will use the user reference.

4.3 Integration of Layers

The layered architecture plus the consistent layer interface provides for ease of integration. Several communication layer implementations can be integrated into a multi-layered communication software entity.

The Global module responsible for integration of the communication layers is expected to be called the "G_" module. A number of features are expected of the G_ module. These features are described in the next section.

The following basic structure is typical of the program that integrates one or more layers into a communication system.

```
- Set the run time configuration parameters.
- Initialize the run time environment and
  the communication layers.
- Wait for any significant event expected by any modules.
- Schedule the execution of the awakened module.
```

4.3.1 The G_ module

By convention the G_ module is responsible for integration of all other modules in the executable entity. By convention "g.h" is designated to contain global configuration and integration information to be shared among independent modules. G_Env may be shared among all modules to convey global environment information.

The specific nature of G_ module and "g.h" is specific to each target environment. The following figure illustrates the usage of g.h for configuration and integration purposes.

```
/*
 * SCCS Revision: %W% Released: %G%
 */

#ifndef _G_H_ /*{*/
#define _G_H_
```

```
#include "du.h"

typedef struct G_Env {
    Bool    hardReset;
    Bool    softReset;
} G_Env;

extern DU_Pool *G_mainDuPool;

#define K_schQuLen 22        /* Number of Scheduler queue items */

#endif /*}*/
```

Figure 4.1: Example of g.h

Potential usages of the G_ module are demonstrated in other examples in this book.

Chapter 5

MODULE MANAGEMENT ARCHITECTURE

The OSI Basic Reference Model introduces the concept of management within OSI, and identifies a category of management activity described as Systems Management. It is recognized that OSI Management Framework is young. Details of Management Information Services are presently not fully understood. However, it is within the scope of this book to address the issue of management.

The Network Management implementation architecture outlined here is based on [?]. The concepts described in [?] trickle down to the lower layers. Layer management facilities, should therefore be designed to support these concepts. This sections deals with Common Management Information Services as they apply to the lower layers.

Following figure illustrates OCE's view of Network Management Architecture.

Each layer has a Layer Management Entity (LME) which provides an interface for manipulating and monitoring that layer's performance. Each System has an entity that can access individual LME and perform Network Management Functions. This entity will be called "Layer Management Entity Interface" Common Management Information Services Element (CMISE) is responsible for monitoring the system on behalf of a Network Management Administrator (NMA).

The Common Management Information Protocol (CMIP) provides request/response service between a CMISE in one real open system and a CMISE in a second real open system which may be carrying out management activities in that real open system on behalf of the CMISE in the first open system. The CMIP also provides an event reporting service between an event reporting CMISE and an event monitoring CMISE.

5.1 Module Management (MM)

```
\#include "mm.h"

void
MM_init(char * pApplicationEntityInstanceName);

ReturnCode
MM_registerModule(char * pModuleName,
```

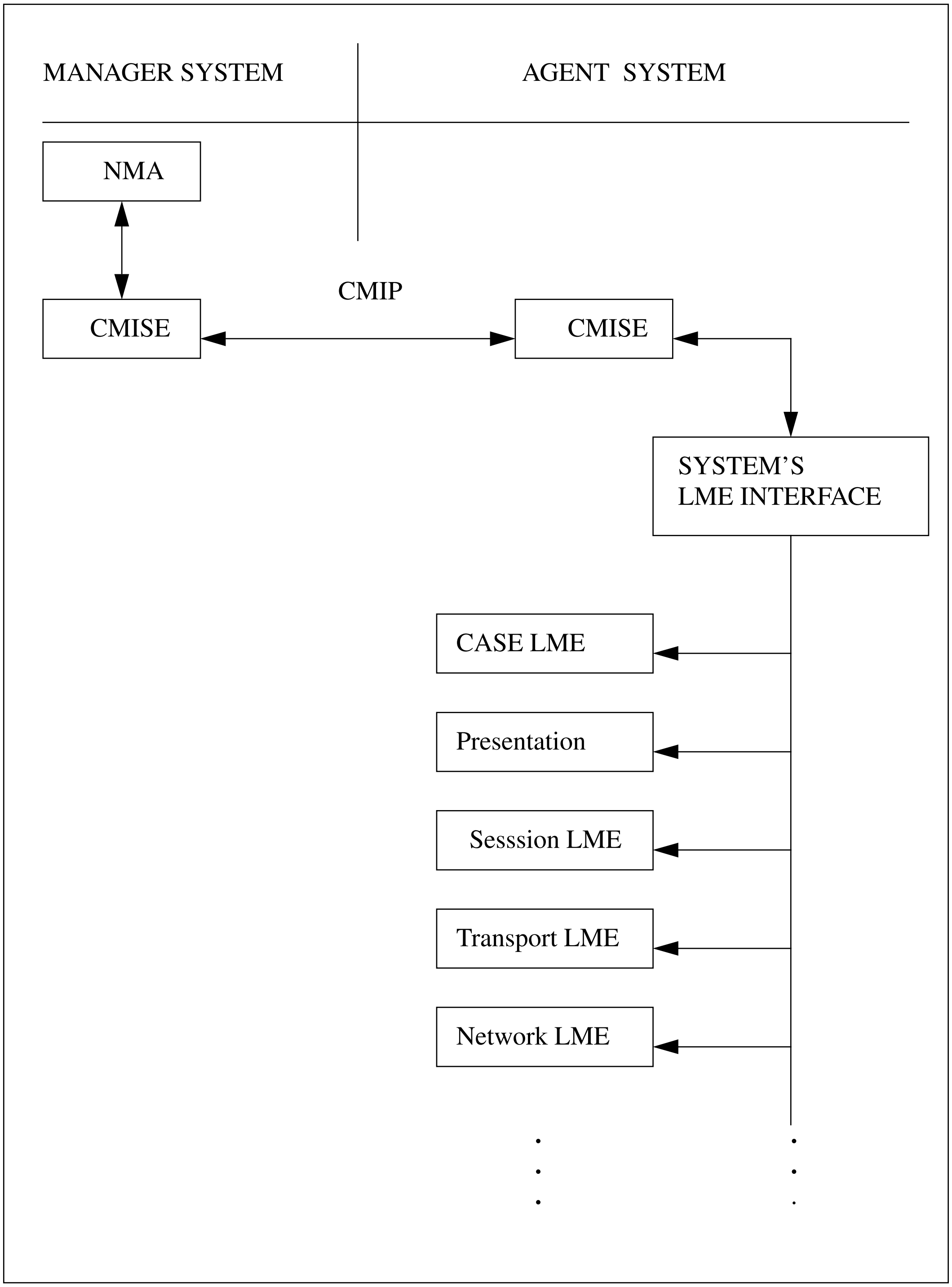

Figure 5.1: Network Management Architecture

```
                  void ** phModule);

ReturnCode
MM_registerManagableEntity(void * hModule,
                           MM_ManagableEntityType managableEntityType,
                           char * pManagableEntityName,
                           char * pIdentificationMessage,
                           OS_Uint16 initialNotifyMask,
                           void ** phManagableEntity);

ReturnCode
MM_setThreshold(void * hManagableEntity,
                MM_ThresholdType thresholdType,
                OS_Sint32 value);

ReturnCode
MM_incrementValue(void * hManagableEntity,
                  OS_Sint32 incrementBy);

ReturnCode
MM_startTimer(void * hManagableEntity,
              OS_Uint32 milliseconds);

ReturnCode
MM_stopTimer(void * hManagableEntity);

ReturnCode
MM_logMessage(void * hManagableEntity,
              char * pFormat,
              ...);

ReturnCode
MM_registerDestination(ReturnCode (* pfAlert)(char * pApplicationEntityInstanceName,
                                              char * pModuleName,
                                              char * pIdentificationMessage,
                                              MM_EventType eventType,
                                              ...),
                       OS_Uint16 initialNotifyMask,
                       void ** phDestination);

ReturnCode
MM_modifyDestination(void * hDestination,
                     OS_Uint16 newNotifyMask);

ReturnCode
MM_processEvents(OS_Boolean * pbFoundOne);
```

```
ReturnCode
MM_getValueByHandle(void * hManagableEntity,
                    void * pValue);
ReturnCode
MM_getValueByName(char * pModuleName,
                  char * pManagableEntityName,
                  void * pValue);

ReturnCode
MM_setValueByHandle(void * hManagableEntity,
                    void * pValue);

ReturnCode
MM_setValueByName(char * pModuleName,
                  char * pManagableEntityName,
                  void * pValue);
```

Note that there are no ”modify”, ”delete”, or ”create” operations. Creation must occur at the Module Management Entity level. Deletion can be accomplished by setting the Notification Mask to zero. Modify is accomplished via the ”set” operation.

The Module Management Module uses the following data types:

```
typedef enum MM_ManagableEntityType
{
    MM_ManagableEntityType_CounterSigned,       /* Value Type: OS_Sint32 */
    MM_ManagableEntityType_CounterUnsigned,     /* Value Type: OS_Uint32 */
    MM_ManagableEntityType_GuageSigned,         /* Value Type: OS_Sint32 */
    MM_ManagableEntityType_GuageUnsigned,       /* Value Type: OS_Uint32 */
    MM_ManagableEntityType_String,              /* Value Type: STR_String */
    MM_ManagableEntityType_Timer,               /* No maintained value */
    MM_ManagableEntityType_Log                  /* No maintained value */
} MM_ManagableEntityType;
```

These are the currently supported types of entities which we can manage.

```
typedef enum MM_NotificationType
{
    /* Counter events */
    MM_NotificationType_HighPriCounter  = (1 << 13)
    MM_NotificationType_MidPriCounter   = (1 << 7)
    MM_NotificationType_LowPriCounter   = (1 << 1)

    /* Event requires immediate notification */
    MM_NotificationType_Urgent          = (1 << 15)
```

```
    /* Event is only informational and may not require any action */
    MM_NotificationType_Info            = (1 << 0)

    /* Generic values for all possible bits */
    MM_NotificationType_15              = (1 << 15) /* also Urgent */
    MM_NotificationType_14              = (1 << 14)
    MM_NotificationType_13              = (1 << 13) /* also HighPriCounter */
    MM_NotificationType_12              = (1 << 12)
    MM_NotificationType_11              = (1 << 11)
    MM_NotificationType_10              = (1 << 10)
    MM_NotificationType_9               = (1 << 9)
    MM_NotificationType_8               = (1 << 8)
    MM_NotificationType_7               = (1 << 7)  /* also MidPriCounter */
    MM_NotificationType_6               = (1 << 6)
    MM_NotificationType_5               = (1 << 5)
    MM_NotificationType_4               = (1 << 4)
    MM_NotificationType_3               = (1 << 3)
    MM_NotificationType_2               = (1 << 2)
    MM_NotificationType_1               = (1 << 1)  /* also LowPriCounter */
    MM_NotificationType_0               = (1 << 0)  /* also Info */
} MM_NotificationType;
```

Each managable entity may be assigned one or more notification types which are to be generated when an event on that managable entity is raised. The module itself may specify an initial set of notification types for the managable entity, but the manager entity may modify that set.

A general rule to follow in deciding what notification type to use, is that the higher-order bits should indicate more urgent events while lower-order bits should indicate less urgent events.

NOTE: The maximum number of bits allowed here is 16, as enum types are 16-bits when using some compilers.

```
typedef enum MM_EventType
{
    MM_EventType_MaxThresholdExceededSigned,
    /*
     * Parameters passed to the Alert function when an event of this
     * type is raised:
     *
     *     Two optional parameters are passed:
     *
     *           - the threshold value, as an "OS_Sint32"
     *           - the value which caused the event by exceeding the
     *             threshold, as an "OS_Sint32"
     *
     */

    MM_EventType_MaxThresholdExceededUnsigned,
    /*
```

```
 * Parameters passed to the Alert function when an event of this
 * type is raised:
 *
 *     Two optional parameters are passed:
 *
 *           - the threshold value, as an "OS_Uint32"
 *           - the value which caused the event by exceeding the
 *             threshold, as an "OS_Uint32"
 *
 */

MM_EventType_MinThresholdExceededSigned,
/*
 * Parameters passed to the Alert function when an event of this
 * type is raised:
 *
 *     Two optional parameters are passed:
 *
 *           - the threshold value, as an "OS_Sint32"
 *           - the value which caused the event by exceeding the
 *             threshold, as an "OS_Sint32"
 *
 */

MM_EventType_MinThresholdExceededUnsigned,
/*
 * Parameters passed to the Alert function when an event of this
 * type is raised:
 *
 *     Two optional parameters are passed:
 *
 *           - the threshold value, as an "OS_Uint32"
 *           - the value which caused the event by exceeding the
 *             threshold, as an "OS_Uint32"
 *
 */

MM_EventType_TimerExpired,
/*
 * Parameters passed to the Alert function when an event of this
 * type is raised:
 *
 *     No optional parameters are passed.
 */

MM_EventType_LogMessage
```

```
    /*
     * Parameters passed to the Alert function when an event of this
     * type is raised:
     *
     *     One optional parameter is passed:
     *
     *          - the log message string, as a "char *"
     */
} MM_EventType;
```

This is the current set of events which may be generated by a module's management entity.
IMPORTANT NOTE:
If additional event types are added, be sure to add comments specifying what optional parameters are passed to the Alert function (see MM_registerDestination()) when an event of this type is raised.

5.1.1 Module Management Initialization

This function is to be called by all applications making use of any of the Module Management facilities.

```
void
MM_init(char * pApplicationEntityInstanceName);
```

Initialize the Module Management Entity module.

```
Parameters:

  pApplicationEntityInstanceName --
    The name of the application entity instance in which the module
    is located.  Note that if the same application is running in
    more then one instance, the instance name must be unique in
    each one.
```

5.1.2 Module Management Entity (MME) facilities

These functions are to be called by each module which wishes to use module management facilities.

Register Module

```
ReturnCode
MM_registerModule(char * pModuleName,
                  void ** phModule);
```

Allocate management resources for a code module or protocol layer.

```
Parameters:

  pModuleName --
```

```
    The name of the module for which managable entities are to be
    registered.

  phModule --
    Pointer to a handle.  The handle is generated by this function.
    Future requests to register a managable entity will use this
    handle.
```

```
ReturnCode
MM_registerManagableEntity(void * hModule,
                           MM_ManagableEntityType managableEntityType,
                           char * pManagableEntityName,
                           char * pIdentificationMessage,
                           OS_Uint16 initialNotifyMask,
                           void ** phManagableEntity);
```

Register a managable entity for use by the specified module.

```
Parameters:

  hModule --
    A module handle previously returned by MM_registerModule().

  managableEntityType --
    The type of managable entity being registered.

  pManagableEntityName --
    The name of the managable entity being registered.  This name must be
    unique within the scope of this module.

  pIdentificationMessage --
    An identification string which will be passed to the module management
    agent when an event is raised.

  initialNotifyMask --
    Bits identifying the urgency of an event rasied for this managable
    entity.  Multiple bits may be specified, but this use is discouraged.

  phManagableEntity --
    Pointer to a handle.  The handle is generated by this function.  Future
    requests to set thresholds, modify values, etc. will require use of
    this handle.
```

subsubsectionSet Threshold

```
ReturnCode
MM_setThreshold(void * hManagableEntity,
```

```
               MM_ThresholdType thresholdType,
               OS_Sint32 value);
```

Set the maximum or minimum threshold value for a managable entity.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously returned by
    MM_registerManagableEntity().

  thresholdType --
    Indication of whether the threshold to be set is a Maximum threshold or
    a Minimum threshold.

  value --
    Value to which the threshold should be set.

Note:
  Thresholds are only applicable to certain managable entity types, such as
  Counters and Guages.
```

Increment Value

```
ReturnCode
MM_incrementValue(void * hManagableEntity,
                  OS_Sint32 incrementBy);
```

Increment the numeric value of the specified managable entity (probably either a counter or a guage) by the specified value.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously returned by
    MM_registerManagableEntity().

  incrementBy --
    Amount by which the managable entity's value should be incremented.
    The increment value may be negative to decrement the value.
```

Timer

```
ReturnCode
MM_startTimer(void * hManagableEntity,
              OS_Uint32 milliseconds);
```

Start a timer. When it expires, an event will be raised.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously returned by
    MM_registerManagableEntity().

  milliseconds --
    Number of milliseconds before the timer should expire.
```

```
ReturnCode
MM_stopTimer(void * hManagableEntity);
```

Stop a previously started timer.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously returned by
    MM_registerManagableEntity().
```

Logging Message

```
ReturnCode
MM_logMessage(void * hManagableEntity,
              char * pFormat,
              ...);
```

Generate a message for logging, using a printf-style format.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously returned by
    MM_registerManagableEntity().

  pFormat --
    Printf-style format string specifying the format for the remainder of
    the parameters.

  ... --
    Additional parameters, as specified by pFormat.
```

5.1.3 Module Management Agent facilities

Each MMA talks to multiple Module Management Entities (MME's).

```
ReturnCode
MM_registerDestination(ReturnCode (* pfAlert)(char * pApplicationEntityInstanceName,
                                              char * pModuleName,
                                              char * pIdentificationMessage,
                                              MM_EventType eventType,
                                              ...),
                       OS_Uint16 initialNotifyMask,
                       void ** phDestination);
```

Register a new destination to which events may be sent. A destination is a place where an event is sent. All Module Management Agents should register at least one destination – the Module Management Manager. Additional destinations may be registered, such as to a log file, to send email, etc.

```
Parameters:

  pfAlert --
    Pointer to a function which will be called when events are
    destined to this registered destination.

    If "pfAlert" is NULL, a default function is used, which sends events
    for this destination to a non-standard-based Module Management Manager,
    using a non-standard-based data format.

    When the function pointed to by this parameter is ultimately
    called, it will be passed a set of zero or more optional
    parameters which are specific to the type of event which has
    been raised.  See the comments associated with the definition
    of MM_EventType.

  initialNotifyMask --
    Bits specifying that events of notification types include in this mask
    are to be sent to this destination (in addition, possibly, to other
    destinations).

  phDestination
    Pointer to a handle.  The handle is generated by this function.  Future
    requests to modify the notification mask for this destination will
    require use of this handle.
```

```
ReturnCode
MM_modifyDestination(void * hDestination,
                     OS_Uint16 newNotifyMask);
```

Modify the set of notification types which should be sent to this destination.

```
Parameters:
```

```
  hDestination --
    Handle, previously provided by MM_registerDestination().

  newNotifyMask --
    New bit mask indicating which notification levels are to be sent to
    this destination.
```

Process Events

```
ReturnCode
MM_processEvents(OS_Boolean * pbFoundOne);
```

Event notification does not happen asynchronously. The reason for this is that the event could be raised during interrupt routines, critical sections, etc. We therefore enqueue the event notification for action when this function is called.

This function should be called on a regular basis, either in a main loop, or via a timer expiration.

```
Parameters:

  pbFoundOne --
    Pointer to a boolean variable, which is set to TRUE by this function if
    an event was found to process.  This variable is _not_ modified if no
    event was found to process, enabling a pointer to the same variable to
    be passed to multiple functions to see if any of them had anything to
    do.

    This pointer may be NULL if an indication of whether an event was
    processed is not required.
```

Get Current Value

```
ReturnCode
MM_getValueByHandle(void * hManagableEntity,
                    void * pValue);
```

Get the current value of a managable entity.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously provided by
    MM_registerManagableEntity().

  pValue --
    Pointer to the location where the current value of the specified
    managable entity is to be placed.  It is up to the caller to provide a
```

```
    pointer to the correct type of variable into which the value will be
    placed.
```

```
ReturnCode
MM_getValueByName(char * pModuleName,
                  char * pManagableEntityName,
                  void * pValue);
```

Get the current value of a managable entity.

```
Parameters:

  pModuleName --
    Name of the module in which the manangable entity resides.

  pManagableEntityName --
    Name of the managable entity for which the value is desired.

  pValue --
    Pointer to the location where the current value of the specified
    managable entity is to be placed.  It is up to the caller to provide a
    pointer to the correct type of variable into which the value will be
    placed.
```

Set Current Value

```
ReturnCode
MM_setValueByHandle(void * hManagableEntity,
                    void * pValue);
```

Set the current value of a managable entity.

```
Parameters:

  hManagableEntity --
    Handle to a managable entity, previously provided by
    MM_registerManagableEntity(), for which the value is to be
    modified.

  pValue --
    Pointer to the new value for this managable entity.  It is up
    to the caller to provide a pointer to the correct type of
    variable into which the value will be placed.
```

```
ReturnCode
MM_setValueByName(char * pModuleName,
                  char * pManagableEntityName,
                  void * pValue);
```

Get the current value of a managable entity.

```
Parameters:

  pModuleName --
    Name of the module in which the manangable entity resides.

  pManagableEntityName --
    Name of the managable entity for which the value is to be
    modified.

  pValue --
    Pointer to the new value for this managable entity.  It is up
    to the caller to provide a pointer to the correct type of
    variable into which the value will be placed.
```

5.2 Layer Management Entity Interface

Each LME should define the complete set of management information types and facilities that it supports. Mechanism for access to LME of each layer is same as the layer's other interfaces – Direct non-blocking function calls with a well defined set of arguments.

Chapter 6

OPEN C PLATFORM

Open C Platform is a set of well defined interfaces and service definitions for basic facilities needed for implementation of lower layer OSI protocols in C.

Platform facilities are grouped into sets of related facilities. Module naming conventions, mentioned earlier is used to highlight this grouping. A list of these common facilities is provided in the following table:

Module Name	Facility Description	Environment Dependencies
SF_	System Facilities	Yes
SCH_	Scheduling	Yes
BO_	Byte Ordering	Yes
BS_	Byte String Manipulation	Can be
QU_	Linked List Management	No
SEQ_	QU_ extensions	SF_
EH_	Exception Handling	Yes
TM_	Tracing	Yes
DU_	Data Unit manipulation	SF_
TMR_	Timer Management	Yes
SAP_	SAP Address Management	No
RELID_	Release Identifier	
LIC_	License Checking	

EH_ and TM_ provide exception handling, event logging and tracing facilities. This type of basic facilities are required by any type of serious software development.

SF_ (System Facilities) module defines an interface for a number of inherently non portable facilities. Implementation of many of the facilities defined here can be made portable by relying on SF_ facilities.

SCH_ (Scheduling) module provides for efficient usage of the CPU in multi-processing environments.

BO_ provides for simple CPU independent value representation. BS_ defines an interface for manipulation of blocks of data.

6.1 OCP Interfaces

6.2 Environment Specific Facilities (SF_)

```
#include "sf.h"

Ptr SF_memObtain(Uns nuOfBytes);
SuccFail SF_memRelease(Ptr data);

SF_Status SF_critBegin();
Void SF_critEnd(SF_Status status);

Int SF_quInsert(QU_Head *head, QU_Elem *elem);
Int SF_quRemove(QU_Head *head, Ptr *elem);
```

A set of inherently environment dependent services are expected of the operating environment. Environment independent services and interfaces defined in Open C Platform can be implemented by relying on the primitive environment specific facilities available in each environment.

