DECnet
Digital Network Architecture

Maintenance Operations Protocol
Functional Specification

Version V4.0.0

June 1992

This document describes the structure, functions, interfaces, and protocols needed for the low level maintenance of a DECnet network.

digital™

To order additional copies of this document, contact your local Digital Equipment Corporation Sales Office.

Order Number: EK-DNA11-FS-001

EK-DNA11-FS-001
First Printing, June 1992

Preface

This document is one of a series of specifications which describe Phase V of the Digital Network Architecture. Other DNA Phase V specifications may be obtained by ordering one or more of the following documentation kits:

DIGITAL NETWORK ARCHITECTURE (PHASE V) DOCUMENTATION KIT #1
Order No. EK-DNAP1-DK-001 includes:

Digital Network Architecture (Phase V) *General Description*, No. EK-DNAPV-GD

Digital Network Architecture (Phase V) *Network Services Protocol Functional Specification*, No. EK-DNA15-FS-001

Digital Network Architecture (Phase V) *Open Systems Interconnect Transport Functional Specification*, No. EK-DNA12-FS-001

Digital Network Architecture (Phase V) *X.25 Access Functional Specification*, No. EK-DNA17-FS-001

DIGITAL NETWORK ARCHITECTURE (PHASE V) DOCUMENTATION KIT #2
Order No. EK-DNAP2-DK-001 includes:

Digital Network Architecture (Phase V) *Session Control Functional Specification*, No. EK-DNA07-FS-001

Digital Network Architecture (Phase V) *Naming Service Functional Specification*, No. EK-DNANS-FS-002

Digital Network Architecture (Phase V) *Network Routing Layer Functional Specification*, No. EK-DNA03-FS-001

Digital Network Architecture (Phase V) *Unique Identifier Functional Specification*, No. EK-DNA16-FS-001

Digital Network Architecture (Phase V) *Time Service Functional Specification*, No. EK-DNA04-FS-001

DIGITAL NETWORK ARCHITECTURE (PHASE V) DOCUMENTATION KIT #3
Order No. EK-DNAP3-DK-001 includes:

Digital Network Architecture (Phase V) *High-Level Data Link Control Functional Specification*, No. EK-DNA08-FS-001

Digital Network Architecture (Phase V) *Carrier Sense Multiple Access with Collision Detection Functional Specification*, No. EK-DNA13-FS-001

Digital Network Architecture (Phase V) *Digital Data Communications Message Protocol Functional Specification*, No. EK-DNA14-FS-001

Digital Network Architecture (Phase V) *Modem Connect Functional Specification*, No. EK-DNA10-FS-001

Digital Network Architecture (Phase V) *LAN Node Product Functional Specification*, No. EK-DNA06-FS-001

DIGITAL NETWORK ARCHITECTURE (PHASE V) DOCUMENTATION KIT #4
Order no. EK-DNAP4-DK-001 includes:

Enterprise Management Architecture (EMA) *Entity Model Functional Specification*, No. EK-EMAEM-FS-002

Digital Network Architecture (Phase V) *Common Management Information Protocol Functional Specification,* No. EK-DNA01-FS-001

Digital Network Architecture (Phase V) *Event Logging Functional Specification*, No. EK-DNA09-FS-001

Digital Network Architecture (Phase V) *Maintenance Operations Protocol Functional Specification*, No. EK-DNA11-FS-001

Digital Network Architecture (Phase V) *Network Control Language Functional Specification*, No. EK-DNA05-FS-001

Digital Network Architecture (Phase V) *Network Management Functional Specification*, No. EK-DNA02-FS-001

Table of Contents

Figures

Tables

Chapter 1

Introduction

Certain maintenance functions need to be performed remotely at a low level in the overall network architecture. These are functions that cannot depend on high level software being operational in the node (system) being maintained.

In the context of this specification, low level implies direct usage of data link services. High level means such network functions as routing and end-to-end, virtual circuit type protocols, both of which are also users of data link services. This specification assumes that only a minimal level of data link services are available to support maintenance operations, and that these maintenance operations provide a base on which any higher level functions can be built.

This document describes the structure, functions, interfaces, and protocols needed for low level maintenance. DNA is the model on which DECnet implementations are based. A DECnet network is a family of software modules, data bases, and hardware components used to tie DIGITAL systems together for resource sharing, distributed computation or remote system communication.

This document incorporates the support of the new data links such as HDLC data link, CSMA/CD (IEEE 802.3) data link, and FDDI (ANSI X3.139–1987) data link. The term Local Area Network (LAN) used throughout this document applies to all Ethernet data link, CSMA/CD data link, and FDDI data links.