Ability to Obtain memory from the environment, a periodic interrupt and protection against preemption are among the few environment specific facilities that are assumed by Open C Platform. Other environment specific facilities such as atomic queue operations, and timer facilities may be available in some environments. When available these facilities may be used for efficient implementation of Open C Platform. Facilities described in this section are considered to be low level. Their direct usage by the application is not recommended.

6.2.1 Obtaining Memory

During the initialization each module may reserve some memory for its usage and upon a soft reset each module may return the memory back to the environment. The lower layers should not assume the existence of a dynamic memory allocation facility such as malloc. All the memory that required by a module is expected be reserved upon initialization.

```
Ptr SF_memObtain(Uns nuOfBytes);
SuccFail SF_memRelease(Ptr data);
```

Provide a model for such environment specific facilities.

6.2.2 Synchronization

Asynchronous intercations with synchronous processing can result into inconsistencies. To prevent this the availability of a simple preemption protection mechanism is assumed. When the synchronous software is accessing a critical resource, it protects itself against preemption and once the critical section is completed preemption status is restored.

```
SF_Status SF_critBegin();
Void SF_critEnd(SF_Status status);
```

Provide a model for this service. Enabling and disabling interrupts is an extreme way of implementing this facilities in un-hosted environments.

6.2.3 Atomic Queue Operations

Most modern CPUs include support for atomic queue (circular doubly linked list) operations. When available this facility may be exploited for coordination of synchronous and asynchronous interactions.

```
Int SF_quInsert(QU_Head *head, QU_Elem *elem);
Int SF_quRemove(QU_Head *head, QU_Elem **elem);
```

Provide a model for this service. The QU_ module describes the characteristics of circular doubly linked lists. QU_ facilities are not protected against asynchronous preemption and should not be used when asynchronous preemption can result into inconsistencies. SF_quInsert and SF_quRemove are expected to be atomic (non-preemptable during the entire operation).

SF_quInsert inserts elem at the tail of head. elem need not be initialized. If head was empty, -1 is returned. Otherwise 0 is returned.

SF_quRemove removes the first element of the queue from the head if there is one. If head was not empty *elem is a pointer to the removed element. The removed element will not be initialized at the completion of SF_quRemove. If head was empty, SF_quRemove returns -1 and *elem is untouched. Otherwise 0 is returned.

An example implementation of SCH_ module based on SF_qu facilities is illustrated in the following code fragment.

```
#ifdef SCCS_VER /*{*/
static char sccs[] = "%W%  Released: %G%";
#endif /*}*/

#include "estd.h"
#include "sf.h"
#include "eh.h"

/*
 * Scheduler Information.
 */
typedef struct SchInfo {
    struct SchInfo *next;
    struct SchInfo *prev;
    Int (*func)();        /* Function to Call */
    Ptr arg;
} SchInfo;
```

```
typedef struct SchInfoSeq {
    SchInfo *first;
    SchInfo *last;
} SchInfoSeq;

STATIC SchInfoSeq availSchInfo;
STATIC SchInfoSeq activeSchInfo;

STATIC SchInfo *schInfoBuf;
STATIC SchInfo *schInfoBufEnd;


Void SCH_init(maxSchInfo)
Int maxSchInfo;
{
    SchInfo *schInfo;

    /*
     * Create a Pool
     */
    schInfoBuf = (SchInfo *)
                    SF_memObtain(maxSchInfo * sizeof(*schInfoBuf));
    if (!schInfoBuf) {
        EH_fatal("SF_memObtain");
    }
    schInfoBufEnd = &schInfoBuf[maxSchInfo -1];
    QU_init(&availSchInfo);
    QU_init(&activeSchInfo);
    for (schInfo = schInfoBuf; schInfo <= schInfoBufEnd; ++schInfo){
        SF_quInsert(&availSchInfo, schInfo);
    }
}

Void SCH_term()
{
    SF_memRelease(schInfoBuf);
}

Void SCH_submit(func, arg)
Int (*func)();
Ptr arg;
{
    SchInfo *schInfo;
    Int status;

    /*
     * Queue Up the function and argument for synchronus processing.
     * Notice The Qu insertion must be protected.
     */
    if ((status = SF_quRemove(&availSchInfo, &schInfo)) != 0) {
        EH_fatal("No SchInfo");
        return ;
    }

    schInfo->func = func;
    schInfo->arg = arg;
```

```
    if (SF_quInsert(&activeSchInfo, schInfo)) {
        /* PORTATION SPECIFIC
         * Use environment specific facilities to wake up.
         */
         sys$wake(0, 0);
    }
}

Void SCH_block()
{
    /*
     * Should map into environment specific facilities
     * that puts this process into sleep.
     */
    if (activeSchInfo.first == &activeSchInfo) {
        /* PORTATION SPECIFIC
         * Use Environment Specific facilities to Block
         */
        sys$hiber();
    }
}

Void SCH_run()
{
    SchInfo *schInfo;
    Int status;

    while (SF_quRemove(&activeSchInfo, &schInfo) == 0) {
        (*schInfo->func)(schInfo->arg);
        SF_quInsert(&availSchInfo, schInfo);
    }
}
```

Figure 6.1: SF Example Source Code

In this example it is assumed that environment specific facilities for scheduling are provided through syshiber and syswake.

6.2.4 Scheduling

In multi-processing hosted environments, it is desirable to share the CPU with other processes. In these environments the availability of facilities for blocking and waking up are assumed. When available these facilities may be used in the implementation of SCH_ facilities.

SCH_ section defines the scheduling facility to be used by Open C Platform users.

6.2.5 Timer Facilities

Environment specific timer facilities are expected to be enhanced to conform to TMR_ facilities described later in this chapter. In un-hosted environments the only expected timer facility from the environment is a periodic interrupt.

6.3 Execution Scheduler (SCH_)

```
#include "sch.h"

Void SCH_init(Int nuOfSchQuItems);
Void SCH_reset();

Void SCH_submit(func, arg)
Int (*func)();
Ptr arg;

Void SCH_block();
Void SCH_run();
```

In environments where multi processing is supported, it may be desirable to time share the CPU with other processes. In this scenario when the communication software has no more work to do it can block and deliver CPU to other processes.

SCH_ module can be used for rescheduling of known modules. One of the common usages of SCH_ module rescheduling of further processing with in the same module. This happens most often to prevent re-entry to non-reentrant code.

Take the case of an (N-1) Action Primitive resulting into an (N-1) Event Primitive. If (N) layer code is non-reentrant, this should not happen. The expected behavior of (N-1) module is:

- Detect that an Action Primitive is resulting into an Event Primitive.
- Store the information required for the future issuance of the Event Primitive.
- Request re-scheduling of this layer.
- Complete the Action Primitive, and return.

The following example illustrates an example of such a sequence.

```
#ifdef SCCS_VER /*{*/
static char sccs[] = "%W%  Released: %G%";
#endif /*}*/

#include <stdio.h>
#include "estd.h"
#include "tm.h"
#include "getopt.h"
#include "g.h"

Void SSP_init(), SSP_sapCreate(), SSP_actionPrim();
Void USER_init();

PUBLIC G_Env G_env;
```

```
main(argc, argv)
Int argc;
String argv[];
{
    Int c;

    TM_init();

    while ((c = getopt(argc, argv, "T:t:")) != EOF) {
        switch ( c ) {
        case 'T':
        case 't':
            TM_setUp(optarg);
            break;
        case '?':
        default:
            G_exit(1);
        }
    }

    G_env.hardReset = FALSE;
    G_env.softReset = FALSE;

    while ( !G_env.hardReset ) {
        SCH_init(K_schQuLen);
        SSP_init();
        USER_init();
        TM_validate();

        while ( !G_env.softReset ) {
            SCH_block();
            SCH_run();
        }

        SCH_term();
    }

    G_exit(1);
}

G_exit(code)
Int code;
{
    exit(code);
}

/*
 * USER_ module.
 */

Void userEventPrim()
{
    printf("SSP Event Primitive, invoked by the scheduler\n");
    G_env.softReset = TRUE;
    G_env.hardReset = TRUE;
}
```

```
Void USER_init()
{
    SSP_sapCreate(userEventPrim);
    SSP_actionPrim();
}

/*
 * Some Service Provider (SSP_) Module.
 */
static Void (*sapEventPrim)();

Void SSP_init()
{
    /* Initialization could have been done here */
}

Void SSP_sapCreate(eventPrim)
Void (*eventPrim)();
{
    sapEventPrim = eventPrim;
}

Void SSP_actionPrim()
{
    /* Let's say this action primitive was to result
     * into an event primitive, but since we can not rely
     * on the user context being re-enterant.
     * The event primitive is scheduled for exection
     * outside the user's context.
     */
    printf("SSP Action Primitive, invoked by USER\n");
    SCH_submit(sapEventPrim, (Ptr)0);
}
```

Figure 6.2: SCH Example Source Code

6.4 Byte Ordering (BO_)

```
#include "bo.h"

Void BO_put1(netPtr, cpuValue)
Octet *netPtr; /* OUT */
Byte cpuValue;

Void BO_get1(cpuValue, netPtr) /* MACRO */
Octet *netPtr;
Byte cpuValue; /* OUT, VALUE EFFECTED */
```

```
BO_put2(netPtr, cpuValue)
Octet *netPtr; /* OUT */
MdUns cpuValue;

BO_get2(cpuValue, netPtr) /* MACRO */
Octet *netPtr;
MdUns cpuValue; /* OUT, VALUE EFFECTED */

BO_put4(netPtr, cpuValue)
Octet *netPtr; /* OUT */
LgUns cpuValue;

BO_get4(cpuValue, netPtr) /* MACRO */
Octet *netPtr;
LgUns cpuValue; /* OUT, VALUE EFFECTED */

BO_putN(netPtr, cpuPtr, n)
Octet *netPtr; /* OUT */
Byte *cpuPtr;
Int n;

BO_getN(cpuValue, netPtr, n)
Octet *netPtr;
Byte *cpuPtr; /* OUT */
Int n;
```

BO_ facilities provide primitive abstract data presentation facilities for simple types of values.

BO_ facilities convert CPU presentation of values to a machine independent byte ordering and vice versa. 8 bit values, 16 bit values, 32 bit values, and octet string types are converted to an abstract presentation commonly used by the lower layer protocols. This abstract presentation is expressed through a sequence of octets. Octets at the lower address always contain the most significant byte of the value. Least Significant Byte of the value is always at the higher address.

netPtr points to where the machine independent presentation value should be stored. Upon completion of all BO_ facilities netPtr is incremented by the appropriate value so that it can be used in subsequent BO_ operations. cpuValue is the machine dependent presentation of a value.

Get and Put verbs in BO_get/BO_put are with respect to the abstract presentation (the network). BO_get always converts the abstract presentation of contents of netPtr into a machine dependent value (cpuValue). Note that since cpuValue is an out put value and is not passed as a pointer BO_get facility must be implemented as a MACRO. BO_put always converts the machine dependent presentation of a value into an abstract presentation. netPtr is always incremented.

6.5 Byte String (BS_):

```
#include "bs.h"

BS_memCopy(src, dst, nuOfBytes)
Byte *src;
Byte *dst;
Int nuOfBytes;

BS_memCmp(src, dst, nuOfBytes)
Byte *src;
Byte *dst;
Int nuOfBytes;

BS_memFill(dst, value, nuOfBytes)
Byte *dst;
Byte value;
Int nuOfBytes;
```

A Byte String (BS_) is a consecutive memory address range. A byte string is specified by its starting address and its length. BS_ module operates on byte strings.

BS_memCopy copies *nuOfBytes* from string *src* to dst. *src* and *dst* are assumed to be non over lapping.

BS_memCmp compares byte string *src* against *dst* .

Byte string operations are often one of the most execution intensive parts of a protocol implementation. Environment specific facilities provided in the target environment can sometimes be used to implement these facilities more efficiently that they can be done as portable code.

6.6 Queue Module (QU_)

```
#include "queue.h"

typedef struct QU_Elem {
    Ptr next;
    Ptr prev;
} QU_Elem;

typedef struct QU_Head {
    Ptr first;
    Ptr last;
} QU_Head;

QU_init(QU_Elem *q);

QU_insert(QU_Elem *q1, QU_Elem *q2);

QU_remove(QU_Elem *q);
```

```
QU_move(QU_Elem *q1, QU_Elem *q2);

QU_HEAD

QU_ELEMENT

QU_INIT(p)

QU_INSERT(pInsertThisElement, pInFrontOfThisElement)

QU_PREPEND(pInsertThisElement, pAtBeginOfThisQueue)

QU_APPEND(pInsertThisElement, pAtEndOfThisQueue)

QU_REMOVE(p)

QU_MOVE(pMoveMe, pBeforeThisElement)

QU_FIRST(p)

QU_LAST(p)

QU_NEXT(p)

QU_PREV(p)

QU_EQUAL(p1, p2)
```

QU_ facilities provide a uniform mechanism for manipulation of doubly linked circular lists. A queue is a circular doubly linked list. Queues are often used for implementation of sequences and sets. A queue entry is linked to the next by a pair of pointers. The first pointer (next) is the forward link. It specifies the location of the succeeding entry. The second pointer (prev) is the backward link, it specifies the location of the preceding entry.

A queue is specified by a queue header (QU_Head). Structure of queue header is same as queue element (two pointers). The forward link of the header (first) is called head of the queue. The backward link of the header (last) is called the tail of the queue.

IMPORTANT NOTE 1:

Make no assumptions about the structure of this type. It may change in the future. All manipulation of queue elements must be accomplished solely by the functions in this queue module.

(As an example, if this code is ever moved into a multi-threading environment, some elements may be added to the header, for mutual exclusion while manipulating the queue pointers.)

IMPORTANT NOTE 2:

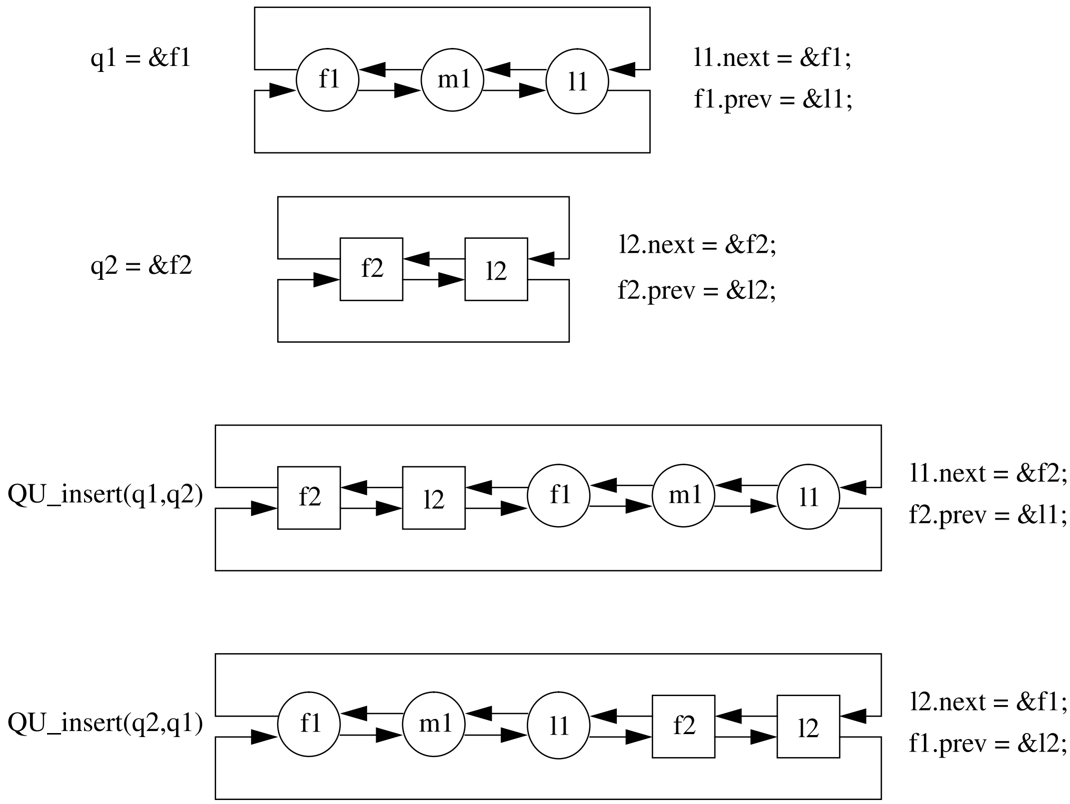

Figure 6.3: Queue Insertion

Do not declare queue pointers in your own structures. Instead, declare the first element of your structures as either QU_ELEMENT or QU_HEAD. No variable name is necessary.

Two basic operations can be performed on queues: insertion of entries and removal of entries.

QU_ facilities are not protected against asynchronous preemption and should not be used when asynchronous preemption can result into inconsistencies. SF_quInsert and SF_quRemove are expected to be atomic (non-preemptable during the entire operation).

QU_init, QU_insert, QU_remove and QU_move all operate on an abstract data type "QU_Elem". Objects that queue management facilities manipulate are expected to be data types that allow for a QU_Elem to be casted over them.

QU_init initializes a *QU_Elem* so that it can be used in subsequent operations. A QU_Elem is initialized by having its "next" and "prev" field point to itself. An initialized QU_Elem is an empty circular list.

QU_insert(q1, q2) inserts linked list q1 before list q2. The result is a linked list that contains all members of q1 and q2.

It is interesting to note that the order of arguments is not important. The result of QU_insert(q1, q2) and QU_insert(q2, q1) is the same. Figure 6.3 illustrates this.

The facility QU_remove removes QU_Elem q from the list to which it belonged. QU_Elem q is initialized upon completion of QU_remove.

QU_move moves QU_Elem q1 to end of q2. This is equivalent to the commonly used coding sequence:

```
QU_remove(q1);
QU_insert(q2, q1);
```

6.6.1 QU_ Macros

QU_ also provides the following set of macros to simplify coding.

- Use one of these as the first element within a structure which is to be a queue head or queue element.

```
QU_HEAD
QU_ELEMENT
```

- Use this macro to statically initialize a queue head.

```
QU_INITIALIZE(q)
```

- These macros may be used to initialize, insert, and remove queue elements. By using these macros, rather then direct calls to the functions, you assure future compatibility.

```
QU_INIT(p)
QU_INSERT(pInsertThisElement, pInFrontOfThisElement)
QU_PREPEND(pInsertThisElement, pAtBeginOfThisQueue)
QU_APPEND(pInsertThisElement, pAtEndOfThisQueue)
QU_REMOVE(p)
QU_MOVE(pMoveMe, pBeforeThisElement)
```

- Use these macros to find the first or last element on a queue.

```
QU_FIRST(p)
QU_LAST(p)
```

- Use these macros to find the next or previous element on a queue.

```
QU_NEXT(p)
QU_PREV(p)
```

- Use this macro to determine if two queue elements are equal. The following code may be used to iterate through a queue:

```
QU_EQUAL(p1, p2)

for (pElement = QU_FIRST(pHead);
     ! QU_EQUAL(pElement, pHead);
     pElement = QU_NEXT(pElement))
{
   ...
}
```

- If you want to free all elements on a queue, and each element was allocated with OS_alloc() and no element contains pointers to other dynamically allocated memory, you can use this function. Just pass it a pointer to the queue head.

  ```
  QU_FREE(pQHead){
  ```

6.7 Sequence Module (SEQ_)

```
#include "seq.h"

SEQ_PoolDesc SEQ_poolCreate(Int sizeOfElem, Int nuOfElems);
Void SEQ_poolFree(SEQ_PoolDesc pool);

Ptr SEQ_elemObtain(SEQ_PoolDesc pool);
Void SEQ_elemRelease(SEQ_PoolDesc pool, Ptr elem);
```

SEQ_ module provides simple fixed size dynamic memory allocation capabilities for linked list elements. In conjunction with the QU_ module, sequences and sets implemented as linked lists can conveniently be maintained.

Memory for a number of elements within a sequence can initially be obtained through SEQ_poolCreate facility. As new elements of a set or a sequence are needed they can be obtained through SEQ_elemObtain facility. Sequence or set elements can be released back into the pool through the SEQ_elemRelease facility.

6.7.1 Example Usage

The following code fragment demonstrate the use of Queue management facilities.

```
#ifdef SCCS_VER /*{*/
static char sccs[] = "%W%     Released: %G%";
#endif /*}*/

#include <stdio.h>
#include "estd.h"
#include "queue.h"
#include "seq.h"

typedef struct SomeInfo {
    struct SomeInfo *next;
    struct SomeInfo *prev;
#define DATASIZE 16 /* No Special Significance */
    Char data[DATASIZE];
    Int len;
} SomeInfo;

typedef struct SomeInfoSeq {
    SomeInfo *first;
    SomeInfo *last;
} SomeInfoSeq;
```

```
SEQ_PoolDesc someInfoPool;
SomeInfoSeq someInfoSeq;

Void seqInsert(), seqProcess(), process();
main()
{
    static Char *someData = "Some Data";

#define POOLSIZE 22 /* 22 is one of my favorite numbers */
    someInfoPool = SEQ_poolCreate(sizeof(*someInfoSeq.first),
                                  POOLSIZE);
    QU_init(&someInfoSeq);

    seqInsert(someData, strlen(someData)+1);
    seqProcess();
}

/*
 * Insert An Element into someInfoSeq
 */
Void seqInsert(data, len)
Char *data;
Int len;
{
    SomeInfo *someInfo;

    someInfo = (SomeInfo *) SEQ_elemObtain(someInfoPool);
    BS_memCopy(data, someInfo->data, len);
    someInfo->len = len;
    QU_insert(&someInfoSeq, someInfo);
}

Void seqProcess()
{
    SomeInfo *someInfo;
    Char *p;

    while ((someInfo = someInfoSeq.first) !=
            (SomeInfo *) &someInfoSeq) {
        QU_remove(someInfo);
        process(&someInfo->data[0], someInfo->len);
        SEQ_elemRelease(someInfoPool, someInfo);
    }
}

Void process(data, len)
Char *data;
Int len;
{
    printf("Processing %s\n", data);
}
```

Figure 6.4: QU and SEQ Example Usage Source Code

This example does not perform any useful task but demonstrates how the Queue manipulation facilities can be used to transfer some data through a queue. Flow of this example program is:

```
/* Call Graph */
```

6.8 Non-Volatile Queue (NVQ_)

6.8.1 NVQ_ReturnCode

```
enum NVQ_ReturnCode
{
    NVQ_RC_CreateFailed                          = (1 | ModId_Nvq),
    NVQ_RC_OpenFailed                            = (2 | ModId_Nvq),
    NVQ_RC_WriteError                            = (3 | ModId_Nvq),
    NVQ_RC_ReadError                             = (4 | ModId_Nvq),
    NVQ_RC_Overflow                              = (5 | ModId_Nvq),
    NVQ_RC_NonExistantElement                    = (6 | ModId_Nvq),
    NVQ_RC_DataSizeTooLarge                      = (7 | ModId_Nvq),
};


typedef int     NVQ_Element;

#define NVQ_NO_ELEMENTS         ((NVQ_Element) -1)
```

6.8.2 NVQ_create

```
ReturnCode
NVQ_create(char * pQueueName,
           OS_Uint16 maxNumElements,
           OS_Uint32 maxElementSize,
           void ** phQueue);
```

Create a Non-Volatile Queue.

```
Parameters:

  pQueueName --
    Identifier, or name for this queue.

  maxNumElements --
    Maximum number of elements that may be on this queue (ever).

  maxElementSize --
    Maximum number of octets of data in any one element.
```

```
  phQueue --
    Pointer to a location in which a handle to this queue is placed, upon
    successful completion of this function.
```

6.8.3 NVQ_open

```
ReturnCode
NVQ_open(char * pQueueName,
         OS_Uint16 * pMaxNumElements,
         OS_Uint32 * pMaxElementSize,
         void ** phQueue);
```

Open an already-existant Non-Volatile Queue.

```
Parameters:

  pQueueName --
    Identifier, or name for this queue.

  pMaxNumElements --
    Pointer to a location in which, upon successful completion of this
    function, the maximum number of queue elements will be placed.

  pMaxElementSize --
    Pointer to a location in which, upon successful completion of this
    function, the maximum number of octets in any element's data will be
    placed.

  phQueue --
    Pointer to a location in which a handle to this queue is placed, upon
    successful completion of this function.
```

6.8.4 NVQ_close

```
void
NVQ_close(void * hQueue);
```

Close a previously opened or created Non-Volatile Queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_create() or NVQ_open(), to the queue
    to be closed.
```

6.8.5 NVQ_delete

```
ReturnCode
NVQ_delete(charpQueueName);
```

Delete a Non-Volatile Queue.

```
Parameters:

  pQueueName --
    Identifier, or name of the queue to be deleted.
```

6.8.6 NVQ_VIEW_HEAD

```
#define NVQ_VIEW_HEAD(hQueue, ppData, pDataSize)                    \
                nvq_viewData(hQueue, ppData, pDataSize, NVQ_Location_Head)
```

Retrieve a pointer to the data in the element at the head of the queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  ppData --
    Pointer, of type unsigned char **, to a location in which to put a
    pointer to the data.

  pDataSize --
    Pointer, of type OS_Uint32 *, to a location in which to put Number of
    octets of data pointed to by *ppData.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.

WARNING:

  DO NOT MODIFY THE DATA POINTED TO BY THE RETURNED DATA POINTER!
```

6.8.7 NVQ_VIEW_TAIL

```
#define NVQ_VIEW_TAIL(hQueue, ppData, pDataSize)                    \
                nvq_viewData(hQueue, ppData, pDataSize, NVQ_Location_Tail)
```

Retrieve a pointer to the data in the element at the tail of the queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  ppData --
    Pointer, of type unsigned char **, to a location in which to put a
    pointer to the data.

  pDataSize --
    Pointer, of type OS_Uint32 *, to a location in which to put Number of
    octets of data pointed to by *ppData.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.

WARNING:

  DO NOT MODIFY THE DATA POINTED TO BY THE RETURNED DATA POINTER!
```

6.8.8 NVQ_VIEW_ELEM

```
#define NVQ_VIEW_ELEM(hQueue, ppData, pDataSize, elem)  \
                nvq_viewData(hQueue, ppData, pDataSize, \
                             NVQ_Location_Element, (NVQ_Element) (elem))
```

Retrieve a pointer to the data in the element at the specified location in the queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  ppData --
    Pointer, of type unsigned char **, to a location in which to put a
    pointer to the data.

  pDataSize --
    Pointer, of type OS_Uint32 *, to a location in which to put Number of
    octets of data pointed to by *ppData.

  elem --
    Identifier, of type NVQ_Element, specifying for which element a pointer
```

```
    to the data is to be returned.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.

WARNING:

  DO NOT MODIFY THE DATA POINTED TO BY THE RETURNED DATA POINTER!
```

6.8.9 NVQ_INSERT_AT_HEAD

```
#define NVQ_INSERT_AT_HEAD(hQueue, pData, dataSize)     \
                nvq_insert(hQueue,                      \
                           (unsigned char *) (pData),   \
                           dataSize,                    \
                           NVQ_Location_Head)
```

Insert an element at the head of a Non-Volatile Queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  pData --
    Pointer to the start of the data to be inserted in this queue element.

  dataSize --
    Number of octets of data pointed to by pData.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.
```

6.8.10 NVQ_INSERT_AT_TAIL

```
#define NVQ_INSERT_AT_TAIL(hQueue, pData, dataSize)     \
                nvq_insert(hQueue,                      \
                           (unsigned char *) (pData),   \
                           dataSize,                    \
                           NVQ_Location_Tail)
```

Insert an element at the tail of a Non-Volatile Queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  pData --
    Pointer to the start of the data to be inserted in this queue element.

  dataSize --
    Number of octets of data pointed to by pData.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.
```

6.8.11 NVQ_INSERT_BEFORE

```
#define NVQ_INSERT_BEFORE(hQueue, pData, dataSize, beforeMe)    \
                nvq_insert(hQueue,                               \
                           (unsigned char *) (pData),            \
                           dataSize,                             \
                           NVQ_Location_Before,                  \
                           beforeMe)
```

Insert an element before the specified element of a Non-Volatile Queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  pData --
    Pointer to the start of the data to be inserted in this queue element.

  dataSize --
    Number of octets of data pointed to by pData.

  beforeMe --
    Identifier, of type NVQ_Element, of the element before which this new
    element should be inserted into the queue.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.
```

6.8.12 NVQ_INSERT_AFTER

```
#define NVQ_INSERT_AFTER(hQueue, pData, dataSize, afterMe)      \
                nvq_insert(hQueue,                              \
                           (unsigned char *) (pData),          \
                           dataSize,                            \
                           NVQ_Location_After,                  \
                           afterMe)
```

Insert an element after the specified element of a Non-Volatile Queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    on which an element is to be inserted.

  pData --
    Pointer to the start of the data to be inserted in this queue element.

  dataSize --
    Number of octets of data pointed to by pData.

  aftere --
    Identifier, of type NVQ_Element, of the element after which this new
    element should be inserted into the queue.

Returns:

  Success upon success insertion; a non-success ReturnCode otherwise.
```

6.8.13 NVQ_REMOVE_HEAD

```
#define NVQ_REMOVE_HEAD(hQueue)                          \
                nvq_remove(hQueue, NVQ_Location_Head)
```

Remove the element from the Non-Volatile Queue, which is at the queue's head.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    from which the element is to be removed.

Returns:

  Success upon success removal; a non-success ReturnCode otherwise.
```

6.8.14 NVQ_REMOVE_TAIL

```
#define NVQ_REMOVE_TAIL(hQueue)                        \
                nvq_remove(hQueue, NVQ_Location_Tail)
```

Remove the element from the Non-Volatile Queue, which is at the queue's tail.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    from which the element is to be removed.

Returns:

  Success upon success removal; a non-success ReturnCode otherwise.
```

6.8.15 NVQ_REMOVE_ELEM

```
#define NVQ_REMOVE_ELEM(hQueue, elem)                          \
                nvq_remove(hQueue, NVQ_Location_Element, (NVQ_Element) elem)
```

Remove a specified element from the Non-Volatile Queue

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    from which the element is to be removed.

  elem --
    Identifier, of type NVQ_Element, of the element which is to be removed
    from the queue.

Returns:

  Success upon success removal; a non-success ReturnCode otherwise.
```

6.8.16 NVQ_FIRST

```
#define NVQ_FIRST(hQueue)                              \
                (nvq_getTail(hQueue) >= 0 ? 0 : NVQ_NO_ELEMENTS)
```

Obtain a handle to the first element of the queue.

```
Parameters:
```

```
  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    for which the element handle is desired.

Returns:

  A handle, of type NVQ_Element, to the first element on the queue.
```

6.8.17 NVQ_LAST

```
#define NVQ_LAST(hQueue)                          \
                (nvq_getTail(hQueue))
```

Obtain a handle to the last element of the queue.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    for which the element handle is desired.

Returns:

  A handle, of type NVQ_Element, to the last element on the queue.
```

6.8.18 NVQ_NEXT

```
#define NVQ_NEXT(hQueue, elem)                            \
                (++elem, elem > nvq_getTail(hQueue) ?    \
                 NVQ_NO_ELEMENTS : elem)
```

Obtain a handle to the next element in the queue, given the current element.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    for which the element handle is desired.

  elem --
    Identifier, of type NVQ_Element, of the element to which the successor
    element is desired.

Returns:

  A handle, of type NVQ_Element, to the next element in the queue.
```

6.8.19 NVQ_PREV

```
#define NVQ_PREV(hQueue, elem)                    \
                (--elem)
```

Obtain a handle to the previous element in the queue, given the current element.

```
Parameters:

  hQueue --
    Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
    for which the element handle is desired.

  elem --
    Identifier, of type NVQ_Element, of the element to which the
    predecessor element is desired.

Returns:

  A handle, of type NVQ_Element, to the previous element in the queue.
```

6.8.20 NVQ_EQUAL

```
#define NVQ_EQUAL(hQueue, elem1, elem2)           \
                (elem1 == elem2)
```

Compare two NVQ_Element handles for equivilency.

```
Parameters:

   hQueue --
     Handle, previously returned by NVQ_open() or NVQ_create(), to the queue
     for which the element handles are being compared

   elem1, elem2 --
     Identifiers, of type NVQ_Element, of the elements to be compared for
     equality.

Returns:

   TRUE if the identifiers reference the same queue element; FALSE
   otherwise.

/*
 * THE REMAINDER IS NOT PART OF THE USER INTERFACE.  DO NOT REFERENCE THESE
 * DIRECTLY IN USER PROGRAMS.  OTHER PORTATIONS OF THIS MODULE MAY NOT CONTAIN
 * THEM.
```

```
 */


typedef enum NVQ_Location
{
    NVQ_Location_Head,
    NVQ_Location_Tail,
    NVQ_Location_Element,
    NVQ_Location_Before,
    NVQ_Location_After
} NVQ_Location;


ReturnCode
nvq_insert(void * hQueue,
           unsigned char * pUserData,
           OS_Uint32 userDataSize,
           NVQ_Location location,
           ...);

ReturnCode
nvq_viewData(void * hQueue,
             unsigned char ** ppData,
             OS_Uint32 * pDataSize,
             NVQ_Location location,
             ...);

ReturnCode
nvq_remove(void * hQueue,
           NVQ_Location location,
           ...);

NVQ_Element
nvq_getTail(void * hQueue);
```

6.9 Exception Handling (EH_)

```
#include "eh.h"
```

This module provides a uniform interface for handling of software exceptions.

EH_ defines the Exception Handling interface. An abstract description of the service to be performed for each of these facilities is described. The specific actions to be performed by each of these facilities may be altered by the integrator depending on the availability of the operating environment facilities.

6.10 Log Module (LOG_)

```
#include "log.h"


Void
LOG_init(void);

SuccFail
LOG_config(Char *tmFileName);

log_ModInfo *
LOG_open(Char *moduleName);

SuccFail
LOG_event(Char *format, ...);

Bool
LOG_modEvent(log_ModInfo *modInfo,  Char *format, ...);
```

The Log Module (LOG_) generates log messages in order to facilitate program monitoring. Within OCP, each module may have its own particular degree of LOG_ tracing enabled at run-time. Developers may also use LOG_ tracing within applications in a similar manner.

Use of the Log Module has several advantages over other traditional tracing methods such as printf() statements:

1. Log messages have a consistent appearance.
2. Log calls need ever be removed from the code.
3. The log module output can be routed to selected devices or files.

Loging may also be globally enabled or disabled at compile time without performing any changes to the source code, other than changing the value of a defined value in a single include file.

6.10.1 Log Module Initialization

```
Void
LOG_init()


SuccFail
LOG_config(char *pFileName)
```

Global initialization of LOG_ is performed by LOG_init(), which should be called once at the start of a user program and prior to any other LOG_ facilites. LOG_Init() creates two queues of log module information - an active queue and a setup queue. All information about LOG_ users is entered in these queues.

Use LOG_config() to redirect LOG_ output to a device or file other than stdout.

6.10.2 LOG_ User Initialization

```
Void
LOG_OPEN(log_ModInfo *hModCB, char *modName)
```

LOG_open() creates a new log module information entry in the active queue. modName is assigned to the entry's module name member. It's bit mask setting is determined by the state of the setup queue. If there exists an entry in the setup queue with the same module name (typically made by a call to LOG_SETUP), then the mask is taken from that queue. Otherwise the mask is set to zero (loging disabled).

LOG_open() returns a pointer to the new entry in the active queue in hModCB.

6.10.3 Log Message Display

```
LOG_event(args)
```

Log messages are generated by the LOG_event() facility, which accepts an argument list string and a variable number of associated arguments.

6.11 Trace Module (TM_)

```
#include "tm.h"

Void
TM_INIT()

QU_Head *
TM_GETHEAD()

SuccFail
TM_CONFIG(char *pFileName)

Void
TM_OPEN(tm_ModInfo *hModCB, char *modName)

Void
TM_TRACE(args)

Bool
TM_QUERY(tm_ModInfo *modInfo, TM_Mask mask)

SuccFail
TM_VALIDATE()

Int
TM_SETUP(char *args)
```

```
LgInt
TM_HEXDUMP(hModCB, bits, msg, pdu, len)

Void
TM_CALL(hModCB, bits, pFunc, hParam1, hParam2)

For backwards compatibility:

Void
TM_init(void);

QU_Head *
TM_getHead(void);

SuccFail
TM_config(Char *tmFileName);

tm_ModInfo *
TM_open(Char *moduleName);

SuccFail
TM_setMask(Char *moduleName, TM_Mask moduleMask);

Bool
tm_trace(tm_ModInfo *modInfo, TM_Mask mask, Char *format, ...);

Bool
TM_query(tm_ModInfo *modInfo, TM_Mask mask);

SuccFail
TM_validate(void);

Int
TM_setUp(Char * str);

LgInt
TM_hexDump(tm_ModInfo *modInfo, TM_Mask mask, String str,
           unsigned char *address, Int length);

Void
TM_call(tm_ModInfo *modInfo,
        TM_Mask mask,
        void (* pfCallMe)(void * hParam1, void * hParam2),
        void * hParam1,
        void * hParam2);
```

The Trace Module (TM_) selectively and dynamically generates trace messages in order to facilitate program debugging. Within OCP, each module may have its own particular degree of TM_ tracing enabled at run-time. Developers may also use TM_ tracing within applications in a similar manner.

Use of the Trace Module has several advantages over other traditional tracing methods such as printf() statements:

1. Trace messages have a consistent appearance.
2. Trace calls need ever be removed from the code.
3. Tracing can be altered at run-time.
4. The trace module output can be routed to selected devices or files.

Tracing may also be globally enabled or disabled at compile time without performing any changes to the source code, other than changing the value of a defined value in a single include file. This allows fully debugged programs to occupy minimal space.

The specific actions actually performed by each of the TM_ facilities may be altered by the integrator depending upon the availability of the operating environment facilities. For instance time stamping may not be available on systems lacking a time-of-day facility.

IMPORTANT NOTE:

The public interface to the Trace Module, as presently defined, consists of a set of all-upper-case macros. The mixed-case function calls are included here for backwards compatibility only and should not be used for new applications.

The macro interface contains hidden `#ifdef TM_ENABLED` preprocessor directives, thereby eliminating the need to bracket each and every reference to TM_ module facilities with conditional compilation directives. This way, for those applications that require minimal memory usage, the Trace Module can be eliminated entirely by undefining TM_ENABLED in oe.h.

6.11.1 Trace Module Initialization

```
Void
TM_INIT()

QU_Head *
TM_GETHEAD()

SuccFail
TM_CONFIG(char *pFileName)

SuccFail
TM_VALIDATE()
```

Global initialization of TM_ is performed by TM_INIT(), which should be called once at the start of a user program and prior to any other TM_ facilites. TM_INIT() creates two queues of trace module information - an active queue and a setup queue. All information about TM_ users is entered in these queues.

Use TM_CONFIG() to redirect TM_ output to a device or file other than stdout.

TM_VALIDATE() searches the setup queue for the names of invalid TM_ user modules and deletes them from the queue.

6.11.2 TM_ User Initialization

```
Int
TM_SETUP(char *args)

Void
TM_OPEN(tm_ModInfo *hModCB, char *modName)
```

Independent module tracing functionalities may be selected by creating tracing modules through the TM_SETUP() and TM_OPEN() facilities. Within each TM_ user module 16 different tracing types may be selected. Each type is specified by a bit within a bit mask. Trace types range from 0x0000 to 0xffff.

The type of trace information to be displayed, is defined by the owner of the module with the exception of trace type 0 (TM_ENTER). Trace type (TM_ENTER) is by convention used for external function entry tracing.

The criteria for displaying the trace information is that the bitwise and (&) result of the trace statement's mask and the dynamic tracing mask associated with that module is non-zero.

TM_SETUP() reads an argument list of the form "MODULE_NAME,BITMASK" and enters the module name and bit mask in either the setup queue or the active queue. If an associated user module has not yet registered itself with TM_ via a call to TM_OPEN(), then a new trace module information entry is made in the setup queue. Otherwise, the information goes in an exisiting entry in the active queue. For example, if args equals `"G_,ffff"` then a queue entry will be made in which the G_ module is assiged the bit mask 0xffff. If the G_ module has not yet registered itself, then this information will go in the setup queue. Otherwise it will go in the active queue.

TM_OPEN() creates a new trace module information entry in the active queue. modName is assigned to the entry's module name member. It's bit mask setting is determined by the state of the setup queue. If there exists an entry in the setup queue with the same module name (typically made by a call to TM_SETUP), then the mask is taken from that queue. Otherwise the mask is set to zero (tracing disabled).

TM_OPEN() returns a pointer to the new entry in the active queue in hModCB.

6.11.3 Trace Message Display

```
TM_TRACE(args)
```

Trace messages are generated by the TM_TRACE() facility, which accepts an argument list string and a variable number of associated arguments.

6.11.4 Example Usage

The following code fragment demonstrates the usage of the Trace Module.

```
#define TM_ENABLED

#include <stdio.h>
#include "estd.h"
#include "tm.h"
#include "getopt.h"

Void G_init(), T_init(), T_action();
```

```
TM_ModDesc G_tmDesc;

main(argc, argv)
Int argc;
String argv[];
{
    Int c;

    G_init();

    while ((c = getopt(argc, argv, "T:t:")) != EOF) {
        switch ( c ) {
        case 'T':
        case 't':
            TM_setUp(optarg);
            break;
        case '?':
        default:
            exit(1);
        }
    }

    TM_trace(G_tmDesc, TM_ENTER, "G_ can be used by orphand modules\n");

    T_init();
    TM_validate();

    T_action();
}

Void G_init()
{
    TM_init();
    G_tmDesc = TM_open("G_");
}

/*
 * T_ MODULE
 */

TM_ModDesc t_tmDesc;

Void T_init()
{
    t_tmDesc = TM_open("T_");
}

Void T_action()
{
    static Char *someData = "SOME DATA";

#ifdef TM_ENABLED
    TM_trace(t_tmDesc, TM_ENTER, "Action, someData=%s\n",
            TM_prAddr(someData));
    TM_dumpHex(t_tmDesc, TM_ENTER, "someData",
               someData, strlen(someData)+1);
```

```
#endif

#ifdef TM_ENABLED
    if (TM_query(t_tmDesc, TM_BIT1)) {
        printf("%s This way we can extend TM_ capabilities\n",
                TM_here());
    }
#endif
}
```

Figure 6.5: TM Example Usage Source Code

6.11.5 Run Time Control of TM_

In the previous example, if you wished to enable trace options for a module, you would specify, on the command line, the following:

-T ¡module_name¿,¡hex_bits¿

where ¡module_name¿ is a valid OCP module, i.e. SCH_, TMR_, etc, and ¡hex_bits¿ is a hexidecimal value (without a leading ”0x”) specifying which trace bits for that module to enabled.

Multiple sets of -T options may be specified to enable tracing for more than one module.

For example:

```
ops_xmpl -T UDP_,ffff -T IMQ_,3
```

General usage dictates that lower-order bits produce less output. The lowest-order bits are often useful even during normal operation of the application. Bits above the first byte are reserved for trace options that provide a lot of output, such as dumping of complete PDUs.

TM_ Output

The actual format of a Trace Module output may vary between implementations. The following example, however, is typical of many hosted systems. Each trace contains the source file name and line number from which the trace was generated, followed by a variable number of user data fields. For example:

```
clinvktd.c, 197: fsm_ePass:         machine=0x60e38  evtId=0x8
```

6.12 SAP management (SAP_)

```
#include "addr.h"
#include "sap.h"

String SAP_selGet(SapSelector *sel, String str);
String SAP_selPr(SapSelector *sel, Char *first, Char *last);
Int SAP_selCmp(SapSelector *sel1, SapSelector sel2);
```

```
String SAP_nAddrGet(N_SapAddr *nsap, String str);
String SAP_nAddrPr(N_SapAddr *nsap, Char *first, Char *last);
Int SAP_nAddrCmp(N_SapAddr *nsap1, N_SapAddr nsap2);
```

SAP_ module is responsible for SAP_ address management. An overview of SAP address representation is first provided.

6.12.1 Representation of a SAP Address

A hierarchical SAP addressing scheme is used. At the Network layer, communicating entities are identified by their NSAP addresses. The Network layer uniquely identifies each of the open systems by their NSAP addresses. Let's take the case of a layer (N) above the Network layer. In this case, an (N) address consists of two parts:

1. an (N-1)-SAP address of the (N) entity which is supporting the current (N)-SAP of the (N+1) entity.

2. an (N)-SAP address selector (suffix) which makes the (N)-SAP uniquely identifiable within the scope of the (N-1)-SAP address.

The (N)-SAP address is a unique representation of a SAP to the OSI environment. The (N)-SAP address selector is a unique representation of a SAP to the service provider. The SAP address selector is communicated to the service provider during the creation of the SAP.

All software layers use a uniform representation for SAP address selectors and SAP addresses. Data abstractions used by software layers to represent SAP addressing are in addr.h.

A SAP address selector in general has the form:

```
typedef struct SapSelector {
    Int len;
    Byte addr[SAPSZ];
} SapSelector;
```

The Network layer SAP address is of particular importance and has the form:

```
typedef struct N_SapAddr {
    Int len;
    Byte addr[NSAPSZ];
} N_SapAddr;
```

Above the network layer, SAP addresses are constructed on top of N_SapAddr. For example, the Presentation SAP address has the form:

```
typedef struct P_SapAddr{
    S_SapSelector ssap;
    T_SapSelector tsap;
    N_SapAddr nsap;
} P_SapAddr;
```

6.12.2 SAP Address Manipulation Facilities

SAP_selGet is facility that converts an ascii representation of a SAP address into a representation suited for SapSelector or SapAddr storage.

SAP_selCmp is a facility that compares two SapSelectors for equality. Zero is returned if the two were identical. A non-zero value is returned if the two were not identical.

6.13 Timer Management Module (TMR_)

```
#include "tmr.h"

void
TMR_init(OS_Uint16 numberOfTimers, OS_Uint16 millisecondsPerTick);

ReturnCode
TMR_start(OS_Uint32 milliseconds,
          void * hUserData1,
          void * hUserData2,
          ReturnCode (* pfHandler)(void * hTimer,
                                   void * hUserData1,
                                   void * hUserData2),
          void ** phTimer);

void *
TMR_create(LgInt milliseconds, Int (*pfHandler)());

void
TMR_stop(void * hTimer);

void
TMR_cancel(void * hTimer);

ReturnCode
TMR_processQueue(OS_Boolean * pProcessedSomething);

void
TMR_poll(void);

void
TMR_startClockInterruptPlus(void (* pfHandler)(void));

SuccFail
TMR_startClockInterrupt(Int period);

void
```

```
TMR_stopClockInterrupt(void);

void
TMR_setLocalDataSize(OS_Uint16 size);

OS_Uint16
TMR_getLocalDataSize(void);

void *
TMR_getData(void * hTimer);

void *
TMR_getDesc(void * pLocalData);

OS_Uint32
TMR_diff(OS_Uint32 time1, OS_Uint32 time2);
```

Many data communication protocols rely on the availability of timer facilities, yet these facilities are inherently environment dependent. The TMR_ module defines a model and an interface for providing timer facilities to Open C Layers, regardless of the environment, provided that all implementations of the TMR_ module conform to the interface defined here.

A timer is created through the TMR_start() or TMR_create() facilities. Unless the timer is canceled through the TMR_stop() or TMR_cancel() facilities it will expire at the specified time. The time for the expiration of a timer is specified in millisecond relative to now. When the timer expires, the TMR_ module schedules a user-supplied function for synchronous invocation. The granularity of timers is environment specific. Timers created through TMR_start() or TMR_create() are not "sticky". If a periodic timer is desired it must be repeatedly re-created.

6.13.1 Starting a Timer

```
ReturnCode
TMR_start(OS_Uint32 milliseconds,
          void * hUserData1,
          void * hUserData2,
          ReturnCode (* pfHandler)(void * hTimer,
                                   void * hUserData1,
                                   void * hUserData2),
          void ** phTimer);

void *
TMR_create(LgInt milliseconds, Int (*pfHandler)());
```

TMR_start() and TMR_create() both start a new timer and arrange, via the scheduler module (SCH_), for the synchronous invocation of the user-supplied function pfHandler when this timer expires.

A limited amount of user data can be associated with each timer. A pointer to the user specific data buffer is passed as a parameter to pfHandler, i.e.