This document assumes that the reader is familiar with computer communications and DECnet. The primary audience consists of those who implement DECnet systems or other systems under different architectures, but requiring the same functions. The following are the relevant documents (refer to the Preface for the order numbers):

- Digital Network Architecture (Phase V) *Digital Data Communications Message Protocol Functional Specification*
- Digital Network Architecture (Phase V) *High-Level Data Link Control Functional Specification*
- Digital Network Architecture (Phase V) *Carrier Sense Multiple Access with Collision Detection Functional Specification*
- Digital Network Architecture (Phase V) *FDDI Data Link Architecture Specification,* (to be published)
- Digital Network Architecture (Phase V) *Network Management Functional Specification*

- Digital Network Architecture (Phase V) *Common Management Information Protocol Functional Specification*
- Digital Network Architecture (Phase V) *LAN Node Product Functional Specification*
- Enterprise Management Architecture (EMA) *Entity Model Functional Specification*
- Digital Network Architecture (Phase V) *Event Logging Functional Specification*
- Digital Network Architecture (Phase V) *Naming Service Functional Specification*
- Digital Network Architecture (Phase V) *Unique Identifier Functional Specification*
- Digital Network Architecture (Phase V) *Time Service Functional Specification*

The relevant National and International Standards are:

- IEEE Standard 802: Overview and Architecture.
- IEEE Standard 802.1 (Part B) : Addressing, Internetworking, and Network Management.
- IEEE Standard 802.2 : IEEE Standards for Local Area Networks: Logical Link Control, ANSI/IEEE Std 802.2–1985, ISO 8802–2.
- IEEE Standard 802.3 : Carrier Sense Multiple Access with Collision Detect (CSMA/CD), ANSI/IEEE Std 802.3–1985 ISO 8802–3.
- ANSI Standard X3.139–1987 : Fiber-distributed data interface (FDDI) — token ring media access control (MAC).
- ANSI X3T9/92-037, Rev 7.1 : Fiber-distributed data interface (FDDI) — Station Management (SMT).

1.1 Functional Description

Low level maintenance functions are divided into three categories. Operation within any category depends on the operability of at least part of the preceding category. The categories are:

- Communications test
- System console
- System load/dump

Each of these functions can be viewed either from the active or passive end. The active end is the one that is driving the maintenance function and the passive end is the one that is responding.

Communications test determines if the data link communications path is operative.

System console provides low level access to a system for the functions of:

- Identify
- Read data link

- Boot system
- Console carrier

The console carrier is a general purpose console input/output channel. It provides a common communication mechanism to allow remote access regardless of console command specifics.

System load/dump copies the contents of processor memory to or from a remote system.

Throughout this document, the term boot is used to mean the process of causing a system to initialize itself. Initialization may include loading system memory. A boot command is a cause. The term load is used to mean the process of transferring a system image into processor memory from some source. This is one potential effect of a boot command. The source of major interest in this specification is a remote system, accessed via a communication channel.

1.2 Design Scope

The low level maintenance operations require certain characteristics to be present, attempt to meet certain goals, and lack some features that are not within the scope of the design.

1.2.1 Requirements

The maintenance operation architecture must have the following characteristics:

- The communications test function, system console functions, and system load/dump function previously mentioned must be included in the design.
- Active and passive sides of maintenance operations can be implemented and used independently.
- Effects of errors (such as operator errors, protocol errors, and hardware errors) are minimized, always leaving a system in a well defined state.
- On CSMA/CD data link, the LAN Loop Test Protocol must be compatible with the intercompany standard Ethernet Loopback Protocol as specified in the DEC STD 134–0 (Digital CSMA/CD (Ethernet) Local Area Network Specification)
- Implementations may select subsets of functions based on particular product need. The required subsets of functions are defined in the LAN Node Product Architecture Specification.

1.2.2 Goals

The maintenance operation design tries to have the following characteristics:

- Functions and protocols are compatible with the DNA Maintenance Operation Protocol (MOP) version V3.1.0.
- Algorithms, particularly those found in memory-only systems, are processing and memory efficient. Communications efficiency is a secondary goal. In the specific case of down-line load and up-line dump, overall speed of operation is an important goal.
- Extensible to accommodate newly developed functions or modification of current functions.
- Operates independently of the underlying communication mechanism (e.g., DDCMP, Ethernet, CSMA/CD, HDLC, etc.).

- No complex algorithms or data bases. Minimal state kept in the smallest systems.

1.2.3 Non-goals

The maintenance operation design does not try to have the following characteristics:

- Isolation of components that have failed in a failing system.
- System security in the low level maintenance functions.

Chapter 2

Models

This chapter describes the relationship of the low level maintenance operations to other network layers and modules. Although this specification primarily relates the maintenance operations to DNA, the same relationships can also be applied within other network architectures, such as the DIGITAL System Communication Architecture.

2.1 Relationship to DIGITAL Network Architecture

The maintenance operations reside in the DNA Network Management Layer. They are direct users of the DNA Data Link Layer. The other DNA layers are not required in the support of the low level maintenance operations unless such services as remote file access are to be used.

DNA is a layered structure. Modules in each layer perform distinct functions. Modules within a single DNA layer (but typically in different computer systems) communicate using specific protocols. Modules in different layers (but typically in the same computer system) interface using subroutine calls or a system-dependent method. In this document interfaces are described in terms of calls to subroutines.

The diagram in Figure 1 below shows the overall layering of DNA. A later diagram (Figure 2) shows the simplified model that is applicable to the low level maintenance operations.

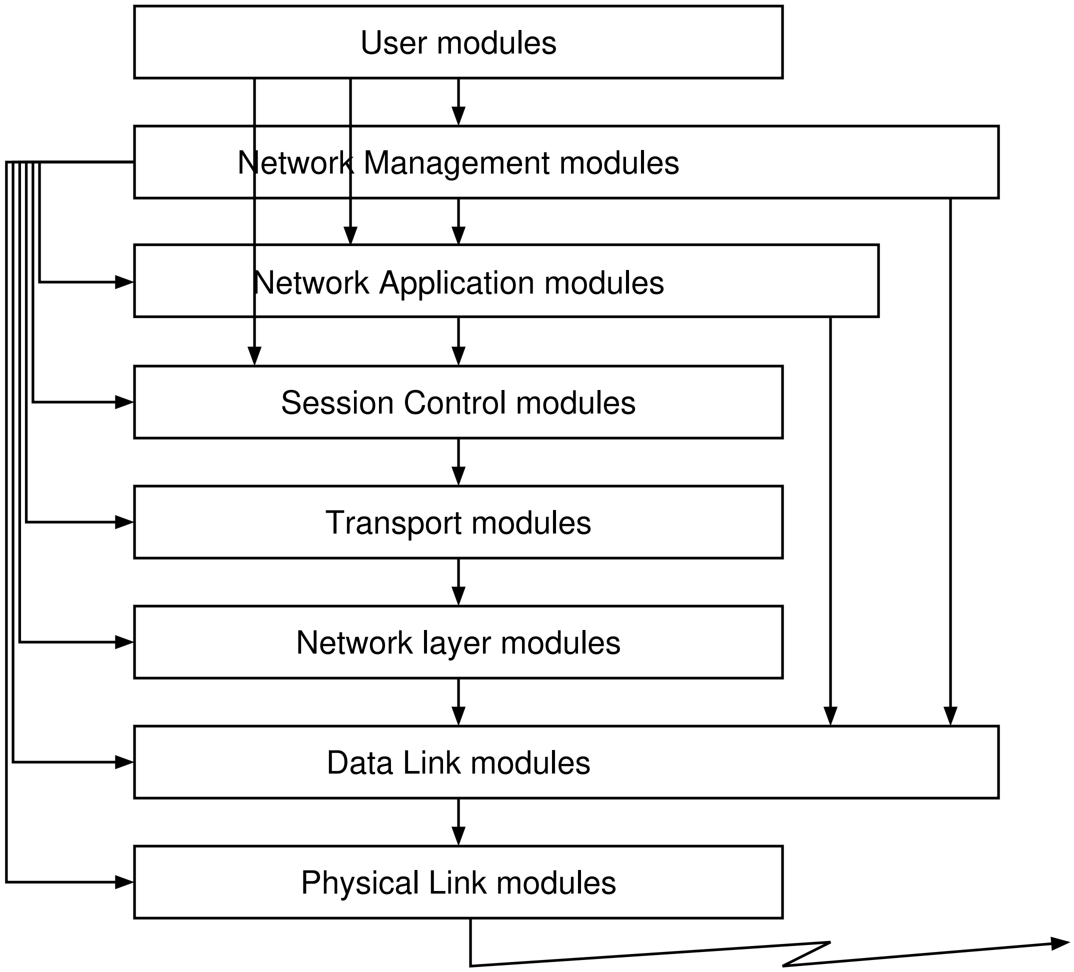

Figure 1: DNA layer model

Note:

Horizontal arrows show direct access for control and observation of parameters, counters, etc. Vertical arrows show interfaces between layers for normal user operations such as file access, down-line load, and logical link usage.

Each layer in DNA consists of functional modules and protocols. Generally, modules use the services of the next lower layer. In this document, the service relationship is demonstrated in the way the interfaces are modeled, as calls to subroutines. Note that the Network Management Layer interfaces directly with each of the lower layers. Also, the layers above Session Control interface directly with it. For this reason the upper three layers are sometimes referred to as the "end user".

Modules of the same type in the same layer communicate with each other to provide their services. The rules governing this communication and the messages required constitute the protocol for those modules. Messages are typically exchanged between equivalent modules in different nodes. However, equivalent modules within a single node can also exchange messages.

A brief description of each layer follows in order from the highest to the lowest layer:

- User Layer. The highest layer, the User Layer supports user services and programs. Programs such as the Network Control Program, which interfaces with the Network Management Layer, and file transfer programs, which interface with the Network Application Layer, reside in the User Layer.

- Network Management Layer. The Network Management Layer is the only one that has direct access to each lower layer for control purposes. Modules in this layer provide user control over and access to network parameters and counters. These modules also perform up-line dumping, down-line loading, and testing functions.

- Network Application Layer. Modules in the Network Application Layer support network functions, such as remote file access and file transfer, used by the User and Network Management Layers.

- Session Control Layer. The Session Control defines the system-dependent aspects of logical link communication, which allows messages to be sent from one node to another in a network. Session Control functions include name-to-address translation, process addressing, and, in some systems, process activation and access control.

- Transport Layer. The Transport Layer defines the system-independent aspects of logical link communication.

- Network Routing Layer. Modules in the Network Routing Layer route messages, called packets, between source and destination nodes.