```
(*pfHandler)(tmrData);
```

The timer expiration time is specified in absolute milliseconds relative to present. The time argument must be positive; a negative value or a value of 0 is illegal.

6.13.2 Stopping a Timer

```
void
TMR_stop(void * hTimer);

void
TMR_cancel(void * hTimer);
```

TMR_stop() and TMR_cancel() both halt an active timer and remove it from the timer queue. hTimer is a timer descriptor previously obtained by TMR_create() or TMR_start().

6.13.3 Associating User Data with Timers

```
void *
TMR_getData(void * hTimer);

void *
TMR_getDesc(void * pLocalData);
```

A limited amount of user data can be associated with each timer. The size limit of the user data is implementation specific. Likewise, the semantics of this data are specific to the user and irrelevant to the timer module.

The location of the user data may be obtained by the TMR_getData() facility.

Conversely, given a pointer to some user data, the timer associated with that data can be obtained by the TMR_getDesc() facility.

6.13.4 Implementation Specific Interfaces

```
void
TMR_init(OS_Uint16 numberOfTimers, OS_Uint16 millisecondsPerTick);

void
TMR_startClockInterruptPlus(void (* pfHandler)(void));

SuccFail
TMR_startClockInterrupt(Int period);

void
TMR_stopClockInterrupt(void);

ReturnCode
TMR_processQueue(OS_Boolean * pProcessedSomething);

void
TMR_poll(void);
```

The timing mechanisms that underlie the TMR_ module are implementation-specific. For instance, in unhosted environments, the TMR_ module can be implemented based on a periodic interrupt. On the other hand, in hosted environments, the TMR_ module can be implemented based on operating system-supplied timing mechanisms such as the Unix signal() facility. In either case the interface described below can be used for initialization and integration of the TMR_ module.

- Initialization of the TMR_ module is performed through TMR_init(), which must be invoked prior to using any of the other timer module facilities. numberOfTimers specifies the maximum number of timers that may be in use at any one moment. millisecondsPerTick specifies how often a clock ”tick” is generated.
- TMR_startClockInterruptPlus() and TMR_startClockInterrupt() both start the timer module clock tick. On unhosted systems this has the effect of enabling the TMR_ module clock interrupt source. TMR_startClockInterruptPlus() allows the user to specify the routine that is called when a clock tick occurs but uses a built-in number of milliseconds per tick. TMR_startClockInterrupt() uses a predefined service routine but allows the user to specify the number of milliseconds per tick.
- TMR_stopClockInterrupt() disables the clock tick. On unhosted systems this has the effect of disabling the TMR_ module interrupt source.
- TMR_poll() or TMR_processQueue() must regularly be called to process expired timers. TMR_processQueue() provides additional information about the status of its completion via the Boolean parameter pProcessedSomething. In both hosted and unhosted systems, TMR_poll() or TMR_processQueue() are scheduled for execution by the clock tick service routine via the OCP scheduler (SCH_) facility SCH_submit(). It is during the execution of these facilities that the user functions themselves are invoked.
- TMR_clock() is the entry point to the clock tick handler routine.

6.13.5 Example Usage

The following code fragment demonstrates the use of the timer facilities.

```
/*
 * tmr_ex.c
 *
 * This simple program demonstrates the use of timer facilities
 * for a relative measurement of CPU speed.
 */


#include <stdio.h>
#include <signal.h>

#ifdef MSDOS
#include <conio.h>
#endif

#include "estd.h"
#include "eh.h"
#include "tmr.h"
#include "getopt.h"
#include "tm.h"
```

```
#include "sch.h"

#define CLOCK_PERIOD 1100 /* millisec. Granularity of the timer tick is 54.94msec
                             for DOS */
#define SAMPLE_PERIOD 2000 /* millisec. Sampling Period */
#define LOAD_FACTOR 10 /* Simulated Load */

typedef enum TmrId {
    PURE,
    IDLE
} TmrId;

typedef struct G_Env {
    Char *progName;
    /* Application Specific Information */
} G_Env;

#ifdef MSDOS
int G_isrActive;
#endif

LgUns pureCpu;          /* Measure of "CPU cycles" in one Sample */
LgUns idleCpu;          /* Measure of "CPU cycles" in one Sample, under load */
Bool reset = FALSE;
LgUns samplePeriod = SAMPLE_PERIOD;
Int loadFactor = LOAD_FACTOR;
Int clockPeriod = CLOCK_PERIOD;
Bool pureSampling;
PUBLIC G_Env G_env;

Void samplePureCpu();
Void IDLE_tmrHandler(void*);
Void IDLE_init(void);
Void IDLE_action(void*);
Void LOAD_init(int), LOAD_action(void*);
Void G_usage(void);
Void G_exit();



/*<
 * Function:
 *
 * Description:
 *
 * Arguments:
 *
 * Returns:
 *
>*/
int
main(int argc, char **argv)
{
    Int c;
    Bool badUsage;

    G_env.progName = argv[0];
```

```
TM_init();

/*
 * Process the command line arguments.
 */
badUsage = FALSE;
while ((c = getopt(argc, argv, "T:s:l:h:c:")) != EOF) {
    switch (c) {
    case 'T':
        TM_setUp(optarg);
        break;

    case 's':
        samplePeriod = atol(optarg);
        break;

    case 'l':
        loadFactor = atoi(optarg);
        break;

    case 'c':
        clockPeriod = atoi(optarg);
        break;

    case 'h':
    default:
        badUsage = TRUE;
        break;
    }
}

if (badUsage) {
    G_usage();
    G_exit(1);
}

printf("\nclock period = %d milliseconds", clockPeriod);
printf("\nsample period = %d milliseconds", (int)samplePeriod);
printf("\nload factor = %d\n", loadFactor);

/*
 * OCP Intialization.
 */
signal(SIGINT, G_exit); /* ctrl-c handler */
SCH_init(22);
TMR_init(22, clockPeriod);
TMR_startClockInterrupt(clockPeriod);
TM_validate();

/*
 * Step 1.
 */
samplePureCpu();

/*
 * Step 2.
 */
```

```
    IDLE_init();
    LOAD_init(loadFactor);

#ifdef MSDOS
    while (!kbhit()) {
#else
     while (1) {
#endif
        SCH_block();
        SCH_run();
    }

    G_exit(0);
    return 0;
}


/*<
 * Function:    G_exit
 *
>*/

Void
G_exit(Int code)
{
  TMR_stopClockInterrupt();
  exit(code);
}

/*<
 * Function:    G_usage
 *
>*/

Void
G_usage(void)
{
    String usage1 = "[-T module_name,trace_mask] [-s sample_period] [-l load_factor] [-c clock_period]";
    printf("\n%s: Usage: %s\n", G_env.progName, usage1);
}


/*<
 * Function: samplePureCpu
 *
 * Description:
 *
 * Count the number of passes made through the scheduler in one sample
 * period with nothing else happening. This is the "pure CPU" measure.
 *
 *   while (pureSampling) {
 *      ++pureCpu;
 *      SCH_block();
 *      SCH_run();
 *   }
 *
 *
```

```
>*/

Void samplePureCpu()
{
    TMR_Desc tmrDesc;
    TmrId *tmrId;
    static int n;

    /*
     * Create a timer which expires after 'samplePeriod' milliseconds
     * and which then invokes (via the scheduler) the function 'IDLE_tmrHandler'
     */
    tmrDesc = TMR_create((LgInt)samplePeriod,(void*)IDLE_tmrHandler);
    tmrId = (TmrId *) TMR_getData(tmrDesc);
    *tmrId = PURE;
    pureSampling = TRUE;
    pureCpu = 0;

    while (pureSampling) {
      /* do this until the timer expires */
      ++pureCpu;
      SCH_block();
      SCH_run();        /* expired timers are invoked here */
    }

    /* Make a note of the value of pureCpu */
    printf("%d: pureCpu = %ld\n", n++, pureCpu);
}


/*<
 * Function: IDLE_tmrHandler
 *
 * Description:
 *
 * Repetitively calculate and display the ratio of the idleCpu counter to
 * the pureCpu counter. As loadFactor is made larger this ratio will become
 * smaller.
 *
>*/

Void
IDLE_tmrHandler(void *tmrData)
{
    TMR_Desc tmrDesc;
    TmrId *tmrId;
    static int n;

#ifdef MSDOS
    if (kbhit())
        G_exit(0);
#endif

    tmrId = (TmrId *)tmrData;

    /* Did we get here as the result of samplePureCpu() ?
     * If so, just set the flag and exit.
```

```
     */
    if (*tmrId == PURE) {
        pureSampling = FALSE;
        n = 0;

    /* we must have gotten here as the result of IDLE_init(), so
     * perform the calculation
     */
    } else if (*tmrId == IDLE) {
        if ( idleCpu != 0 )
            printf("%d: idleCpu=%ld, idleCpu/pureCpu=%f\n", n++, idleCpu,
                   (float)idleCpu/(float)pureCpu);
        else
            printf("Oops! idleCpu = %ld\n", idleCpu);

        idleCpu = 0;
        tmrDesc = TMR_create((LgInt)samplePeriod, (void*)IDLE_tmrHandler);
        tmrId = (TmrId *) TMR_getData(tmrDesc);
        *tmrId = IDLE;
    } else {
        EH_oops();
    }
}


/*<
 * Function: IDLE_init
 *
 * Description:
 *
 * Schedule the invocation of IDLE_tmrHandler() after one sample period.
 * Schedule the immediate invocation IDLE_action().
 *
>*/

Void
IDLE_init(void)
{
    TMR_Desc tmrDesc;
    TmrId *tmrId;

    idleCpu = 0;
    tmrDesc = TMR_create((LgInt)samplePeriod, (void*)IDLE_tmrHandler);
    tmrId = (TmrId *) TMR_getData(tmrDesc);
    *tmrId = IDLE;
    SCH_submit((void*)IDLE_action, (Ptr)0,0,"in IDLE_init()");
}

/*<
 * Function: IDLE_action
 *
 * Description:
 *
 * This function increments the idleCpu counter and then reschedules itself.
 * In effect, this is as infinite loop but performed through the scheduler,
 * allowing other concurrent action to take place.
 *
```

```
 * As the CPU becomes more heavily loaded with other scheduled activity,
 * this function will be invoked less frequently.
>*/

Void
IDLE_action(void *argunused)
{
    ++idleCpu;
    SCH_submit((void*)IDLE_action, (Ptr)0,0,"in IDLE_action()");
}


/*<
 * Function: LOAD_init
 *
 * Description:
 *
 * Schedule the immediate invocation of LOAD_action().
 *
>*/

Void
LOAD_init(Int load)
{
    loadFactor = load;
    SCH_submit((void*)LOAD_action, (Ptr)0,0,"in LOAD_init()");
}

/*<
 * Function: LOAD_action
 *
 * Description:
 *
 * This function hogs some CPU time and then reschedules itself.
 * Thus, the larger loadFactor, the fewer CPU cycles there are available
 * for other activity.
 *
>*/

Void
LOAD_action(void *argsunused)
{
    Int i;

    /* Hog some time */
    for (i=0; i < loadFactor; ++i) {
    }

    /* Schedule to do it again */
    SCH_submit((void*)LOAD_action, (Ptr)0,0,"in LOAD_action()");
}
```

```
/* Call Graph for the example program */
```

Figure 6.6: TMR Module Example Usage Source Code

6.14 Data Unit Management (DU_)

```
#include "du.h"

DU_PoolDesc  DU_poolCreate(bufSize, nuOfBufs, nuOfViews)
Int bufSize, nuOfBufs, nuOfViews;

DU_View DU_alloc(DU_PoolDesc pool, Int nuOfBytes);

DU_View DU_link(DU_View view);

Void DU_free(DU_View view);

Void DU_prepend(DU_View view, Int nuOfBytes);
Void DU_adjust(DU_View view, Int nuOfBytes);
Void DU_strip(DU_View view, Int nuOfBytes);
Ptr DU_data(DU_View view);
Int DU_size(DU_View view);

Ptr DU_vToUinfo(DU_View view);
DU_View DU_uInfoToV(Ptr uInfo);
```

DU_ module provides an environment independent interface to a set of facilities providing data unit management capabilities. Propagation of protocol data through the lower layers is accomplished through the DU_ module. Data is not copied as it propagates through Open C Layers. By minimizing memory-to-memory block transfers higher performance is achieved.

When propagating down the layers, initially a data unit is allocated at the top layer. This data unit is stored in a buffer that is large enough to contain all protocol control information that any of the lower layers may prepend to it. (N+1) Open C Layer passes a "view" of the data unit as an (N)-SDU to (N) Open C Layer. (N) Open C Layer appends its (N)-PCI to the view and delivers its (N)-PDU to (N-1) Open C Layer. This is repeated until the data unit is reached to the bottom layer.

When propagating up the layers, initially the bottom layer allocates the data unit. (N-1) Open C Layer delivers a view of the data unit as an (N-1)-SDU to (N) Open C Layer. (N) layer strips its PCI from the (N)-PDU and delivers the resulting (N)-SDU to (N+1) layer. This is repeated until the data unit is reached to the top layer.

The Data Unit management module is a set of facilities and data abstractions that can create, manipulate, and delete data units. DU_ facilities may be implemented based on SF_mem facilities described in the previews sections. Description of DU_ services in this section is based around such an implementation. This concrete example implementation eases the understanding of the DU_ module interface description.

Four basic data structures are central to DU_ module:

struct du_PoolInfo *

du_PoolInfo *

DU_PoolDesc Is a reference to du_PoolInfo which contains information about a pool of du_BufInfos. The maximum size of buffers in the pool is fixed. Each pool maintains a list of available du_BufInfos.

struct du_BufInfo

du_BufInfo Contains the space for the largest potential data unit of the pool that it belongs to. du_BufInfo is an abstraction private to the DU_ module. It is not expected to be operated upon by the DU_ user. It is described here, to ease the understanding of the operation of the DU_ module.

struct DU_Elem *

DU_Elem *

DU_View The central abstraction in DU_ module is DU_View. DU_Elem contains specific information that can provide a view over a particular du_BufInfo.

```
typedef struct DU\_Elem *DU\_View;
```

provides a convenient abstraction over this commonly manipulated data structure. One field in struct DU_Elem is a pointer to *du_BufInfo* . Each DU_Elem contains an area of memory reserved for the user. The size of this user information area is implementation specific. Each DU_Elem is expected to allow for a QU_elem to be casted over it.

Byte uInfo[ ] Each DU_Elem contains an area of memory reserved for its user. The semantics of the data that may be stored in this area is specific to the user and irrelevant to the DU_ module.

An allocated DU_View and a free DU_View are illustrated in the following two figures.

DU_poolCreate is expected to be invoked during initialization of the system. DU_poolCreate creates a data unit pool and returns a pool descriptor. All Open C Layers may rely on the existence of a buffer pool called G_duMainPool. The external declaration of G_duMainPoool is expected to be in g.h. Specified number of du_BufInfo and DU_Views are reserved. Several buffer pools with potentially different buffer sizes may coexists in Open C Environment.

DU_alloc involves the allocation of one *du_BufInfo* from the specified pool and the allocation of one DU_Elem. DU_Elem is then associated with du_BufInfo. The DU_Elem is a specific view private to the caller. Each DU_Elem is expected to allow for a QU_elem to be casted over it. DU_alloc performs a *QU_init* to the allocated *DU_View* . The amount of data specified for allocation, must be smaller than the maximum buffer size associated with the specified pool. The remaining space (maximum buffer size minus the requested buffer size) can be accessed through the DU_prepend facility.

The following figure illustrates the status of DU_ structures after a DU_alloc.

DU_link provides an additional view over an existing *du_BufInfo* . The DU_Link facility enables the user to maintain more than one view of the same buffer.

DU_free deallocates the specified *DU_View* . If no other views of the buffer exists then the du_BufInfo associated with DU_View is also deallocated. du_BufInfo is preserved for as long as there is at least one DU_View referring to it. The number of references to each du_BufInfo is maintained in refCount field. Upon allocation this refCount is set to 1. Each DU_link increments refCount by 1. Each DU_free decrements refCount by 1.

The following figure demonstrates the status of DU_ structures after DU_free invocation.

The external interface to the buffer management module is described below.

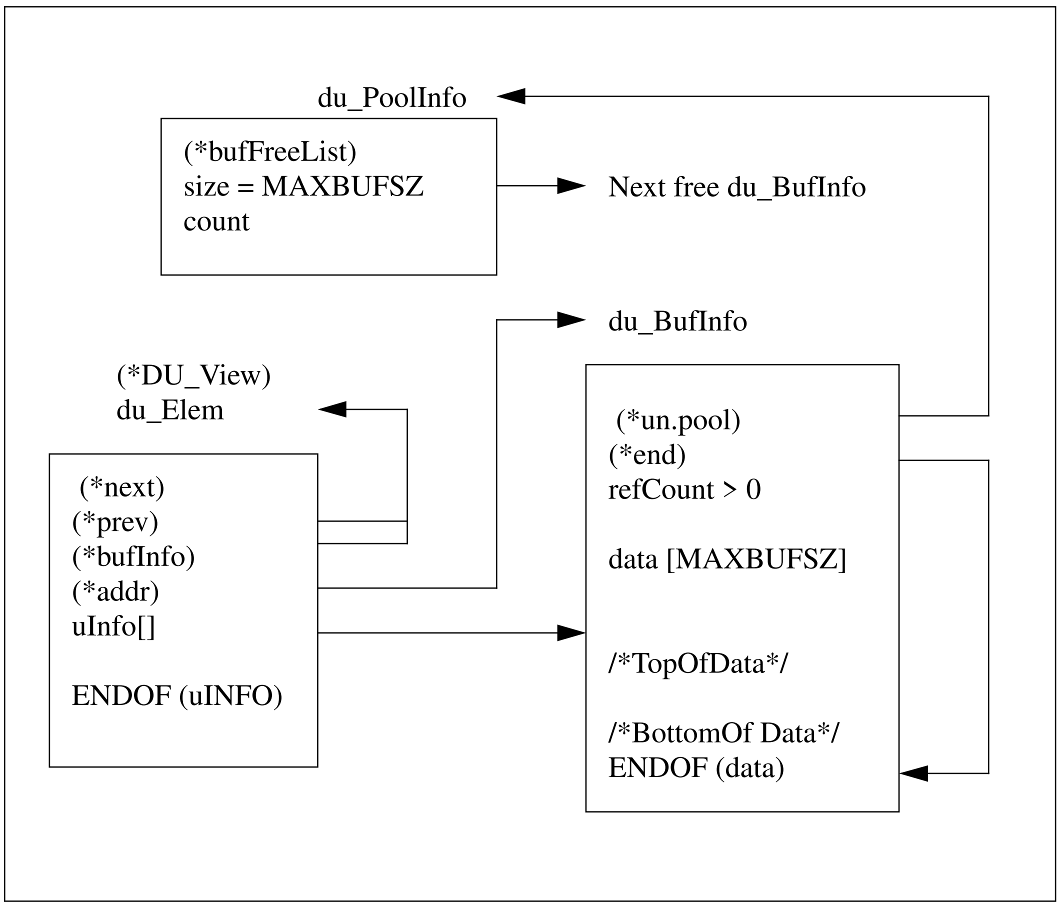

Figure 6.7: An Allocated View

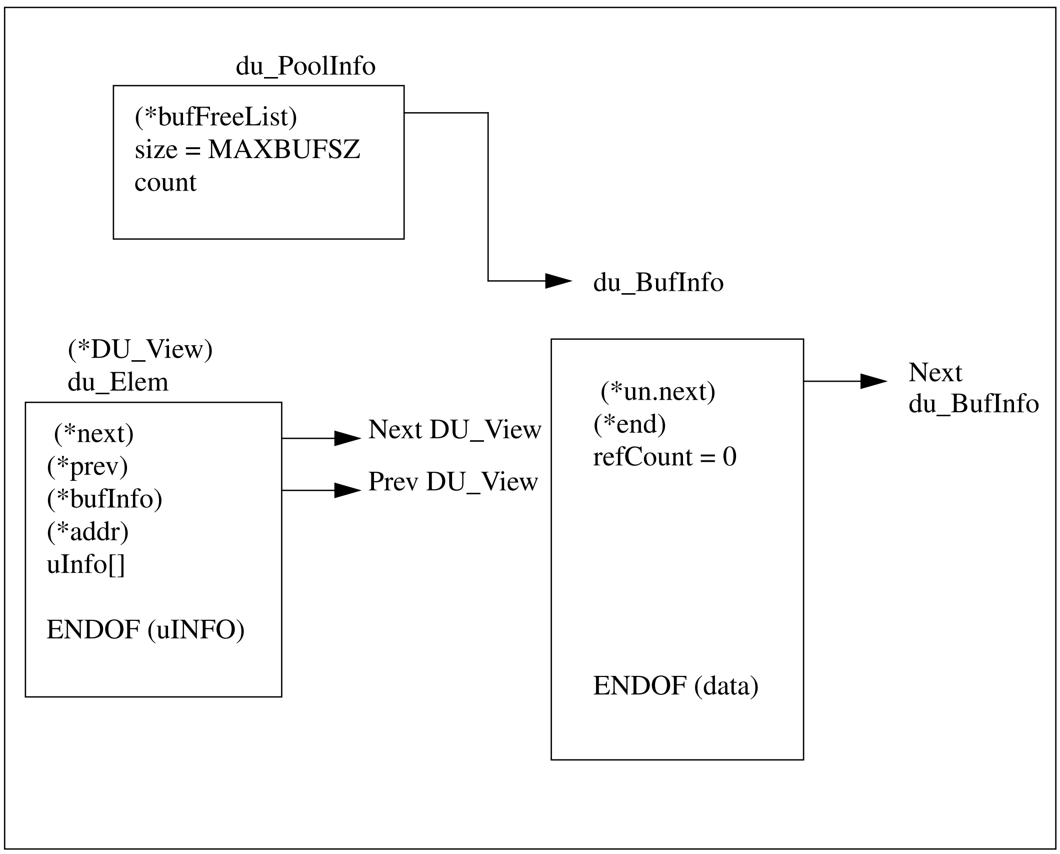

Figure 6.8: A Free View

6.14.1 Creation of a Pool