- Data Link Layer. The Data Link Layer defines the protocol concerning data integrity and physical channel management.

- Physical Link Layer. The Physical Link Layer encompasses a part of the device driver for each communications device plus the communications hardware itself. The hardware includes interface devices, modems, and the communication lines.

2.2 Simplified Network Model

The diagram in Figure 2 shows a simplified relationship of the maintenance operations to the rest of the network architecture.

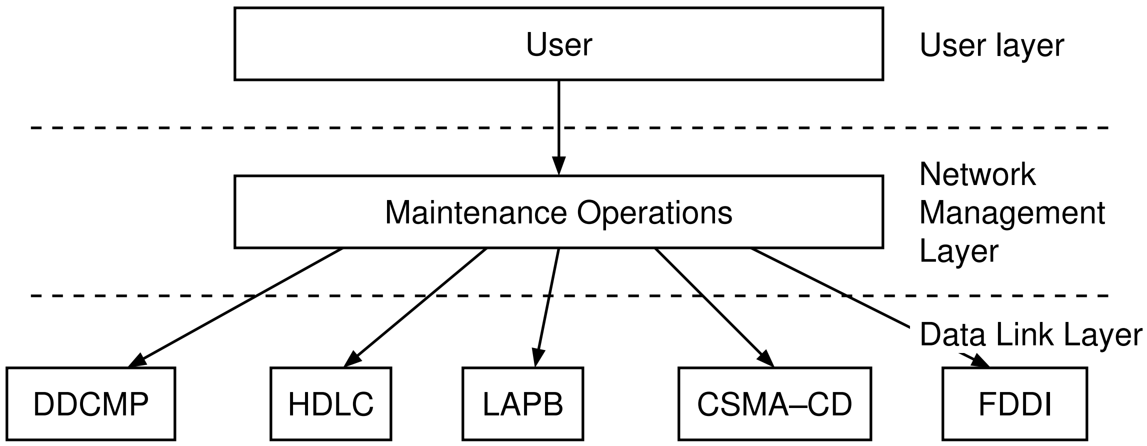

Figure 2: Simplified network model

2.3 Low Level Maintenance Operation Model

The diagram in Figure 3 shows the components within the maintenance operation module.

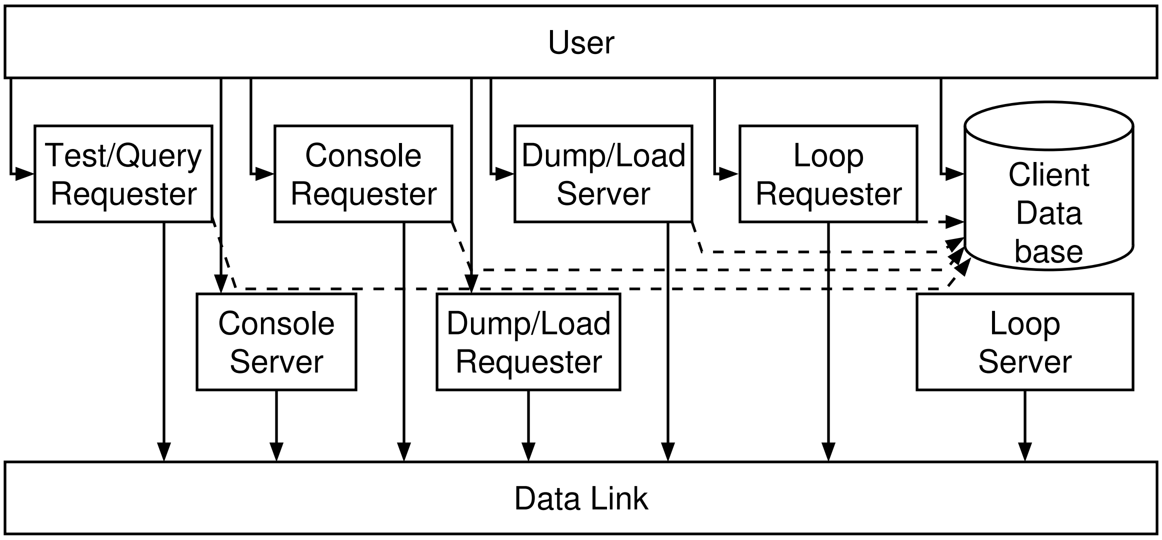

Figure 3: MOP components

Requesters are the processes responsible for initiation of maintenance operations. This can be done either at higher level user request, or because of information obtained from a lower level. Requesters are the active side of a maintenance operation.

Servers are the processes that respond to maintenance requesters. They are the passive side of a maintenance operation. Servers should not try to do more than they are capable of. For example, it is not acceptable to always volunteer to load every system that requests it and then take too long to get done because the local resources are overextended. The diagram shows servers and requesters as separate to represent their functional independence. In an implementation that supports multiple servers and/or requesters that use the same protocol type, they may have to be more closely coupled so that messages received through the data link are properly demultiplexed. Also, servers and requesters that allow multiple users must further demultiplex messages to the proper user processes.

The Client Data Base contains default information that the Dump/Load Server, Loop Requester, Console Requester, and Test/Query Requester use to fill in necessary values in incomplete requests.

Lines to the top of processes indicate flow of the control data that initiates processing. Lines to the side indicate Network Management control. The dashed lines indicate data base access.

Chapter 3

Management

The operation of the Maintenance Operations components may be monitored and controlled using DNA Network Management. There is a single Module entity, called MOP. It has subordinate entities which define the database of MOP clients, and the Data Link circuits to be used. Figure 4 shows the complete entity structure of MOP.

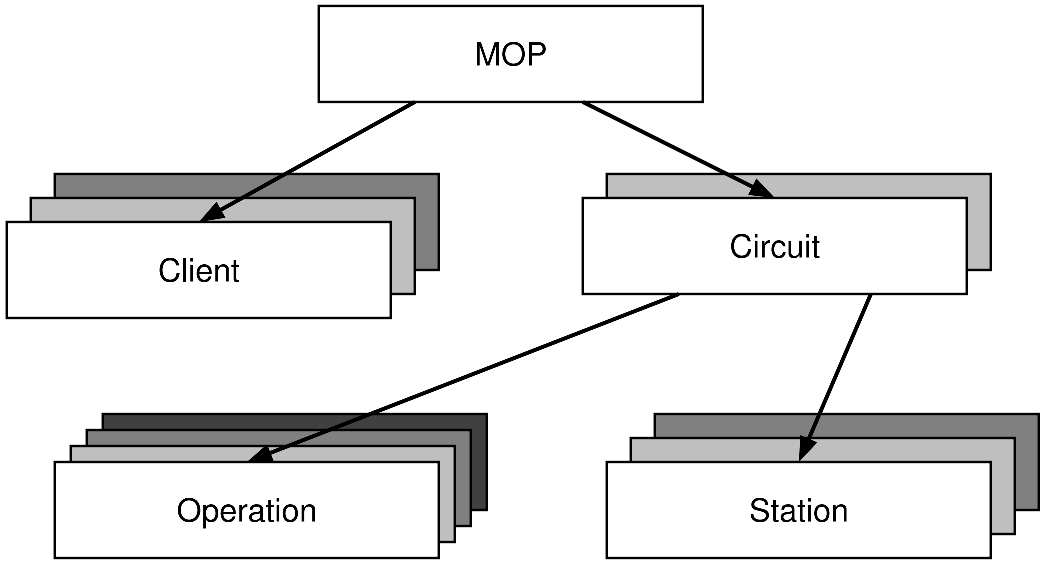

Figure 4: MOP entity structure

3.1 Client subentity

The Client subentity is used to store default parameters for the Dump/Load Server, Console Requester, Loop Requester, and Test/Query requester. When a Load, Boot, Loop, Test, or Query directive is received, any required parameters not supplied with the directive are obtained from the corresponding characteristics of a Client subentity, if one is specified in the directive. Similarly, when a request for Dump or Load service is received by the Dump/Load Server component from a Data Link, any parameters required to complete the request but not supplied in the received message are obtained from the characteristics of a Client subentity. In this case, the Client subentity to be used is determined by the Circuit and (for LAN circuits) the Data Link source address or the Device Type from the received request message.

The set of Client subentities forms the “Client Database”. A lookup in this Database consists of a search through the set of Client subentities for one that has the specified name, or the specified Circuit and (if applicable) LAN Address and/or Device Type. A LAN Address match takes precedence over a Device Type match; apart from that, the result of a search when more than one Client subentity matches the specified value is *undefined*.

3.2 Circuit subentity

The Circuit subentity describes a Data Link circuit to be used by MOP. For each defined Circuit subentity, MOP will use the specified Link Name attribute to open a port in the appropriate Data Link module. Thus, each Circuit subentity of MOP corresponds to a Data Link circuit on which MOP services are available. Another Circuit characteristic defines which of the possible MOP services are enabled.

3.2.1 Operation subentity

The Operation subentity is created dynamically by the MOP components to allow management to observe the state of MOP operations that are in progress.

3.2.2 Station subentity

The Station subentity is created dynamically by the Configuration Monitor component, if Configuration is set as one of the services enabled for a given circuit. This is applicable only to LAN circuits. The Station subentity is created in response to a received System ID message, and records the information contained in that message.

Note that MOP does not relate the Station and Client subentities in any way. They serve unrelated purposes.

3.3 Support categories

In the description of attributes below, certain directive arguments and entity attributes are flagged to indicate they are required only for some data links. Any arguments or attributes not flagged are applicable regardless of data link type.

All directive arguments shall be accepted by all implementations; when an argument is not applicable, it is ignored. Implementations which do not support a particular data link type at all must still accept (and ignore) any arguments applicable to the unsupported data link.

Implementations which don’t support a particular data link type may omit any entity attributes applicable only to those data links. In that case, attempts to access these attributes are rejected (with the standard GetListError or SetListError CMIP error code).

In this version, the only defined support category is “LAN”, which indicates any Local Area Network data link, i.e., any of the IEEE 802 family data links, Ethernet, and FDDI.

3.4 Data type definitions

The following are the definitions for the management data types used in this chapter.