```
DU_PoolDesc DU_poolCreate(bufSize, nuOfBufs, nuOfViews)
Int int bufSize, nuOfBufs, nuOfViews;
```

A buffer pool descriptor (DU_PoolDesc) must be obtained before any views that are associated with it can be allocated. The return value of this function must be used in future usage of DU_ facilities referring to this pool. bufSize specifies the maximum size of the buffers that can be allocated from this pool. nuOfBufs specifies how many du_BufInfos will be available for allocation. nuOfBufs specifies how many DU_Views will be available for allocation or linkage.

Open C Layers can assume the existence of one buffer pool named G_duMainPool. Therefore:

```
G_duMainPool = DU_poolCreate(bufSize, nuOfBufs, nuOfViews);
```

is expected of G_ module.

6.14.2 Creation of a Data Unit

```
DU_View DU_alloc(DU_PoolDesc  pool, int nuOfBytes);
```

DU_alloc, allocates a new du_BufInfo structure from the specified pool pool. pool must have been obtained through *DU_poolCreate* . A DU_Elem is allocated and initialized according to the specified nuOfBytes. A reference to DU_Elem is returned to the user as a DU_View. NULL is returned if no du_BufInfo were available or if no DU_View were available or if the nuOfBytes argument was larger than the maximum buffer size associated with pool. du_BufInfo.refCount is set to 1.

6.14.3 Creating a new View

```
DU_View DU_link(DU_View view);
```

DU_link provides an additional view over an existing du_BufInfo. view must have been obtained through DU_alloc. NULL is returned if there are no DU_Views remaining. du_BufInfo.refCount is incremented.

6.14.4 Freeing a View

```
Void DU_free(DU_View view);
```

DU_free deallocates view. view must have been obtained through DU_alloc or DU_link. du_BufInfo.refCount is decremented. If refCount becomes zero, the du_BufInfo is no longer in use and du_BufInfo is returned to the buffer pool from which it was originally allocated.

6.14.5 Manipulating a View's Data

```
Void DU_prepend(DU_View view, Int nuOfBytes);
Void DU_adjust(DU_View view, Int nuOfBytes);
Void DU_strip(DU_View view, Int nuOfBytes);
Ptr DU_data(DU_View view);
Int DU_size(DU_View view);
```

DU_prepend facility prepends bytes at the beginning of the specified data unit. It is often used to create space for the Protocol Control Information (PCI) to be prepended by Open C Layers. It is the responsibility of the user to to assure the availability of space. DU_prepend does not do any range checking for excessive prepends. The appended bytes are not initialized to any specific value.

DU_strip facility strips (removes) nuOfBytes bytes from the beginning of the view. It is often used to remove protocol control information from PDUs as they propagate up the layers.

DU_adjust facility adjusts (moves) the end of buffer pointer by nuOfBytes bytes. The effect of DU_adjust is not private to view. All views of DU_Elem.bufInfo are affected.

DU_data facility returns a pointer to the first byte of data as viewed through the specified view. If view is NULL, NULL is returned.

DU_size facility returns the current length of the buffer as observed by the specified view. If view is a NULL pointer, zero is returned.

6.14.6 Accessing a View's User Information

```
Ptr DU_vToUinfo(DU_View view);

DU_View DU_uInfoToV(Ptr uInfo);
```

In order to access the user specific information associated with a view two facilities are provided. Given a view, *DU_vToUinfo* delivers a pointer to the area of memory that can be used by the user. Given a pointer to an area of memory that was previously obtained through DU_vToUinfo, *DU_uInfoToV* can be used to obtain the DU_View associated with the *uInfo* .

6.14.7 Example Usage

The following code fragment demonstrates the use of buffer management facilities.

```
/*
 * This example program demonstrates the propagation of a
 * Data Unit through Open C Layers;
 * UPPER_ N_ and LOWER_
 */

#include "estd.h"
#include "eh.h"
#include "du.h"

DU_PoolDesc *G_duMainPool; /* Buffer Pool */
#define MAXBUFSIZE 1548 /* no special significance */
#define NUOFBUFS 22 /* no special significance */
#define NUOFVIEWS (2*NUOFBUFS) /* no special significance */

main()
{
    static char n_sdu[] = "(N)-SDU";
    DU_View view;
```

```
    G_duMainPool = DU_poolCreate(MAXBUFSIZE, NUOFBUFS, NUOFVIEWS);
    if ( ! G_duMainPool ) {
        EH_fatal("G_duMainPool");
    }
    UPPER_init(&n_sdu[0], (Int) sizeof(n_sdu));
}

UPPER_init(data, size)
Char *data;
Int size;
{
    /* Allocate a Data Unit */
    view = DU_alloc(G_duMainPool, size);
    if ( ! view ) {
        EH_fatal("view");
    }
    BS_memCopy(data, DU_data(view), size);

    /* deliver n_sdu to (N) layer */
    N_dataReq(view);
}

N_dataReq(view)
DU_View view;
{
    static char n_pci[] = "(N)-PCI ";
#define NPCI_SIZE (sizeof(n_pci)-1) /* Don't want the '\0' */

    /* Add (N) layer Protocol Control Information
     * and pass it down to layer below.
     */
    DU_prepend(view, NPCI_SIZE);
    BS_memCopy(n_pci, DU_data(view), NPCI_SIZE);

    LOWER_dataReq(view);
}

LOWER_dataReq(view)
DU_View view;
{
    printf("(N-1)-SDU = %s\n", (char *) DU_data(view));
    DU_free(view);
}
```

Figure 6.9: DU Module Example Usage Source Code

Upon execution this example program produces:

```
(N-1)-SDU = (N)-PCI + (N)-SDU
```

The flow of this example program is:

```
DU_ Module Example Usage Call Graph
```

6.15 Buffer (BUF)

```
ReturnCode
BUF_alloc(OS_Uint16 minSize, void ** phBuf)

void
BUF_free(void * hBuf)

ReturnCode
BUF_addOctet(void * hBuf, OS_Uint8 octet)

ReturnCode
BUF_getOctet(void * hBuf, OS_Uint8 * pOctet)

ReturnCode
BUF_ungetOctet(void * hBuf)

ReturnCode
BUF_prependChunk(void * hBuf,
                 STR_String string)

ReturnCode
BUF_prependBuffer(void * hBuf,
                  void * hPrependThisBuf);

ReturnCode
BUF_getChunk(void * hBuf,
             OS_Uint16 * pChunkLength,
             unsigned char ** ppData)

ReturnCode
BUF_appendChunk(void * hBuf,
                STR_String string)

ReturnCode
BUF_appendBuffer(void * hBuf,
                 void * hAppendThisBuf)

void
```

```
BUF_resetParse(void * hBuf)

ReturnCode
BUF_copy(void * hBufSrc,
               void ** phBufDest);

ReturnCode
BUF_cloneBufferPortion(void * hBuf,
                       OS_Uint32 len,
                       OS_Boolean bStripClonedPortion,
                       void ** phNewBuf)

OS_Uint32
BUF_getBufferLength(void * hBuf)

void
BUF_dump(void * hBuf, char * pMsg)
```

6.15.1 Allocate and Free a Buffer

```
ReturnCode
BUF_alloc(OS_Uint16 minSize, void ** phBuf);
```

Allocate a new buffer. The buffer will contain, initially, one segment which is large enough to hold (at least) the specified number of octets.

If the minimum size is not known, zero may be passed, and a default zero-size buffer segment will be allocated, initially.

```
Parameters:
        minSize --
                Minimum size of for the data area within the
                buffer segment.  A buffer segment will be
                provided (if possible) which has at least this
                much space available.

        phBuf --
                Pointer to memory where a buffer handle is to be
                placed.

Returns:

        Success or ResourceError or one of the BUF_RC_* return
        codes.

void
BUF_free(void * hBuf);
```

Free the specified buffer and all of its associated segments.

```
Parameters:

        hBuf -- Handle to a buffer, previously returned by
                BUF_alloc(), which is to be freed.

Returns:
        Nothing.
```

6.15.2 Add, Get, and Unget an Octet

```
ReturnCode
BUF_addOctet(void * hBuf, OS_Uint8 octet);
```

Prepend a single octet to a buffer. If there is insufficient space in the current segment, a new segment is allocated, of the default size.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                BUF_alloc(), in which the octet is to be
                prepended.
        octet --
                The octet value to be prepended to the buffer.

Returns:
        Success upon success;
        Fail if allocating a new segment failed.
```

```
ReturnCode
BUF_getOctet(void * hBuf, OS_Uint8 * pOctet);
```

Get the next octet in the buffer.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                Buf_alloc(), from which the octet is to be
                obtained.

        pOctet --
                Pointer to a memory location in which the
                retrieved octet is to be placed.

Returns:
        Success if an octet was available; Fail otherwise.
```

```
ReturnCode
BUF_ungetOctet(void * hBuf);
```

Return the most recently retrieved octet to the input buffer stream.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                Buf_alloc(), from which an octet is to be
                returned.

Returns:
        Nothing.
```

6.15.3 Allocate and Assign a New Buffer Segment

```
ReturnCode
BUF_prependChunk(void * hBuf,
                 STR_String string);
```

This function allocates a new buffer segment, and assigns the specified string to that segment. The internal buffer pointers are left in such a state as to allow prepending additional octets. Any additional octets prepended will cause a new buffer segment to be created, as the segment for this prepended chunk takes up its own whole segment.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                BUF_alloc(), in which space is being requested.

        string --
                A string handle, which is to be prepended to the
                buffer.

Returns:

        On error, ResourceError is returned.
```

6.15.4 Prepend a Buffer

```
ReturnCode
BUF_prependBuffer(void * hBuf,
                  void * hPrependThisBuf);
```

This function prepends one buffer to another. The internal buffer pointers are left in such a state as to allow prepending additional octets.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                BUF_alloc(), in which space is being requested.

        hPrependThisBuf --
                Handle to the buffer, previous returned by
                BUF_alloc(), which is to be prepended to hBuf.

Returns:

        Currently, this function always returns Success.
```

6.15.5 Get the Next Chunk of a PDU

```
ReturnCode
BUF_getChunk(void * hBuf,
             OS_Uint16 * pChunkLength,
             unsigned char ** ppData);
```

When parsing, this function returns a pointer to the next chunk of the PDU. The size of the chunk is determined by the value of *pChunkLength when this function is called, and by the amount of data remaining in the current (or first non-zero-length) segment. A chunk of no more then the requested chunk length will be provided.

```
Parameters:
        hBuf --
                Handle to a buffer, previously returned by
                BUF_alloc(), from which a chunk of data is being
                requested.

        pChunkLength --
                Pointer to memory containing the number of octets
                being requested in the chunk.  This value may be
                zero, to indicate that a pointer to as many
                octets as possible should be returned.

        ppData --
                A pointer to a location in which a pointer to the
                data in the chunk is placed.  Also, the value
                pointed to by pChunkLength is updated to contain
                the length of data being provided.  This value
                may equal the requested chunk length, or may be
                less then that length.

Returns:
```

```
        Success or ResourceError or one of the BUF_RC_* return
        codes.
```

6.15.6 Append a String to a Buffer

```
ReturnCode
BUF_appendChunk(void * hBuf,
                STR_String string);
```

Append a string to the end of a buffer. This function is primarily for use when receiving data from the network, which is to later be parsed.

```
Parameters:
        hBuf --
                Handle to a buffer, previously returned by
                BUF_alloc(), to which a chunk of data is to be
                added.

        string --
                String to be appended to the buffer.

Returns:
        Success or ResourceError.
```

6.15.7 Append One Buffer to Another

```
ReturnCode
BUF_appendBuffer(void * hBuf,
                 void * hAppendThisBuf);
```

This function appends one buffer to another. The internal buffer pointers are left in such a state as to allow appending additional octets.

```
Parameters:
        hBuf --
                Handle to the buffer, previously returned by
                BUF_alloc(), in which space is being requested.

        hAppendThisBuf --
                Handle to the buffer, previous returned by
                BUF_alloc(), which is to be appended to hBuf.

Returns:

        Currently, this function always returns Success.
```

6.15.8 Reset Buffer Pointers

```
void
BUF_resetParse(void * hBuf);
```

Reset the internal buffer pointers for another parse of the buffer.

```
Parameters:

        hBuf --
                Handle to a buffer, previously returned by
                BUF_alloc(), which is to be freed.

Returns:
        Nothing.
```

6.15.9 Copy a Buffer

```
ReturnCode
BUF_copy(void * hBufSrc,
               void ** phBufDest);
```

Copy an entire buffer. All data is copied, so string data is independent of the source buffer (as opposed to the way BUF_cloneBufferPortion() works).

```
Parameters:

     hBufSrc --
            Handle to a buffer, previously returned by
            BUF_alloc(), which contains the data to be copied

     phBufDest --
            Pointer to a buffer handle.  A new buffer is allocated
            by this function, the data from hBufSrc is copied to
            it, and the location pointed to by this parameter is
            set to be the new buffer handle.

Returns:
     Success or ResourceError
```

6.15.10 Clone a Portion of a Buffer

```
ReturnCode
BUF_cloneBufferPortion(void * hBuf,
                       OS_Uint32 len,
                       OS_Boolean bStripClonedPortion,
                       void ** phNewBuf);
```

Clone a portion of a buffer. A new buffer handle is provided, that contains a (possibly) partial list of the segments from the cloned buffer. The new buffer and original buffer may each be freed or manipulated independently, with the caveat that the String Data pointed to by the segments is the same in both buffers. Any modifications to the data within the cloned buffer that is in the common portion to the original buffer will be reflected in both buffers.

```
Parameters:
        hBuf --
                Handle to a buffer, previously returned by
                BUF_alloc(), a portion of which is to be cloned.

        len --
                Length of data, beginning at the current location
                within the buffer indicated by hBuf, which is to
                be cloned.  If len is BUF_REMAINDER, the buffer
                portion beginning at the current buffer pointer
                and ending at the end of the buffer is cloned.

        bStripClonedPortion --
                If TRUE, update the start of the data in the buffer
                to be just beyond the cloned portion.

        phNewBuf --
                Pointer to location to put the handle of the new
                cloned buffer.

Returns:
        Success or ResourceError.
```

6.15.11 Get the Length of a Buffer

```
OS_Uint32
BUF_getBufferLength(void * hBuf);
```

Determine the length of the buffer, by adding the lengths of the data strings of each of the buffer segments.

```
Parameters:
        hBuf --
                Handle to a buffer, previously returned by
                BUF_alloc(), a portion of which is to be cloned.

Returns:
        The length of the buffer.
```

6.15.12 Display a Buffer

```
void
BUF_dump(void * hBuf, char * pMsg);
```

Display, on STDOUT, the entire contents of the buffer.

6.16 Configuration Module (CFG)

```
#include ``config.h''

ReturnCode
CONFIG_open(char * pFileName,
            void ** phConfig);

void
CONFIG_close(void * hConfig);

ReturnCode
CONFIG_nextSection(void * hConfig,
                   char ** ppSectionName,
                   void ** phSection);

ReturnCode
CONFIG_nextParameter(void * hConfig,
                     char * pSectionName,
                     char ** ppTypeName,
                     char ** ppValue,
                     void ** phParameter);

ReturnCode
CONFIG_getNumber(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 OS_Uint32 * pValue);


ReturnCode
CONFIG_getString(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 char ** ppValue);

ReturnCode
CONFIG_setNumber(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 OS_Uint32 value,
                 CONFIG_Permanence permanence);
```

```
ReturnCode
CONFIG_setString(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 char * pValue,
                 CONFIG_Permanance permanance);
```

The Configuration Module (CFG) provides a set of functions for reading and writing configuration data. The data can be maintained in a file, or can be set on a temporary basis for use during an iteration of the program, or until the data is re-read from the file.

A CFG configuration file contains one or more *Sections*, each of which contain one or more *Parameters*. Parameters are pairs of *Types* and *Values*.

```
section a
   parameterx.type parameterx.value
   parametery.type parametery.value
                  .
                  .
                  .
section b
   parameteri.type parameteri.value
   parameteri.type parameteri.value
                  .
                  .
                  .
```

For examples of configuration file formats for various environments see Chapter 8, Implementations of the Platform.

6.16.1 Open and Close a Configuration File

```
ReturnCode
CONFIG_open(char * pFileName,
            void ** phConfig);

void
CONFIG_close(void * hConfig);
```

CONFIG_open opens the configuration file specified by pFileName and places a file handle in the location pointed to by phConfig. CONFIG_close closes the previously opened configuration file with the handle hConfig.

CONFIG_open has the following possible return values:

- Success
- ResourceError Out of memory
- UnsupportedOption Unsupported/unrecognized configuration data
- OS_RC_NoSuchFile Couldn't open the configuration file

6.16.2 Get a Section

```
ReturnCode
CONFIG_nextSection(void * hConfig,
                   char ** ppSectionName,
                   void ** phSection);
Returns:
        Success
        Fail
```

CONFIG_nextSection scans the configuration file pointed to by hConfig for the next Section and returns a pointer to the character string containing the section name in ppSectionName.

phSectionis a Handle indicating where to begin the search. If the location pointed to by this parameter contains NULL (i.e. *phSection == NULL) then the first section in the configuration file will be returned. Upon return from this function, *phSection is updated with a new value. If this value of phSection is passed to this function again (with the same section name), the next section in the configuration file will be returned.

6.16.3 Get a Parameter

```
ReturnCode
CONFIG_nextParameter(void * hConfig,
                     char * pSectionName,
                     char ** ppTypeName,
                     char ** ppValue,
                     void ** phParameter);
Returns:
        Success
        Fail
```

CONFIG_nextParameter scans the configuration file pointed to by hConfig, under the section pSectionName for the next occurrence of a Parameter. It then places a pointer to the string containing the Parameter's Type in ppTypeName and a pointer to the string containing the Parameter's Value in ppValue.

phParameter is a Handle indicating where to begin the search. If the location pointed to by this parameter contains NULL (i.e. *phParameter == NULL) then the first Parameter in the specified Section will be returned. Upon return from this function, *phParameter is updated with a new value. If this value of phParameter is passed to this function again (with the same section name), the next parameter in the specified Section will be returned.

6.16.4 Operate on Parameter Values

```
#define CONFIG_MAX_PARAMETER_LEN        1024

ReturnCode
CONFIG_getNumber(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 OS_Uint32 * pValue);
```

```
ReturnCode
CONFIG_getString(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 char ** ppValue);
ReturnCode
CONFIG_setNumber(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 OS_Uint32 value,
                 CONFIG_Permanance permanance);

ReturnCode
CONFIG_setString(void * hConfig,
                 char * pSectionName,
                 char * pTypeName,
                 char * pValue,
                 CONFIG_Permanance permanance);

Returns:
        Success
        Fail
```

The facilities CONFIG_getNumber, CONFIG_getString, CONFIG_setNumber, and CONFIG_setString operate on Paramter Values.

CONFIG_getNumber and CONFIG_getString read a configuration file and return an unsigned 32 bit integer or a pointer to a string, respectively.

CONFIG_setNumber and CONFIG_setString take an unsigned 32 bit integer or a pointer to a string, respectively, as input and write their values to a configuration file.

In all cases the configuration file is pointed to by hConfig, a configuration handle previously returned by CONFIG_openFile. The Parameter is taken from the Section indicated by pSectionName. The Paramter Type to search for is given by pTypeName. (Section and Parameter names have white space stripped from them.)

CONFIG_getNumber returns its result in the location pointed to by pValue.

CONFIG_getString returns its result in the location pointed to by ppValue. The returned Value pointer points to static data. Do not modify the data pointed to by the returned Parameter.

CONFIG_setNumber assigns value to the specified Section/Type.

CONFIG_setString assigns pValue the specified Section/Type.

Note: Paramter Values may be no longer than 1024 bytes.

6.16.5 Permanance

```
typedef enum
{
    CONFIG_Permanent,
    CONFIG_ThisExecution,
    CONFIG_Transient
```

```
} CONFIG_Permanance;
```

This module does not yet write to the configuration file. The file should be created by hand, for now. For this reason, the PERMANANCE values have no meaning at this time.

For CONFIG_setNumber and CONFIG_setString, if permanance is set to CONFIG_Permanent, rewrite the configuration file with this new value assigned to the specified Section/Type. If CONFIG_ThisExecution, this value is ”local” to the current run-time environment, and returns to the value saved in the file upon program termination or upon a call to CONFIG_close. If permanance is set to CONFIG_Transient, this value is ”local to the current run-time environment, and returns to the value saved in the file upon re-reading of the configuration file (not yet implemented), or upon a call to CONFIG_close.

6.17 Subscriber Profile Module (PROFILE)

```
#include ‘‘profile.h’’

ReturnCode
PROFILE_open(char * pProfileName,
             void ** phProfile);

void
PROFILE_close(void * hProfile);

ReturnCode
PROFILE_addAttribute(void * hProfile,
                     char * pAttributeName,
                     OS_Uint32 type,
                     OS_Boolean (* pfEqual)(char * pValue1,
                                            char * pValue2,
                                            OS_Uint32 type));

ReturnCode
PROFILE_searchByAttribute(void * hProfile,
                          char * pAttributeName,
                          char * pSearchValue,
                          void ** phSearch);


ReturnCode
PROFILE_searchByType(void * hProfile,
                     OS_Uint32 type,
                     char * pSearchValue,
                     void ** phSearch);


ReturnCode
```

```
PROFILE_getAttributeValue(void * hProfile,
                          void * hSearch,
                          char * pAttributeName,
                          char ** ppValue);
```

The PROFILE module constructs searchable *Profiles* from Profile libraries. As shown in the following diagram, a Profile is a set of *Attributes* and Attribute *Values*. The same set of Attributes may apply to different *Entities*. Each Entity may use different Values for the same set of Attributes.

```
entity_i
   attribute_1 value_a
   attribute_2 value_b
        .
        .
        .

entity_j
   attribute_1 value_x
   attribute_2 value_y
        .
        .
        .
```

Each Attribute is assigned a *Type* by the user. When searching Profiles, searches may be issued based on finding a particular Attribute with a particular Value, or on finding a particular Value within any Attribute which is of a specified Type.