FROM Management IMPORT SystemID, P4Address, P4Name;

FROM Management IMPORT LANAddress;	
FROM Management IMPORT Octet;	
SAPAddress = Octet;	(* A more descriptive type name *)
StatType =	(Off = 0, On = 1);
HelpType =	(None = 0, Transmit = 1, Receive = 2, Full = 3);
OpType =	(Loop Requester = 0, Console Requester = 1, Console Carrier = 2, Load Requester = 3, Load Server = 4, Dump Requester = 5, Dump Server = 6, Configuration Monitor = 7, Test Requester = 8, Query Requester = 9);
FunType =	BIT SET OF OpType;
SIDFunType =	BIT SET OF (Loop Server = 0, Dump Requester = 1, Primary Loader = 2, Secondary Loader = 3, Boot = 4, Console Carrier = 5, Counters = 6);
DLErrType =	(Wrong State = 0, Receive Error = 1, Transmit Error = 2);
ServType =	(Load = 0, Dump = 1, Loop = 2, Boot = 3, Test = 4, Query = 5);
ProgType =	(Secondary Loader = 0, Tertiary Loader = 1, System = 2, Management = 3, CMIP Script = 4, Diagnostic = 5);
SvErrType =	(No Resources = 0, Receive Error = 1, Transmit Error = 2, File Open Error = 3, File I/O Error = 4, Operation Aborted = 5, Unknown Client = 6, Protocol Error = 7, Timeout = 8);
CircuitType =	(CSMA–CD = 0, DDCMP = 1, FDDI = 2, HDLC = 3, LAPB = 4, Token Ring = 5, Token Bus = 6, Z–LAN = 7); [1]
DevType =	(* Enumerated, refer to Appendix A.1 for code assignments. The name to be used for each code is the 3-character mnemonic shown there in the “Name” column *);
SoftwareID = String;	(* Software ID is encoded as a text string. However, it must conform to the format restrictions documented in Section 5.5.1. If an invalid value of this data type is supplied to a directive, it is rejected with the standard CMIP error return Invalid Argument Value. *)

3.5 MOP Module

The MOP module is registered in the Distributed Systems Management Registry as shown below.

Class Name: MOP
Code: 16
Superior Class: Node

[1] Note that these codes are *not* the same as the “Data Link Type” codes used in the System ID message, which are defined in Appendix A.2. To simplify the mapping between these two encodings, new values for Circuit Type and Data Link Type will be assigned such that the following algorithm applies:

```
IF DataLinkType ≤ 10
THEN CircuitType := LookupTable[DataLinkType]
ELSE CircuitType := DataLinkType – 5
```

3.5.1 Action Directives

The MOP module implements the usual directives Create, Delete, Enable, and Disable. In addition, it implements five module-specific directives: Loop, Load, Boot, Test, and Query. If an implementation does not support some of these functions, the corresponding directive is omitted and attempts to invoke it yield the standard Directive Not Supported exception.

3.5.1.1 MOP module specific action directives

In order to avoid awkward command language syntax, the five module-specific directives are not in fact directives of the module. This is because each directive must refer to a Circuit subentity (to identify the circuit to operate on) or a Client subentity (to refer to for directive parameters not explicitly specified in the directive) or both. Thus there are actually two sets of five directives, one associated with the Client subentity, one with the Circuit subentity. Semantically these are equivalent, and the only difference in syntax is that one has an optional Circuit argument and the other an optional Client argument. If the directive is issued to a Client subentity, that subentity is in effect an implied argument of the directive; if the directive is issued to a Circuit subentity, that subentity is in effect an implied argument.

In each of these five directives, each argument individually is optional but for any particular directive enough parameters must actually be supplied to allow the operation to be performed. One possibility is to supply all necessary parameters directly as directive arguments. The other possibility is to reference a Client subentity. In that case, the parameters are obtained from the Client characteristics, unless overridden by a corresponding argument.

For each of these five directives, refer to the description below for the list of parameters that is necessary to complete the function. If all of these are supplied as arguments, it is not necessary to include a Client argument. Otherwise, the Client argument must be supplied. If no matching Client subentity is found, the request is rejected (Unrecognized Client exception). Otherwise, any parameters not supplied in the directive itself are obtained from the corresponding characteristics of the Client subentity. After completion of this “merge”, all necessary parameters must now be available; if not, the standard CMIP error Required Argument Omitted is returned.

Note:

> Since these necessary parameters can be obtained either from the directive arguments or from the Client database, it is not correct to consider them “required arguments” in the sense that they are required to be present on the directive. In particular, as far as director processing is concerned, they have to be considered as optional arguments.

The Client subentity attribute Addresses is a Set data type. For the directives, an address can be explicitly supplied; if it is not, then the operation proceeds with each of the addresses in the Set, until a response is received (except in the case of Boot, which simply loops through the entire set since no response is expected). If the set is empty, the request is rejected (standard CMIP error Required Argument Omitted).

3.5.1.2 Create Directive

The Create directive creates the MOP entity. The Characteristics initially take their default values. The Modula-2+ description follows:

```
DIRECTIVE Create = 0 : Create; (* Create the MOP entity *)
    REQUEST                         (* No input arguments *)
    END;
    RESPONSE Success = 0 :          (* No Output Arguments *)
    END Success;
```

```
    EXCEPTION Already Exists = 2 :      (* MOP Module already exists *)
    END Already Exists;
END Create;

PROCEDURE Create ( )
    RAISES {InsufficientResources};     (* CMIP common error *)
```

3.5.1.3 Delete Directive

The Delete directive deletes the MOP entity and reclaims any associated resources. The Modula-2+ description follows:

```
DIRECTIVE Delete = 1 : Delete;          (* Delete the MOP entity *)
    REQUEST                             (* No Input Arguments *)
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;
    EXCEPTION Has Children = 1 :        (* Subentities must be deleted first *)
    END Has Children;
END Delete;

PROCEDURE Delete ( )
    RAISES {};
```

3.5.1.4 Enable Directive

The Enable directive enables the MOP entity. The Modula-2+ description follows:

```
DIRECTIVE Enable = 2 : Enable;          (* Enable the MOP entity function *)
    REQUEST                             (* No input arguments *)
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;                        (* No Entity-specific Exceptions *)
END Enable;

PROCEDURE Enable ( )
    RAISES {};
```

3.5.1.5 Disable Directive

The Disable directive disables the MOP entity. The Modula-2+ description follows:

```
DIRECTIVE Disable = 3 : Disable;        (* Disable the MOP entity function *)
    REQUEST                             (* No input arguments *)
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;                        (* No Entity-specific Exceptions *)
END Disable;

PROCEDURE Disable ( )
    RAISES {};
```

3.5.2 Identifier Attribute

Modules do not have an identifier attribute.

3.5.3 Characteristic Attributes

The MOP module has the characteristic attributes shown in Table 1.

Table 1: MOP module characteristics attributes

Keyword	Code	Syntax	Value	Description
Version	0	VersionNumber	V4.0.0	The version of the Maintenance Operations specification to which the implementation conforms. This attribute is read-only.
Supported Functions	1	FunType	[System Specific]	Specifies which MOP components are available in this system. This can be used by the Director to avoid issuing directives that use omitted functions. This attribute is read-only.

3.5.4 Status Attributes

The MOP module has the status attributes shown in Table 2.

Table 2: MOP module status attributes

Keyword	Code	Syntax	Description
State	2	StatType	The current State of the MOP module.

3.5.5 Counter Attributes

The MOP module does not have any counter attributes.

3.5.6 Event Reports

The MOP module does not generate any event reports.

3.6 Client subentity

The Client subentity is registered in the Distributed Systems Management Registry as shown below.

Class Name: Client
Code: 0
Superior Class: MOP

The Client subentity is used by the Load/Dump Server, Loop Requester, Console Requester, Test Requester, and Query Requester components of MOP. On systems where none of these are implemented, the Client subentity is omitted and any attempts to issue directives to it are rejected (with the standard No Such Entity exception).

3.6.1 Action Directives

The Client subentity provides action directives for creating and deleting a subentity. These directives and the procedures they call are described below.

3.6.1.1 Create Directive

The Create directive creates a Client subentity with the specified identifier. The Characteristics initially take their default values. The Modula-2+ description follows:

```
DIRECTIVE Create = 0 : Create;           (* Create a new Client subentity *)
    REQUEST                              (* No Input Arguments *)
    END;
    RESPONSE Success = 0 :               (* No Output Arguments *)
    END Success;
    EXCEPTION Already Exists = 2 :       (* Client subentity already exists *)
    END Already Exists;
END Create;

PROCEDURE Create (Identifier: SimpleName)
    RAISES {InsufficientResources};      (* CMIP common error *)
```

3.6.1.2 Delete Directive

The Delete directive deletes a Client subentity and reclaims any associated resources. The Modula-2+ description follows:

```
DIRECTIVE Delete = 1 : Delete;           (* Delete an Client subentity *)
    REQUEST                              (* No Input Arguments *)
    END;
    RESPONSE Success = 0 :               (* No Output Arguments *)
    END Success;                         (* No Entity-specific Exceptions *)
END Delete;

PROCEDURE Delete ( Identifier )
    RAISES {};
```