6.17.1 Open and Close a Profile Library

```
ReturnCode
PROFILE_open(char * pProfileName,
             void ** phProfile);
Returns:
   Success
   ResourceError
   <depending upon the implementation, other non-Success Values>

void
PROFILE_close(void * hProfile);
```

PROFILE_open opens the Profile library hProfile. In some portations, this may be a file name. phProfile is a pointer to a location in which a handle for this Profile Library will be placed.

PROFILE_close releases resources allocated on behalf of the user, for accessing a Profile library. hProfile is a handle to a Profile Library, previously provided by PROFILE_open.

6.17.2 Add an Attribute to a Profile

```
ReturnCode
PROFILE_addAttribute(void * hProfile,
                     char * pAttributeName,
                     OS_Uint32 type,
                     OS_Boolean (* pfEqual)(char * pValue1,
                                            char * pValue2,
                                            OS_Uint32 type));

Returns:
    Success
    Fail
```

PROFILE_addAttribute adds an Attribute contained in a Profile library to a searchable, local list of Attributes. It is not necessary to add all Attributes in the library to this list. Only those Attributes of concern to the application making this call need be added. These Attributes are the ones that will be searched by PROFILE_searchByAttribute and PROFILE_searchByType.

There is one exception to the above rule. The ”key” Attribute must be added, and must be the very first Attribute added. The key field is, in this INI-file-based implementation, the INI Section name (in square brackets). In a real database implementation, there would be a primary key, possibly some secondary keys (irrelavent to a profile), and a bunch of record attributes.

hProfile A handle to a Profile library, previously provided by PROFILE_open.

pAttributeName The name of an Attribute within the Profile library.

type A caller-defined value defining the Type of the Attribute. If multiple Attributes have the same Type, then PROFILE_searchByType will search all of those Attributes for a particular Value.

Note: The caller-defined portion of this Value is the low-order 31 bits. The high-order bit is reserved for OR-ing with PROFILE_MULTIVALUE. See the following section.

pfEqual A pointer to a function which is called, during searches, to compare the specified search Value against the Attribute Value in the profile. This function need not be a simple string compare. For example, in comparing an email address, we want to do case-insensitive string compares. In other cases, the user of the profile package might want to strip out some of the information in one or the other of the strings prior to doing the comparison.

PROFILE_MULTIVALUE

If an Attribute Type has the PROFILE_MULTIVALUE bit set, the Attribute name will have a space with sequential numbers appended to it in order to create a list of values for that attribute. If, for example, for the attribute ATTR_ID, PROFILE_addAttribute() is called as follows:

```
rc = PROFILE_addAttribute(hProfile,
                          "Id",
                          ATTR_ID | PROFILE_MULTIVALUE,
                          compareFunction);
```

when PROFILE_searchByAttribute() is called (see below), multiple attributes will be searched for. The attribute names will be:

```
Id 1
Id 2
  .
  .
  .
```

The search will complete when a match is found, or no more attributes in the set of Values are found.

6.17.3 Search a Profile

```
ReturnCode
PROFILE_searchByAttribute(void * hProfile,
                          char * pAttributeName,
                          char * pSearchValue,
                          void ** phSearch);

ReturnCode
PROFILE_searchByType(void * hProfile,
                     OS_Uint32 type,
                     char * pSearchValue,
                     void ** phSearch);

Returns:
    Success
    Fail
```

PROFILE_searchByAttribute searches the specified Profile for an Entity that contains a particular Attribute matching a specified Value.

PROFILE_searchByType searches the specified Profile for an Entity that contains any Attribute with the specified Type, matching a specified Value.

In both cases, the search is sequential, so if there is more than one possible search outcome only the first one is returned.

hProfile Handle to a Profile library, previously provided by PROFILE_open.

pAttributeName Name of the Attribute which is to be compared to the search Attribute Value.

type Value specifying the Attribute types to be scanned for the specified search Value.

pSearchValue Attribute Value to be searched for.

phSearch Pointer to a location in which a search handle is placed upon successful completion of this function. This search handle points to an Entity and may be passed to PROFILE_getAttributeValue() to obtain other Attributes from the Entity found in this search.

6.17.4 Get an Attribute Value

```
ReturnCode
PROFILE_getAttributeValue(void * hProfile,
                          void * hSearch,
                          char * pAttributeName,
                          char ** ppValue);

Returns:
    Success
    Fail
```

PROFILE_getAttributeValue obtains Attribute Values from the Entity determined by PROFILE_searchByAttribute or PROFILE_searchByType.

Profile Handle to a Profile library, previously provided by PROFILE_open.

hSearch Search handle, previously provided by PROFILE_searchByAttribute or PROFILE_searchByType.

pAttributeName Name of the Attribute whose Value is desired.

ppValue Pointer to the location in which a pointer to the character string representation of the requested Value is placed.

6.18 Abstract Syntax Notation (ASN)

```
#include ''asn.h''

ReturnCode
ASN_init(void);

ReturnCode
ASN_format (
        ASN_EncodingRules   encRules,
        void *              pTab,
        void *              hBuf,
        void *              pCStruct,
        OS_Uint32 *         pFormattedLength);


ReturnCode
ASN_parse(
        ASN_EncodingRules   encRules,
        void *              pTab,
        void *              hBuf,
        void *              pCStruct,
```

```
        OS_Uint32 *         pduLength);


ReturnCode
ASN_newTableEntry (
        ASN_TableEntry **   ppTab,
        ASN_TableEntryType  type,
        OS_Uint8            itemTag,
        QU_Head *           pQ,
        unsigned char *     pBase,
        void *              pExists,
        void *              pData,
        char *              pDebugMessage);

void
ASN_printTree (
        ASN_TableEntry *    pTab,
        void *              pCStruct,
        OS_Boolean          printValues);

/* Macro */
ASN_NEW_TABLE (pTab, type, itemTag, pQ, pCStruct, pExists,  \
               pData, pMessage, debugParamList)
```

ASN.1 (Abstract Syntax Notation One, CCITT X.208) is used extensively by ISO protocols to describe data structures in an manner independent of implementation. There are three aspects of ASN.1 handled by this module: the specification of the data, the encoding of the data and the decoding of the data.

The ASN.1 data has two representations in the ASN Module. The internal representation is a machine-dependent data structure which contains the ASN.1 tag, type, length and text used for debug messages; as well as the data in processor's native format and a link to the next element. The external representation is the PDU, which is a byte-stream encoding of the data into a format which is machine-independent and suitable for transmission between protocol layers. At this time, only the Basic Encoding Rules (BER, X.209) are defined in the module, but support for Packed Encoding Rules (PER) is a likely addition.

For each PDU to be used by the protocol, a *table* is created, containing the structure defined by the protocol. This *table* is then used by ASN_format() and ASN_parse() functions as described below. The table is constructed by calling the macro ASN_NEW_TABLE(...), then calling the function ASN_newTableEntry(...) for each element of the PDU.

The decoding of a PDU into a *table* allows received data to be used by the receiving layer. This is done with the ASN_parse(...) function.

The encoding of a *table* into a PDU allows the data to be written into a byte-stream for transmission to another layer. It is done with the ASN_format(...) function.

6.18.1 Trace Flags

Tracing of the ASN.1 encoding/decoding may be turned on with the following trace flags:

TM_ASN_ERROR Display ASN error messages

TM_ASN_WARNING Display ASN warning messages

TM_ASN_FLOW Selected points in the program flow

TM_ASN_TABLE Print a message when the en/decoding of a table begins

TM_ASN_ELEMENT Print a message when each element of a table is encoded or decoded

TM_ASN_DATA Print the value of elements as they are encoded or decoded

6.18.2 Functions

ASN_tableEntry(...) adds a entry to the given table, with the specified tag, type and value.

The ASN_parse(...) function accepts a PDU as input and fills in the internal data structure with the data from the PDU, using the specified encoding rules.

6.18.3 Functions

The ASN_format(...) function accepts an internal table as input and outputs a PDU suitable for network transmission. Again, the specified encoding rules are used.

6.18.4 ASN_tableEntry Declaration

```
typedef struct ASN_TableEntry
{
    QU_ELEMENT;
    ASN_TableEntryType   type;
    OS_Uint8             tag;
    OS_Uint32            minimum;
    OS_Uint32            maximum;
    OS_Uint32            existsOffset;
    OS_Uint32            dataOffset;
    OS_Uint32            maxDataLength;
    OS_Uint32            elementSize;
    char *               pDebugMessage;
    struct { QU_HEAD; }  tableList;
} ASN_TableEntry;
```

When a PDU is to be formatted, the table is scanned, and data is taken from the C structure and placed, according to the specified set of encoding rules, into the PDU. Similarly, when a PDU is to be parsed, the PDU is scanned, and the elements are placed into the C structure in their appropriate locations.

The file sdp+fs/lsm/envcomp.c contains a complete example of building a table for formatting and parsing of LSM Envelope PDUs.

6.19 Finite State Machine (FSM_)

```
Void FSM_init(Void)

FSM_createMachine(FSM_UserData *userData)
FSM_deleteMachine(FSM_Machine *machine)

FSM_runMachine(FSM_Machine *machine, FSM_EventId evtId)

FSM_UserData *FSM_getUserData(FSM_Machine *machine)
Void FSM_setUserData(FSM_Machine *machine)

FSM_generateEvent (FSM_Machine *machine, FSM_EventId evtId)

Void
FSM_TRANSDIAG_init (void);

FSM_TRANSDIAG_TransDiag
FSM_TRANSDIAG_create (String transDiagName, FSM_State *initialState)

SuccFail
FSM_TRANSDIAG_load (FSM_Machine *machine, FSM_TRANSDIAG_TransDiag *transDiag)

SuccFail
FSM_TRANSDIAG_resetMachine (FSM_Machine *machine)
```

This module is used to implement Mealy-model finite state machines. The functions of this module let you create a machine, assign your specific data to the machine, and associate the machine with one or more transition diagrams.

A classic state transition diagram has four components: states, transition arcs, events, and actions. In the FSM module a state is represented by an instance of the FSM_State data structure. A transition arc is represented by an instance of the FSM_Trans data structure. These two structures are defned as follows:

```
typedef struct FSM_State {
    Int (*entry)(FSM_Machine  *machine,
                 FSM_UserData *userData,
                 FSM_EventId  evtId);
    Int (*exit)(FSM_Machine  *machine,
                FSM_UserData *userData,
                FSM_EventId  evtId);
    struct FSM_Trans  *trans;
    String name;
} FSM_State;
```

```
typedef struct FSM_Trans {
    FSM_EventId evtId;
    Bool (*predicate)(FSM_Machine  *machine,
                      FSM_UserData *userData,
                      FSM_EventId  evtId);
    Int (*action)(FSM_Machine  *machine,
                  FSM_UserData *userData,
                  FSM_EventId  evtId);
    FSM_State  *nextStatePtr;
    String name;
} FSM_Trans;
```

The FSM_State structure contains the following members:

entry The function called when the state is entered

exit The function called when the state is exited

trans The transition diagram associated with the state. This is the diagram that defines the events that can result in transition to another state.

name A String containing the name of the state

The FSM_Trans structure contains the following members:

evtId The event that causes the transition to occur

predicate The predicate function (see below)

action The function that is called when the transition is actually taken

nextStatePtr The state to transition to

name A String containing the name of the transition

As seen in the above structures, each transition can have a predicate function. When an event occurs, if there is a predicate function associated with the transition, it is called first. If the predicate function's return value is TRUE, the action function is called. If no predicate is associated with a transition, the action function is called as soon as the event is matched. The predicate function may generate events using `FSM_generateEvent` function.

Two examples at the end of this section show you how to define and use these structures.

6.19.1 State Machine Functions

```
Void
FSM_init(Void)
```

This function initializes FSM tracing, if active, and should be called at the start of a user program.

```
FSM_createMachine(FSM_UserData *userData)
FSM_deleteMachine(FSM_Machine *machine)
```

This function creates a finite state machine.

`FSM_createMachine` allocates memory for a state machine and assigns the user data, which is passed to it as an argument, to the state machine.

```
FSM_runMachine(FSM_Machine *machine, FSM_EventId evtId)
```

This function processes an event. The machine identifier and event id are passed to the function. This function compares the given event with the events associated with the current state of the finite state machine to find the transition function and call it. In the case of an event match, if there is a predicate associated with the transition, the predicate function is called first and based on the result of the predicate function, the transition function is called. If there is no predicate associated with the transition, the transition function associated with the matched event is called unconditionally.

```
FSM_UserData *FSM_getUserData(FSM_Machine *machine)
Void FSM_setUserData(FSM_Machine *machine)
```

These functions get and set user specific data for a given machine respectively.

```
FSM_generateEvent (FSM_Machine *machine, FSM_EventId evtId)
```

This function generates an event for the finite state machine. The machine and the identifier of the event to be generated are passed to the function. The given event is not processed by this function, but instead is scheduled (by OCP's scheduler module) for later execution.

6.19.2 Transition Diagram Functions

```
Void
FSM_TRANSDIAG_init (void);
```

This function initializes the transition diagram queue head pointer and should be called at the start of a user program.

```
FSM_TRANSDIAG_TransDiag
FSM_TRANSDIAG_create (String transDiagName, FSM_State *initialState)
```

This function creates a transition diagram and assigns its name and initial state, which are passed to it as arguments. The return value is a pointer to a transition diagram if the function is successful and NULL if the function can not allocate memory.

```
SuccFail
FSM_TRANSDIAG_load (FSM_Machine *machine, FSM_TRANSDIAG_TransDiag *transDiag)
```

This function assigns a transition diagram to a previously created finite state machine. You can load different transition diagrams at will, but the last one that was loaded is in effect at any point of time.

```
SuccFail
FSM_TRANSDIAG_resetMachine (FSM_Machine *machine)
```

This function resets a finite state machines to the initial state of the transition diagram which was previously assigned to it.

6.19.3 FSM Usage Examples

The following two examples show how the FSM module is used. The first example is a simple one and second one is more complex. The state diagram in the beginning of each example shows what the example is about.

```
/*+
 * File: fsm_ex.h
 *
 * Description:
 *      This File contains the definititions related to the Finite State
 *      Machine of FSM example. Represented by the Module Identifer FSM_
 *
-*/

#ifndef _FSMEX_H
#define _FSMEX_H

#include "extfuncs.h"

    /* User Action Primitives */
#define fsmex_EvtIdleTo1        2
#define fsmex_Evt1ToIdle        3
#define fsmex_Evt1To2           4
#define fsmex_Evt2ToIdle        5

extern FSM_State fsmex_sIdle;
extern FSM_State fsmex_s1;
extern FSM_State fsmex_s2;

STATIC FSM_Trans    fsmex_tIdle[]= {
    {fsmex_EvtIdleTo1, (Bool(*)())0, tr_idleTo1,    &fsmex_s1, "F:IdleTo1"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_sIdle, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_sIdle = {
    fsm_ePass,
    fsm_xPass,
    fsmex_tIdle,
    "Idle-State"};

STATIC FSM_Trans    fsmex_t1[]= {
    {fsmex_Evt1ToIdle, (Bool(*)())0, tr_1ToIdle, &fsmex_sIdle, "F:1ToIdle"},
    {fsmex_Evt1To2,  (Bool(*)())0, tr_1To2,      &fsmex_s2,    "F:1To2"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_s1, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_s1 = {
    fsm_ePass,
    fsm_xPass,
    fsmex_t1,
    "State-1"};

STATIC FSM_Trans    fsmex_t2[]= {
    {fsmex_Evt2ToIdle, (Bool(*)())0, tr_2ToIdle, &fsmex_sIdle, "F:2ToIdle"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_s2, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_s2 = {
    fsm_ePass,
```

```
    fsm_xPass,
    fsmex_t2,
    "State-2"};

typedef union fsmex_EventInfo {

    struct evt1 {
        Int field1;
        Int field2;
    } evt1;

    struct evt2 {
        Int field1;
        Int field2;
    } evt2;

} fsmex_EventInfo;

fsmex_EventInfo fsmex_evtInfo;  /* Global data for all FSM_ handlers */

#endif
```

```
/*+
 * File name: fsm_ex.c (FSM Example)
 *
 * Description: (Example program for FSM module)
 *
 *           ______
 *          / Idle \
 *          \______/\
 *         /\    /\  \
 *        /       \   \
 *       /         \   \
 *      /           \  \/
 *   _______         ______
 *  /State 2\<--- /State 1\
 *  \_______/      \______/
 *
 *
 * Functions:
 *   fsmex_transDiag (void)
 *   fsm_eError (Void *machine, Void *userData, FSM_EventId evtId)
 *   tr_badEventIgnore(Void *machine, Void *userData, FSM_EventId evtId)
 *
 *   fsm_ePass(Void *machine, Void *userData, FSM_EventId evtId)
 *   fsm_xPass(Void *machine, Void *userData, FSM_EventId evtId)
 *   Int tr_idleTo1(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_1ToIdle(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_1To2(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_2ToIdle(Void *machine, Void *userData, FSM_EventId)
 *
 *   fsm_e   Invoked when ENTERING a state.
 *   fsm_x   Invoked when EXITING a state.
 *   fsm_a   Invoked as ACTION of a transition.
 *   fsm_t   TRANSITION specificatin.
```

```c
 *   fsm_s   STATE specification.
-*/

#ifdef RCS_VER /*{*/
static char rcs[] = "$Id: fsmExc.c,v 1.1 2008/01/07 01:04:31 mohsen Exp $";
#endif /*}*/

#include "estd.h"
#include "eh.h"
#include "queue.h"
#include "sch.h"

#include "fsm.h"
#include "fsmtrans.h"
#include "fsm_ex.h"
#include "extfuncs.h"


Void
main(int argc, char **argv)
{
    int c;
    extern char *optarg;
    extern int optind;

    Void *machine;
    Void *transDiag;

    typedef struct UserData {
        char name[20];
    } UserData;

    UserData userData = {"Example program"};

    TM_INIT();

    while ((c = getopt(argc, argv, "T:t:")) != EOF) {
        switch ( c ) {

#ifdef TM_ENABLED
        case 'T':
            TM_setUp(optarg);
            break;
#endif
        }
    }

    SCH_init(100);

    FSM_init();

    FSM_TRANSDIAG_init();

    if ( ! (transDiag = FSM_TRANSDIAG_create("fsmExTransDiag", &fsmex_sIdle)) ) {
        EH_problem("main: FSM_TRANSDIAG_create failed");
        exit(13);
    }
```

```
    machine = FSM_createMachine(&userData);

    FSM_TRANSDIAG_load(machine, transDiag);

    FSM_TRANSDIAG_resetMachine(machine);


    {
        int i;
        static FSM_EventId event[5] = {
            fsmex_EvtIdleTo1,
            fsmex_Evt1To2,
            fsmex_Evt2ToIdle,
            -1
        /* fsmex_Evt1ToIdle, */
        };

        for (i = 0; event[i] != -1; i++) {
            FSM_runMachine(machine, event[i]);
        }

        FSM_deleteMachine(machine);
        SCH_block();
        SCH_run();
    }
}

Int
tr_idleTo1(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_idleTo1: Transition: %s\n",
             FSM_getUserData(machine)));
    return 0;
}

Int
tr_1ToIdle(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_1ToIdle: Transition: %s\n",
             FSM_getUserData(machine)));
    return 0;
}

Int
tr_1To2(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_1To2: Transition: %s\n",
             FSM_getUserData(machine)));
    return 0;
}

Int
tr_2ToIdle(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_2ToIdle: Transition: %s\n",
             FSM_getUserData(machine)));
```

```
    return 0;
}


/*<
 * Function:    tr_badEventIgnore
 *
 * Description: Bad event is ignored.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
>*/

Int
tr_badEventIgnore(Void *machine, Void *userData, FSM_EventId evtId)
{
    EH_problem("tr_badEventIgnore: Bad Event Ignored\n");
    return (SUCCESS);
}


/*<
 * Function:    fsm_ePass
 *
 * Description: Invoked when entering a state.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
>*/

Int
fsm_ePass(Void *machine, Void *userData, FSM_EventId evtId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC,
        "fsm_ePass:    machine=0x%x evtId=0x%x\n", machine, evtId));

    return (SUCCESS);
}



/*<
 * Function:    fsm_xPass
 *
 * Description: Invoked when exiting a state.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
>*/

Int
```

```
fsm_xPass(Void *machine, Void *userData, FSM_EventId evtId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC,
        "fsm_xPass:    machine=0x%x evtId=0x%x\n", machine, evtId));

    return (SUCCESS);
}
```

```
/*+
 * File: fsm_ex.h
 *
 * Description:
 *       This File contains the defininitions related to the Finite State
 *       Machine of FSM example. Represented by the Module Identifer FSM_
 *
-*/

#ifndef _FSMEX_H
#define _FSMEX_H

#include "extfuncs.h"

    /* User Action Primitives */
#define fsmex_EvtIdleTo1        2
#define fsmex_Evt1ToIdle        3
#define fsmex_Evt1To2           4
#define fsmex_Evt2ToIdle        5
#define fsmex_Evt1To3           6
#define fsmex_Evt3ToIdle        7

extern FSM_State fsmex_sIdle;
extern FSM_State fsmex_s1;
extern FSM_State fsmex_s2;
extern FSM_State fsmex_s3;

STATIC FSM_Trans    fsmex_tIdle[]= {
    {fsmex_EvtIdleTo1, (Bool(*)())0, tr_idleTo1,    &fsmex_s1, "F:IdleTo1"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_sIdle, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_sIdle = {
    fsm_ePass,
    fsm_xPass,
    fsmex_tIdle,
    "Idle-State"};

STATIC FSM_Trans    fsmex_t1[]= {
    {fsmex_Evt1ToIdle, (Bool(*)())0, tr_1ToIdle,   &fsmex_sIdle, "F:1ToIdle"},
    {fsmex_Evt1To2,    (Bool(*)())0, tr_1To2,      &fsmex_s2,    "F:1To2"},
    {fsmex_Evt1To3,    (Bool(*)())0, tr_1To3,      &fsmex_s3,    "F:1To3"},
    {FSM_EvtDefault,  (Bool(*)())0, tr_badEventIgnore, &fsmex_s1, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_s1 = {
    fsm_ePass,
    fsm_xPass,
    fsmex_t1,
```

```
    "State-1"};

STATIC FSM_Trans    fsmex_t2[]= {
    {fsmex_Evt2ToIdle, (Bool(*)())0, tr_2ToIdle, &fsmex_sIdle, "F:2ToIdle"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_s2, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_s2 = {
    fsm_ePass,
    fsm_xPass,
    fsmex_t2,
    "State-2"};

STATIC FSM_Trans    fsmex_t3[]= {
    {fsmex_Evt3ToIdle, (Bool(*)())0, tr_3ToIdle, &fsmex_sIdle, "F:3ToIdle"},
    {FSM_EvtDefault, (Bool(*)())0, tr_badEventIgnore, &fsmex_s3, "I:BadEventIgnored"} };

PUBLIC FSM_State fsmex_s3 = {
    fsm_ePass,
    fsm_xPass,
    fsmex_t3,
    "State-3"};

typedef union fsmex_EventInfo {

    struct evt1 {
        Int field1;
        Int field2;
    } evt1;

    struct evt2 {
        Int field1;
        Int field2;
    } evt2;

} fsmex_EventInfo;

fsmex_EventInfo fsmex_evtInfo;  /* Global data for all FSM_ handlers */

#endif
```

```
/*+
 * File name: fsm_ex2.c (FSM Example)
 *
 * Description: (Example program for FSM module)
 *
 *   In this example, the following state machine follows the following path:
 *       Idle -> S1 -> S2 -> Idle -> S1 -> S3 -> Idle
 *
 *   Selection of either S1->S2 or S1->S3 when in state 1, is based on an
 *   internal event generated in transition from Idle to S1.
 *
 *                    ______
 *            ---->/ Idle \<----
 *           /     \______/     \
 *          /       /\  |        \
 *         /        |   |         \
```

```
 *         /             |    |             \
 *        /              |    \/             \
 *    ______          ______        ______
 *  /State 2\<---- /State 1\---->/State 3\
 *  \______/       \______/      \______/
 *
 *
 * Functions:
 *   fsmex_transDiag (void)
 *   fsm_eError (Void *machine, Void *userData, FSM_EventId evtId)
 *   tr_badEventIgnore(Void *machine, Void *userData, FSM_EventId evtId)
 *
 *   fsm_ePass(Void *machine, Void *userData, FSM_EventId evtId)
 *   fsm_xPass(Void *machine, Void *userData, FSM_EventId evtId)
 *   Int tr_idleTo1(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_1ToIdle(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_1To2    (Void *machine, Void *userData, FSM_EventId)
 *   Int tr_2ToIdle(Void *machine, Void *userData, FSM_EventId)
 *   Int tr_1To3    (Void *machine, Void *userData, FSM_EventId)
 *   Int tr_3ToIdle(Void *machine, Void *userData, FSM_EventId)
 *
 *   fsm_e   Invoked when ENTERING a state.
 *   fsm_x   Invoked when EXITING a state.
 *   fsm_a   Invoked as ACTION of a transition.
 *   fsm_t   TRANSITION specificatin.
 *   fsm_s   STATE specification.
 *   G_heartBeat(void)
-*/

#ifdef RCS_VER /*{*/
static char rcs[] = "$Id: fsmEx2c.c,v 1.1 2008/01/07 01:04:31 mohsen Exp $";
#endif /*}*/

#include "estd.h"
#include "eh.h"
#include "queue.h"
#include "sch.h"

#include "fsm.h"
#include "fsmtrans.h"
#include "fsm_ex2.h"
#include "extfuncs.h"

extern Int      G_heartBeat(void);


Void
main(int argc, char **argv)
{
    int c;
    extern char *optarg;
    extern int optind;

    Void *transDiag;
    Void *machine;

    typedef struct UserData {
```

```
        char name[20];
    } UserData;

    UserData userData = {"Example program"};

    TM_INIT();

    while ((c = getopt(argc, argv, "T:t:")) != EOF) {
        switch ( c ) {

#ifdef TM_ENABLED
        case 'T':
            TM_setUp(optarg);
            break;
#endif
        }
    }

    SCH_init(100);

    FSM_init();

    FSM_TRANSDIAG_init();

    if ( ! (transDiag = FSM_TRANSDIAG_create("fsmExTransDiag", &fsmex_sIdle)) ) {
        EH_problem("main: FSM_transDiag_create failed");
        exit(13);
    }

    machine = FSM_createMachine(&userData);

    FSM_TRANSDIAG_load(machine, transDiag);

    FSM_TRANSDIAG_resetMachine(machine);


    {
        int i;
        static FSM_EventId event[5] = {
            fsmex_EvtIdleTo1,
            fsmex_Evt2ToIdle,
            fsmex_EvtIdleTo1,
            fsmex_Evt3ToIdle,
            -1
        /* fsmex_Evt1ToIdle, */
        };

        for (i = 0; event[i] != -1; i++) {
            FSM_runMachine(machine, event[i]);

            if (G_heartBeat()) {
                exit(13);
            }
        }

        FSM_deleteMachine(machine);
        if (G_heartBeat()) {
```

```
            exit(13);
        }
    }
}

Int
tr_idleTo1(Void *machine, Void *userData, FSM_EventId eventId)
{
    static Bool firstTime = 1;

    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_idleTo1: Transition: %s\n",
              FSM_getUserData(machine)));

    if (firstTime) {
        firstTime = 0;
        FSM_generateEvent(machine, fsmex_Evt1To2);
    } else {
        FSM_generateEvent(machine, fsmex_Evt1To3);
    }

    return 0;
}

Int
tr_1ToIdle(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_1ToIdle: Transition: %s\n",
              FSM_getUserData(machine)));
    return 0;
}

Int
tr_1To2(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_1To2: Transition: %s\n",
              FSM_getUserData(machine)));
    return 0;
}

Int
tr_2ToIdle(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_2ToIdle: Transition: %s\n",
              FSM_getUserData(machine)));
    return 0;
}

Int
tr_1To3(Void *machine, Void *userData, FSM_EventId eventId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_1To3: Transition: %s\n",
              FSM_getUserData(machine)));
    return 0;
}

Int
tr_3ToIdle(Void *machine, Void *userData, FSM_EventId eventId)
```

```
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC, "tr_3ToIdle: Transition: %s\n",
            FSM_getUserData(machine)));
    return 0;
}


/*<
 * Function:    tr_badEventIgnore
 *
 * Description: Bad event is ignored.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
>*/

Int
tr_badEventIgnore(Void *machine, Void *userData, FSM_EventId evtId)
{
    EH_problem("tr_badEventIgnore: Bad Event Ignored\n");
    return (SUCCESS);
}


/*<
 * Function:    fsm_ePass
 *
 * Description: Invoked when entering a state.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
>*/

Int
fsm_ePass(Void *machine, Void *userData, FSM_EventId evtId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC,
        "fsm_ePass:   machine=0x%x evtId=0x%x\n", machine, evtId));

    return (SUCCESS);
}



/*<
 * Function:    fsm_xPass
 *
 * Description: Invoked when exiting a state.
 *
 * Arguments:  State machine, event id.
 *
 * Returns:     0 if completed successfule, -1 otherwise.
 *
```

```
>*/

Int
fsm_xPass(Void *machine, Void *userData, FSM_EventId evtId)
{
    TM_TRACE((FSM_modCB, FSM_TMFUNC,
        "fsm_xPass:   machine=0x%x evtId=0x%x\n", machine, evtId));

    return (SUCCESS);
}


/*<
 * Function:    G_heartBeat
 *
 * Description: Heart Beat for the stack.
 *
 * Arguments:  None.
 *
 * Returns:    0 on successful completion, -1 on unsuccessful completion.
 *
>*/

SuccFail
G_heartBeat(void)
{
    if (SCH_block() < 0) {
        EH_fatal("main: SCH_block returned negative value");
        return (FAIL);
    }

    SCH_run();

    return (SUCCESS);
}
```

6.20 User Datagram Protocol Point of Control & Observation (UDP_PCO_)

```
#include udp_pco.h

SuccFail
UDP_init(int number_of_SAPs)

UDP_SapDesc
UDP_sapBind(T_SapSel *sapSel,
            int (*dataInd) (T_SapSel  *remTsapSel,
                            N_SapAddr *remNsapAddr,
                            T_SapSel  *locTsapSel,
                            N_SapAddr *locNsapAddr,
```

```
                           DU_View   data));

SuccFail
UDP_sapUnbind(TSapSel *sapSel)

SuccFail
UDP_dataReq(UDP_SapDesc locSapDesc,
            T_SapSel    *remTsapSel,
            N_SapAddr   *remNsapAddr,
            DU_View     udpSdu)

SuccFail
UDP_PO_init(String errFile,
            String logFile)

SuccFail
UDP_PC_inhibit(Int direction,
               Int next)

SuccFail
UDP_PC_lossy(Int direction,
             Int percent)
```

The UDP_PCO_ package provides a connectionless, non-blocking means for inter-process communication. Any UDP_PCO_ user may create a SAP (Service Access Point) via which other processes may communicate.

The UDP_PCO_ package provides a simplified interface to the datagram sockets communication function. The processes communicating need not be running on the same machine; they may be on any machine on the Internet.

The UDP_PCO_ package provides points of control and observation at the N-1 interface of any UDP_PCO_ user. User data units can be logged for subsequent analysis during the software development process. User data units can also be interrupted under program control to simulate the effects of a lossy network on a UDP_PCO_ user.

6.20.1 Initializing the Package

```
SuccFail UDP_init(int number_of_SAPs)
```

This function initializes the UDP_PCO_ package and creates space for the specified number of SAPs.

6.20.2 Creating a SAP

```
UDP_SapDesc
UDP_sapBind(T_SapSel *sapSel,
            int (*dataInd) (T_SapSel  *remTsapSel,
                            N_SapAddr *remNsapAddr,
                            T_SapSel  *locTsapSel,
                            N_SapAddr *locNsapAddr,
                            DU_View   data));
```

This function creates a SAP and "binds" the application to it. The SAP Selector is returned through the 'sapSel' argument. The user-supplied function 'dataInd' is called whenever data is received through this SAP. See below for a description of this function's parameters.

6.20.3 Removing a SAP

```
SuccFail UDP_sapUnbind(T_SapSel *sapSel)
```

This function removes (unbinds) a SAP.

6.20.4 Sending Data via a SAP

```
SuccFail
UDP_dataReq(UDP_SapDesc locSapDesc,
            T_SapSel    *remTsapSel,
            N_SapAddr   *remNsapAddr,
            DU_View     udpSdu)
```

This function sends data to the remote SAP specified by 'remTsapSel', on the remote machine specified by 'remNsapAddr'. The data is contained in 'udpSdu'. See the DU_ module section for a description of the 'DU_View' data structure.

6.20.5 Receiving Data via a SAP

```
int
(*dataInd) (T_SapSel  *remTsapSel,
            N_SapAddr *remNsapAddr,
            T_SapSel  *locTsapSel,
            N_SapAddr *locNsapAddr,
            DU_View   data));
```

As discussed above, user data is received via a user-supplied function that is registered via the UDP_bind facility. This function is invoked by the UDP_PCO_ module whenever it receives data addressed to the T_SAP selector to which the user function was bound.

'remTsapSel' and 'remNsapAdd' are the T_SAP selector and N_SAP address, respectively, of the data sender. 'locTsapSel' and 'locNsapAdd' are reserved for future use. Do not rely on their values.

The received data is contained in 'data'. See the DU_ module section for a description of the 'DU_View' data structure.

6.20.6 Logging User Data

```
#ifdef UDP_PO_

SuccFail
UDP_PO_init(String errFile,
            String logFile)
```

```
#endif
```

UDP_PCO_ user data units can be logged to a file for subsequent analysis during software development by using the UDP_PO_ facilities. These facilities can be removed from the code at build-time by undefining the UDP_PO_ preprocessor variable in target.h.

UDP_PO_init opens an error file and a log file for writing, where the files are named according to the text strings 'errFile' and 'logFile', respectively.

The error file is not used at present.

The log file accepts log records which consist of user data units prepended with UDP_PO_ headers. The header format is defined in udp_po.h

6.20.7 Intentionally Interrupting User Data

```
#ifdef UDP_PC_

SuccFail
UDP_PC_inhibit(Int direction,
               Int next)

SuccFail
UDP_PC_lossy(Int direction,
             Int percent)

#endif
```

These functions are used to intentionally interrupt user data units to simulate the effects of a lossy network on a UDP_PCO_ layer user. These facilities can be removed from the code at build-time by undefining the UDP_PC_ preprocessor variable in target.h.

UDP_PC_inhibit causes the UDP_PCO_ layer to "drop" data units. If direction is set to UDP_PC_INHIBIT_SEND then outbound data units (with respect to the UDP_PCO_ layer user) are interrupted. If direction is set to UDP_PC_INHIBIT_RECEIVE then inbound data units are interrupted. The value of next determines how many data units to drop. A call to UDP_PC_inhibit with next equal to 0 will unconditionally enable data units in the indicated direction regardless of any previous commands.

UDP_PC_lossy functions in a manner similar to UDP_PC_inhibit but instead of dropping a set number of the next data units, it drops a percent of the data units over time. (The algorithm which determines exactly when to drop a data unit is based on a random number generator.) percent can vary between 0 and 100. 100 causes all data units to be dropped. 0 allows all data units to pass through. A call to UDP_PC_lossy with percent equal to 0 will unconditionally enable data units in the indicated direction regardless of any previous commands.

6.20.8 Build Options

The UDP_PCO_ module allows optional compilation of the point-of-observation and point-of-control features. These features may be independently enabled or disabled from within the file target.h. With the point-of-observation feature disabled PDU transactions are not logged. With the point-of-control feature disabled PDUs can not be inhibited.

The preprocessor directive

```
#define UDP_PC_
```

enables the point-of-control feature, while

```
#define UDP_PO_
```

enables the point-of-observation feature.

6.20.9 Example

Although the two sides of a connection are in most senses peers, they may be thought of as client and server (or 'user' and 'provider'). The server process merely initializes a SAP (service access point) and waits for input (via the scheduler system):

```
void function(T_SapSel *remote_sap, N_SapAddr *remote_address,
        T_SapSel *local_sap, N_SapAddr *local_address, DU_View data);
UDP_init(2);
UDP_sapBind(&localSAP,function);
while (SCH_block())
        SCH_run();
```

(There are a few more details to it, but this the core of what has to happen in the server. See the sample program for details.)

The client has merely to perform a similar process:

```
void function(T_SapSel *remote_sap, N_SapAddr *remote_address,
        T_SapSel *local_sap, N_SapAddr *local_address, DU_View data);
UDP_init(2);
UDP_sapBind(&localSAP,function);

/*
        outbound data
*/

UDP_dataReq(localSAPdesc, &remote_sap, &remote_address, data);
/*
        If expecting a reply:
*/
while (SCH_block())
        SCH_run();
```

6.21 IMQ

```
Synopsis

SuccFail IMQ_init(void);        /* initialize */
String IMQ_nameCreate(void);    /* create a queue name */
```

```
IMQ_Key IMQ_keyCreate(String name, Int subID); /* create a queue key */
IMQ_PrimDesc IMQ_acceptConn(IMQ_Key key); /* accept connections */
int IMQ_connect(IMQ_PrimDesc queue); /* connect to client */
IMQ_PrimDesc IMQ_clientConnect(IMQ_Key key); /* connect from client to server */
Int IMQ_primSnd(IMQ_PrimDesc primDesc, IMQ_PrimMsg *data, Int size); /* send */
Int IMQ_primRcv(IMQ_PrimDesc primDesc, IMQ_PrimMsg *data, Int size); /* receive */
```

The IMQ package is a thin wrapper around the socket interprocess communication mechanism. The UPQ package is an easier-to-use higher-level interface and should normally be used instead of using IMQ directly.

The package uses SOCK_STREAM sockets, so (1) send/receive operations may be done only while the server is actually up and (2) operations are guaranteed properly sequenced and reliable. See the socket man page for details.

Any process using IMQ to act as a server must establish a queue name and a key, which is then advertised to clients as a connection point. (Clients need only the key, not the queue name.)

```
Server:

        IMQ_init();
        name = IMQ_nameCreate();
        key = IMQ_keyCreate(name,1); /* must communicate key to client somehow */
        desc = IMQ_acceptConn(key);
        while ((otherside = IMQ_connect(desc)) == 0) /* does not block */
                sleep(n);
        /* can then send and receive to 'otherside' */

Client:

        IMQ_init();
        serverdesc = IMQ_clientConnect(key);    /* server's key */
        /* can then send and receive to 'serverdesc' */
```

6.22 UPQ

```
Synopsis

        USQ_init();
        PSQ_init();
        USQ_putAction(queue, buf, sizeof(SP_Action));
        PSQ_register(lowbound, hibound, action_routine, cleanup_routine);
        PSQ_putEvent(queue, buf, size);
```

The UPQ_BSD module provides an interprocess communication mechanism between a server machine and an arbitrary number of clients within a single machine (or any entity conforming to AF_UNIX network type). There are two versions, a 'sync' version which blocks, and an 'async' version which does not block.

The provider process advertises its availability on a public queue, named /tmp/SP, available to all. The user processes then connect to the provider, and the UPQ_BSD code establishes a bidirectional "primitive queue" between provider and user. The queue from user to provider is called the 'action' queue and the queue from provider to user the 'event' queue.

The provider process also needs to know the name of the action/event queue pair, which is generated in the user process by the mktemp() routine. It's the responsibility of the application code to arrange to send the name of the queue (via the USQ_putAction routine) from user process to provider process.

The provider expects action messages in the form of the structure SP_Action. The function

```
PSQ_register(lowbound, hibound, processing_routine, clean_connection);
```

is then called to register with the provider shell system functions to handle the actual work of the provider process. For example:

```
PSQ_register(0,800,process_SAT_score, cleanup);
```

More than one such set of functions may be registered. The 'clean_connection' routine is called when the user drops the connection to the provider. This routine should take any action necessary to clean up after a connection that goes dead.

If the type of the incoming action falls outside of the bounds of any set of parameters of all registered processing functions, the system reports an error via the trace logging mechanism.

The provider calls the scheduler routine SCH_BLOCK() to wait for incoming messages and SCH_RUN() to run the appropriate action routine when a message is received.

The user program is trivial; it need merely initialize, create the action/event queue pair (USQ_primQuCreate), and send and receive messages to/from the provider (USQ_putAction and USQ_getEvent).

Service Access Points: How This Package Is Used

This package can not only connect peer processes, but can also be used for a provider to provide a "service access point" (SAP) for a number of users. The provider can then give users access to more than one layer of a multilayer system (such as a computer network based on the TCP/IP or OSI reference models).

6.23 Release Identification Module (RELID_)

```
#include ''relid.h''

        Char *RELID_getRelidNotice(void);
```

This module allows you to incorporate release id information into binary executables. The particular string that RELID_getRelidNotice retrieves is extracted from the "Release Notes" associated with a particular release of a product or a system. The information contained in the string retrieved by RELID_getRelidNotice includes the following information:

System or Product Name: This is a unique name that identifies a program or a set of matching programs belonging to a system.

Release Number: The particular release number (CVS/RCS revision number) for the release notes that captures this release.

Release Tag: The particular release tag (CVS/RCS revision tag) for the release notes that captures this release.

Date of Release: The date (day and time) of checking in of the release notes that captures this release. If the release note is the last module the is checked in for a particular release, then a dated check out can retrieve that particular release.

Date of Build: The date (day and time) the binary was built.

Person of Build: The user-id of the person who built this binary.

Machine of Build: The name of the machine that was used to build this binary.

Location of Build: The directory in which the binary was built.

The RELID_ string containing the above information can easily be formatted in many ways. The current process for generation of the RELID_ string is described below.

6.23.1 Generation of RELID_ String

The current implementation comprises of the following components:

releaseNotes.tex: A text file (often in LaTeX format), which includes RCS expanded names and numbers and tags which identify the particular release.

relid.tpl: A template "C" file that is used to generate "relid.c".

genrelid.sh: The script which extracts the release information from "releaseNotes.tex" and generates "relid.c" through "relid.tpl".

relid.c: A machine generated "C" file that is built by "genrelid.sh" based on "relid.tpl" which implements RELID_getRelid.

Customizing or modifying the RELID_ string, primarily involves editing the "genrelid.sh" script.

6.23.2 Example Usage

Using the RELID_ module can be as simple as:

```
LOG_message("%s\n", RELID_getRelidNotice());
```

The generated string under Solaris appears below.

```
afrasiab-3% relidusr

CURENV.SOL2 1.1.1.1 (CVS tag unspecified) released on 1996/12/02 20:59:10.
Built on Wed Dec 4 18:32:52 PST 1996 by pean on afrasiab SunOS 5.5.
```

6.24 Copyright Notice Module (CPR_)

```
#include ``cpr.h''

        Char *CPR_getCopyrightNotice(void);
```

This module allows you to incorporate copyright into binary executables. The Copyright Notice is subject to content integrity checks and protects against binary editing of the copyright notice.

The following code fragment demonstrates the use of the module.

```
    if ( ! (copyrightNotice = RELID_getCopyrightNotice()) ) {
        EH_problem("main: get copyright failed");
        EH_exit(1);
    }

     LOG_message("%s\n\n", copyrightNotice);
```

6.25 License Module (LIC_)

This module provides a program to generate valid license files and an API for programs to check against their associated license files.

```
the 'licgen' program is used to generate the license files

#include ``lic.h''

        SuccFail LIC_check(char *pLicenseFile);

        SuccFail LIC_computeHash(char *licensee,
                                char *startDate,
                                char *endDate,
                 char *hostID,
                 char **hash);

        SuccFail LIC_checkHash(char *licensee,
                              char *startDate,
                              char *endDate,
                       char *hostID,
                              char *hash);
```

This module is intended for enabling applications to perform license-checking. The parameters of the software license is recorded in a license file. The license checking involves:

- An integrity check on the license file to see that it has not been tampered with.
- License start date. The software should not be run before the license period begins.
- License end date. The software should not be run after the license expires.
- The host identifier of machine running the software. The software should only be run on the machine identified by the hostid.

The integrity check on the license file is done by hashing the licensee, start date, end date and host id information and comparing it against the check digits in the license file to ensure that the license file has not been tampered with.

The host id is a 32-bit unsigned value returned by **INET_gethostid()** expressed as an unpadded decimal integer. The start and end dates are expressed in YYYYMMDD format.

6.25.1 Generating A License File

The program **licgen** is used to generate license files. It prompts you for the name of the licensee, the start and end dates and the host id of the machine that is licensed to run the software. Following is typical licgen session:

```
icarus-15% ./licgen
Enter Licensee: Neda Communications, Inc.
Licensee is <Neda Communications, Inc.>
Enter Start Date: 19961004
Start Date is <19961004>
Enter End Date: 20010101
End Date is <20010101>
Enter Host ID: 272629761
Host ID is <272629761>
Hash is <7011197853557>
checks out using LIC_checkHash()!
icarus-16%
```

Using this information, you then create a license file of the following form:

```
#
# this line is a comment as it starts with a '#' character at the
# beginning of the line...
#
[License]
   Licensee = J Random Company, Inc.
   Start Date = 19970219
   End Date = 19970401
   Host ID = 272629761
   Check Digits = 7013006635469
```

6.25.2 Checking Against a License File

When the application starts up, one of the first things it should do is to check the license file that is associated with the program and machine. This is done by a call to **LIC_check(¡license filename¿)**. Your program should link against **liblic.a**.

6.26 Integer To English (int2english)

```
#include "int2english.h"

Char *
PF_intToCardinalEnglish(strBegin, strEnd, value)
Char *strBegin;   /* output */
Char *strEnd;     /* output */
LgInt value;      /* input */

Char *
PF_intToOrdinalEnglish(strBegin, strEnd, value)
Char *strBegin;    /* output */
Char *strEnd;      /* output */
LgInt value;       /* input */

Char *
PF_intToDigitEnglish(strBegin, strEnd, value)
Char *strBegin;     /* output */
Char *strEnd;       /* output */
LgInt value;        /* input */

char *
PF_strToDigitEnglish(char *strBegin, char *strEnd, String digitStr)
```

This module provides a program to convert integer into a readable ASCII string in English.

6.26.1 PF_intToCardinalEnglish

The PF_intToCardinalEnglish function convert any integer into cardinal English. It takes integer *value* as an input and produces a string of phonemes. Below are several examples of what will be produced by PF_intToCardinalEnglish function:

- if the integer *value* is **1**, the outcome of this will be **"one"**
- if the integer *value* is **2005**, the string output is **"two thousand and five"**, etc.

6.26.2 PF_intToOrdinalEnglish

The PF_intToOrdinalEnglish function convert any integer into ordinal English. It takes integer *value* as an input and produces a string of phonemes. Below are several examples of what will be produced by PF_intToOrdinalEnglish function:

- if the integer *value* is **1**, the outcome of this will be **"first"**
- if the integer *value* is **2005**, the string output is **"two thousandth fifth"**, etc.

6.26.3 PF_intToDigitEnglish

The PF_intToDigitEnglish function convert any integer into ordinal English. It takes integer *value* as an input and produces a string of phonemes. Below are several examples of what will be produced by PF_intToDigitEnglish function:

- if the integer *value* is **1**, the outcome of this will be **"one"**
- if the integer *value* is **11**, the string output is **"one one"**, etc.

6.26.4 PF_strToDigitEnglish

The PF_intToDigitEnglish function convert any integer into ordinal English. It takes string *value* as an input and produces a string of phonemes. Below are several examples of what will be produced by PF_strToDigitEnglish function:

- if the string *value* is **"1"**, the outcome of this will be **"one"**
- if the string *value* is **"58"**, the string output is **"five eight"**, etc.

Chapter 7

EXAMPLE USAGE OF THE PLATFORM

To demonstrate the proper usage of the platform a simple short stack is implemented.

The middle layer is Inactive Network Layer Protocol (INLP_). Below it (LOWER_) is a simple loop back data link layer. Above it (UPPER_) is an entity that periodically sends a datagram.

The following platform facilities are used in this example.

```
DU_, TMR_,
```

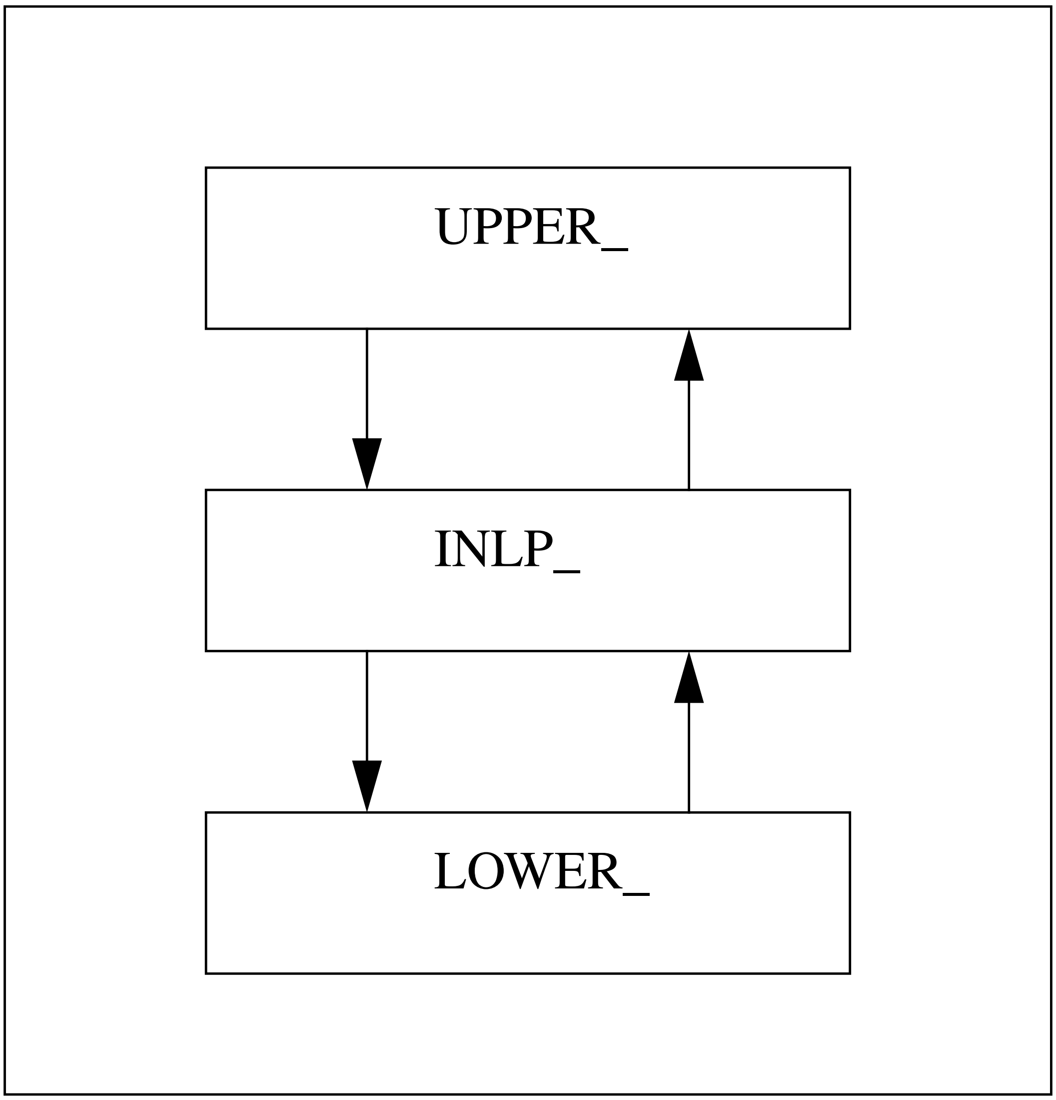

Figure 7.1: Sample Simple Short Stack

Chapter 8

IMPLEMENTATION OF THE PLATFORM

The Open C Platform may be implemented for a large number of independent target environments (compiler, hardware, operating system). Rather than attempt to fully support every one of these environments, OCP uses a ”choose-and-configure” approach to support as many as possible and to the fullest extent possible. Thus, OCP offers a rich ”Components Inventory” to the integrator from which she may build the Open C Platform for her target environment.

Many of the OCP modules do not depend on the underlying operating system. These so-called ’unhosted’ modules are discussed in Section 8.2 below. Some OCP modules, on the other hand, require the presence of a particular underlying operating system. These ’hosted’ modules are discussed in Section 8.3. Still other OCP modules may be implemented with any one of a number of underlying operating systems or even with none at all. These ’portable’ modules are discussed in Section 8.4.

Sections 8.5.1 8.5.2 and 8.5.3 discusses the implementation of the hosted and portable modules under the Unix, MS-DOS, and VMS operating systems, respectively.

8.1 Implementation Map

Figure 8.1 depicts the relative relationships of the OCP modules to the user application, operating system, computing hardware, and compiler. The OCP modules occupy the central block in the diagram. Individual modules, represented as bubbles, occupy one of three regions within this block.

Un-hosted modules are in the lower left. They are completely independent of any operating system but do depend on the underlying computing hardware as well as on the compiler. The global include file `oe.h` contains information which the compiler uses to correctly build un-hosted modules for a given computing platform.

The Queue (QU_) module is one example of an un-hosted module. It requires no operating system services to provide its service.

Hosted modules occupy the right-hand side of the OCP block. They require an underlying operating system to perform such actions as disk file I/O, interprocess communication, and network communication. The global include file `os.h` contains information which the compiler uses to correctly build hosted modules for a given operating system.

The User Process Queue/BSD (UPQ_BSD_) Module is one example of a hosted module. It depends on the presence of interprocess communication facilities provided under BSD implementations of Unix and is thus highly operating system-dependent.

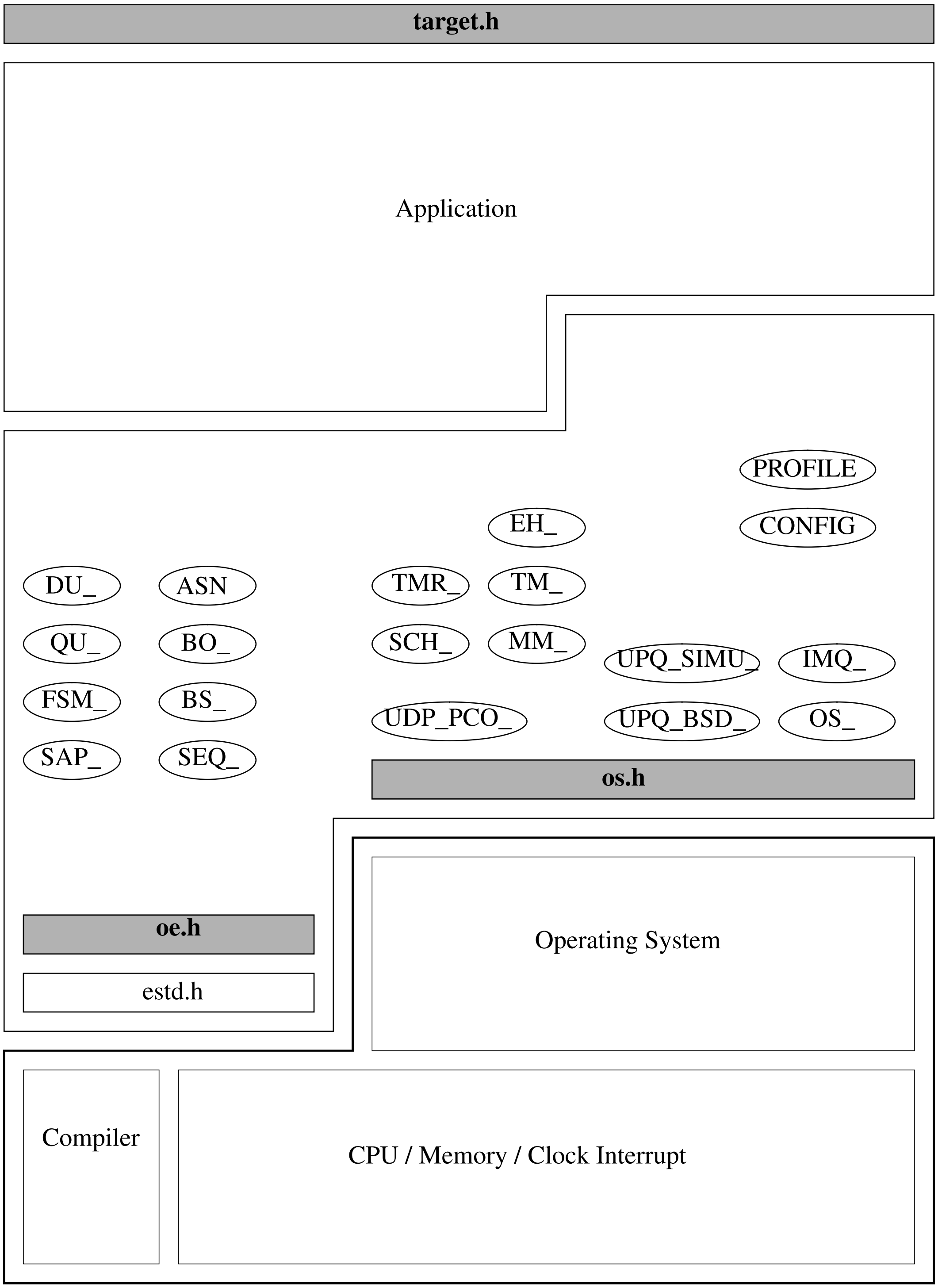

Figure 8.1: Implementation Map

Portable modules occupy the middle of the OCP block. They may use one of several underlying operating systems. Some may also be implemented independently of any operating system.

The User Datagram Protocol Interface (UDP_PCO_) Module is one example of a portable module. It depends on the presence of an underlying TCP/UDP/IP stack for network communications but this stack is available under numerous operating systems.

The Timer (TMR_) Module is an example of a module which may or may not use an underlying operating system. Some implementations might use the Unix signal() facility to provide an underlying timing mechanism whereas others may simply use a periodic hardware interrupt.

Finally, note the importance of the global include file `target.h`. This application-specific file contains information used throughout a given instance of an Open C Layer - information such as the sizes of data buffers and queues, timer periods, etc.

8.2 Un-hosted Facilities

The following Open C Platform modules do not depend on the underlying operating system. However, some of the modules may still depend on the underlying hardware architecture. These dependencies are encapsulated in the global include file `oe.h`. Thus, when building any of the following modules for a given processor, the developer need only substitute the `oe.h` include file for his specific computing platform.

- FSM_ Finite State Machine Module
- DU_
- QU_ Queue Module
- BO_ Byte Ordering Module
- BS_ Byte String

8.3 Hosted Facilities

The following OCP modules require the presence of a specific underlying operating system (O/S).

- UPQ_BSD_ User Process Queue(Berkeley Software Distribution) Module
- UPQ_SIMU_ User Process Queue(Simulated) Module

These modules are all highly dependent on the underlying O/S as well as on the computing hardware and compiler. The O/S dependencies are encapsulated in the global include file `os.h`. Thus, when building any of the above modules the developer must provide the appropriate version of `os.h` as well as `oe.h` for his specific computing platform.

8.4 Portable Facilities

The following OCP modules may be implemented under many operating systems or, in some cases, without the presence of an underlying operating system at all.

OCP Trace Bit Definitions

Module Name	Trace Bit	Mask	Type of Tracing
ASN	0	0001	All ASN activity: formatting, parsing
DU_	10	0400	All DU_ activity: allocate, free, link
FSM_	2	0004	FSM_TMEXEC
	3	0008	FSM_TMGEN
	4	0010	FSM_TMFUNC
IMQ_	0	0001	All IMQ_ activity: allocate, free, link
MM	1	0002	Error
	3	0004	Normal activity
	8	0100	PDU dump
SCH_	0	0001	All Scheduler activity: queue manipulation, task execution
UDP_	0	0001	All UDP_ activity: receive, bind

Figure 8.2: Trace Module Bit Definitions

- UDP_PCO_ User Datagram Protocol Interface Module
- TMR_ Timer Module
- SCH_ Scheduler Module
- PROFILE Module
- CONFIG Module
- TM_ Trace Module

8.4.1 TM_ Module

Figure 8.2 indicates the specific trace features that are enabled by the bit masks for those OCP modules that use tracing.

8.5 Supported Operating Systems

8.5.1 UNIX

The following hosted modules exist for the Unix operating system:

- UPQ_BSD_ User Process Queue(Berkeley Software Distribution) Module
- UPQ_SIMU_ User Process Queue(Simulated) Module

The following portable modules have been ported to the Unix operating system:

- UDP_PCO_ User Datagram Protocol Interface Module
- TMR_ Timer Module
- SCH_ Scheduler Module
- PROFILE Module
- CONFIG Module
- TM_ Trace Module

CONFIG Module Configuration File Format

The following discussion relates to Unix CONFIG Module configuration files.

Sections are denoted by strings enclosed in square brackets. Parameter Types and Parameter Values are separated by “equal sign” characters.

CONFIG Module configuration files may contain macro definitions and macro uses. Macros are defined with a leading percent sign. Macros are used (i.e. the expansion of the macro is replaced with the macro usage) by enclosing the macro name in curly braces and preceding it with a dollar sign. Environment variables are implicitly defined macros, and macros may use environment variable names in the same way as macro names defined within the configuration file.

In addition to the macro capability, a ”shell command” may be executed (in environments which support it, such as Unix). To run a shell command, contain the command within back-quotes.

Comments are indicated by a “number sign” character at the start of a line. Any text on that line is ignored.

The following example contains a number of macro definitions, use of environment variables (SHELL), a fairly complicated example of command execution (with macro expansion), and two Sections, the first containing five parameters (with macro expansion), and the second containing one parameter.

```
%macro = expansion
%myshell = ${SHELL}
%hostname = `uname -n`

%lookup = nslookup -query=a chaos

%awkBegin = BEGIN { foundHost = 0; }
%awkName = /Name:/ { foundHost = 1; }
%awkAddr = /Address:/ { if (foundHost) print $2; }

%ip addr = `${lookup} | awk '${awkBegin} ${awkName} ${awkAddr}'`

# This is a comment line

[s1]
        v1 = hello world
        v2 = macro ${macro} test
        v3 = My shell is ${myshell} equiv to ${SHELL}
```

```
        v4 = My path is ${PATH}
        v5 = IP Address = ${ip addr}

[section 2]
        v1 = 23
```

PROFILE Module File Format

The current portation of the PROFILE module is implemented on top of the CFG (Configuration) module. However, profiles may be stored in any type of database, and additional portations may be created to store profiles in Unix DBM files, Microsoft Access databases, etc.

8.5.2 MS-DOS

The following hosted modules exist for the MS-DOS operating system:

- UPQ_SIMU_ User Process Queue(Simulated) Module

The following portable modules have been ported to the Unix operating system:

- UDP_PCO_ User Datagram Protocol Interface Module
- TMR_ Timer Module
- SCH_ Scheduler Module
- TM_ Trace Module

8.5.3 VMS

Chapter 9

DEVELOPMENT ENVIRONMENT

The Open C Platform may be implemented for a large number of independent target environments (hardware, operating system). Further, the Open C Platform can easily be integrated with a large number of software development environments. A large number are currently supported and adding new ones is quite easy.

Here, we document the general characteristics that are needed to be understood for convenient developement under current supported environments as well as the information needed for integrating OCP into other development environments.

In addition to the portability of the sources, the build procedure for all OCP modules are designed to be location independent and development system independent. By defining certain cononical capabilities for the virtual compiler, the virtual librarian, the virtual linker and the virtual loader, the OCP build procedures are easy to move arround.

9.1 Supported Development Environments

The following development environments are currently supported.

9.1.1 Solaris

Under Solaris 2.X (or any UNIX) standard unix tools (ksh, make, ...) are used. GNU tools (gmake, ...) are also supported.

9.1.2 Windows NT

Under Windows NT the following tools are required:

mksnt An environment that includes UNIX type commands and the Korn shell.

opus-make Opus Make v6.06

9.1.3 Windows 95

Under Windows 95 The following tools are required:

mksnt An environment that includes UNIX type commands and the Korn shell.

opus-make Opus Make v6.06

9.1.4 DOS

Under DOS [limited support] The following tools are required:

mksnt An environment that includes UNIX type commands and the Korn shell.

opus-make Opus Make v6.06

9.2 Supported Compilers

The following compilers are currently supported.

9.2.1 GCC

The GNU C Compiler is supported under Solaris.

9.2.2 Microsoft Visual C++ Developer Studio

Windows CE Developer

9.2.3 Borland C 4.5

The Borland C Compiler (version 4.5) is supported under DOS, Windows 95 and NT.

9.2.4 Purify

Purify tools are supported under Solaris.

9.3 Supported Target Environments

The following target environments are currently supported.

- Solaris 2.X (or any UNIX)
- Windows NT
- Windows 95
- DOS
- Windows CE
- A-Engine (A 186 embedded environment)

9.3.1 Solaris

Almost all OCP modules are supported under Solaris.

9.3.2 Windows NT

All hosted OCP modules are supported under Solaris.

9.3.3 Windows 95

All hosted OCP modules are supported under Solaris.

9.3.4 DOS

All hosted OCP modules are supported under Solaris.

9.3.5 Windows CE

All hosted OCP modules are supported under Solaris.

Windows CE Emulator

9.3.6 Embedded

A-Engine is a generic 186 CPU board which supports a variety of peripherals (e.g., serial ports, keypads, displays).

All un-hosted OCP modules are supported under Solaris.

9.4 General Philosophy

Build procedures for OCP are developed with the following high level design principles.

Self Contained Modules The basic building blocks are: [pkg] and [pdt].

Location Independent OCP Modules can be moved around the build file hierarchy.

Development System Independent The sources and the makefiles make no specific assumptions about the development environment that is currently in use.

Simple and Generic Makefiles similar to xmkmp.

Extensive use of Shell Scripts

9.5 Build Mechanisms

The OCP build procedures hing around the use of **make** utility in a particular way. This specific use of the make utility is centered around the concept of separating the content of a traditional makefile into the following 3 different categories:

1. Target and Components. The name of the target (the program or the library) and the name of the components (source files and libraries and link modules).

2. Rules. Make rules describing how to produce binaries from sources.

3. Parameters. Development environment specific parameters such as the name of the compiler in use,.....

The "Target and Components" category is specific to a particular OCP module. The "Rules" and "Parameters" category are specific to the particular development environment in use. For this reason, the OCP makefiles only contain the "Target and Components" information. The rules and parameters are supplied to make by a make front-end (see below).

Although this description focusses on building binaries from C sources, more complex uses of makefiles such as use of yacc, lex and custom macro processors is also supported.

Build procedures for all OCP modules involve the following components.

Makefile Very naked. List of sources and header files.

Make Initialization File Deals with rules and environment specific issues. For example, using OPUS make, we can deal with DOS's command line length limitations in the make initialization file.

Make Front-End Script The make front-end script takes make parameters such as the compiler name ("CC=bc45") from the "Make Parameters File" and passes them to make. The Make Front-End script is a shell script called **mkp.sh**.

Make Parameters File The make parameters file includes all the product, target environment and development environment specific parameters. The make parameters file is a shell script that is sourced/doted by the make front-end script. The naming convention for the make parameter file is: **product+target+compiler.sh/**

Cononical Tools Compiler, librarian, linker, locator.

9.5.1 Adding New Modules

Adding a new module to OCP process involves creating a new "pkg" or "pdt" makefile for that module.

9.5.2 Adding Support For New Compilers

Adding support for new compilers involves creating new "Make Parameters File" and possibly new "Cononical Tools" such as librarian and linker front-ends.

Appendix A

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

2. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) The modified work must itself be a software library.

   (b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   (c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   (d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

5. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

6. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

7. As an exception to the Sections above, you may also compile or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

(a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

(b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

(c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

(d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

8. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   (a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   (b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

9. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

11. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

12. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

13. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

14. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

15. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

16. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the library's name and an idea of what it does. *Copyright (C)* **year name of author**

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, *1 April 1990 Ty Coon, President of Vice*

That's all there is to it!