3.6.1.3 Loop Directive

The Loop directive causes a loop test to be performed with another system.

```
DIRECTIVE Loop = 4 : Loop;               (* Perform a Loop operation with another system *)
    REQUEST
        ARGUMENT Circuit = 1 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Assistant System = 4 : SimpleName;
        ARGUMENT Assistant Address = 5 : LANAddress;
        ARGUMENT Assistance Type = 6 : HelpType;
        ARGUMENT Count = 7 : Integer;
        ARGUMENT Length = 8 : Integer;
        ARGUMENT Format = 11 : Octet
    END;
    RESPONSE Success = 0 :               (* No output arguments *)
    END Success;
    RESPONSE Invalid Response = 1 :
        ARGUMENT Count = 7 : Integer;    (* Count of messages successfully looped *)
    END Invalid Response;
```

```
    EXCEPTION Unrecognized Circuit = 1 : (* There is no circuit with the specified identification *)
    END Unrecognized Circuit;
    EXCEPTION Data Link Error = 2 :         (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Assistant = 5 : (* No assistant with the specified identification *)
    END Unrecognized Assistant;
    EXCEPTION Invalid Assistant = 6 :       (* The Assistant Address is either a multicast address, or
                                            Assistant System was specified but the corresponding Client
                                            subentity has an empty Addresses list *)
    END Invalid Assistant;
    EXCEPTION Timeout = 7 :                 (* No response was received in the timeout period *)
    END Timeout;
END Loop;

PROCEDURE Loop (Client, Circuit, Address, AssistantSystem,
        AssistantAddress, AssistanceType, Count, Length, Format)
    RAISES {InvalidResponse,
        InsufficientResources,              (* CMIP common error *)
        UnrecognizedCircuit,
        DataLinkError,
        UnrecognizedClient,                 (* Does not apply here *)
        RequiredArgumentOmitted,            (* CMIP common error *)
        UnrecognizedAssistant,
        InvalidAssistant,
        InvalidArgumentValue,               (* CMIP common error *)
        Timeout};
```

For LAN circuits, if Assistance Type is None, the operation is a Loop-direct; If AssistanceType is some other value, the operation is Loop-assisted. For Loop-assisted, the default assistant address is the Loop Assistant multicast address. A different address may be specified by including the Assistant Address parameter, which specifies the address directly, or by specifying Assistant System, which specifies a name of a Client subentity. The Addresses characteristic of the named Client subentity is then used as the assistant address. On non-LAN circuits, the Assistant Address, Assistant System, and Assistance Type arguments are ignored.

The required parameters are:

- Circuit
- Address (for LAN circuits)

The defaults values for the optional parameters are:

- Assistance Type — None
- Assistant Address — Obtained dynamically: an assistant is found by sending a direct (non-assisted) Loop request to the Loop Assistance multicast address. If a response is received, the responding station is then used as the assistant for the rest of the operation.
- Count — 1
- Length — 40
- Format — %x55

3.6.1.4 Load Directive

The Load directive causes a down line load to be performed to another system. A Boot is done first to force the specified system to load.

```
DIRECTIVE Load = 5 : Load;              (* Force load an adjacent system *)
    REQUEST
        ARGUMENT Circuit = 1 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT System Image = 3 : FileSpec;
        ARGUMENT Script File = 4 : FileSpec;
        ARGUMENT Secondary Loader = 5 : FileSpec;
        ARGUMENT Tertiary Loader = 6 : FileSpec;
        ARGUMENT Verification = 7 : OctetString;
        ARGUMENT Management Image = 8 : FileSpec;
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;
    EXCEPTION Unrecognized Circuit = 1 : (* There is no circuit with the specified identification *)
    END Unrecognized Circuit;
    EXCEPTION Data Link Error = 2 :     (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Protocol Error = 5;       (* A protocol error occurred during the load *)
    END Protocol Error;
    EXCEPTION Timeout = 7 :             (* No response was received in the timeout period *)
    END Timeout;
END Load;

PROCEDURE Load (Client, Circuit, Address, SystemImage, ScriptFile, ManagementImage,
        SecondaryLoader, TertiaryLoader, Verification)
    RAISES {InsufficientResources,      (* CMIP common error *)
        UnrecognizedCircuit,
        DataLinkError,
        UnrecognizedClient,             (* Does not apply here *)
        RequiredArgumentOmitted,        (* CMIP common error *)
        ProtocolError,
        InvalidArgumentValue,           (* CMIP common error *)
        Timeout};
```

Required parameters are:

- Circuit
- Address (for LAN circuits)
- System Image
- Script File (if requested by target system)
- Secondary Loader (if requested by target system)
- Tertiary Loader (if requested by target system)
- Management Image (if requested by target system)

The default values for the optional parameters are:

- Verification — %x0000000000000000

3.6.1.5 Boot directive

The Boot directive causes a Boot message to be sent to another system.

```
DIRECTIVE Boot = 6 : Boot;              (* Send a Boot trigger to an adjacent system *)
    REQUEST
        ARGUMENT Circuit = 1 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Verification = 7 : OctetString;
        ARGUMENT Device = 8 : String;
        ARGUMENT Software ID = 9 : SoftwareID;
        ARGUMENT Script ID = 10 : SoftwareID;
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;
    EXCEPTION Unrecognized Circuit = 1 : (* There is no circuit with the specified identification *)
    END Unrecognized Circuit;
    EXCEPTION Data Link Error = 2 :     (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
END Boot;

PROCEDURE Boot (Client, Circuit, Address, Verification, Device, SoftwareID, ScriptID)
    RAISES {InsufficientResources,      (* CMIP common error *)
        UnrecognizedCircuit,
        DataLinkError,
        UnrecognizedClient,             (* Does not apply here *)
        InvalidArgumentValue,           (* CMIP common error *)
        RequiredArgumentOmitted};       (* CMIP common error *)
```

Required parameters are:

- Circuit
- Address (for LAN circuits)

The default values for the optional parameters are:

- Verification — %x0000000000000000

Note:

The default value for the Verification parameter is a default only. It is not a "wild card" value. For any value of the Verification parameter including the default value, the Console Server (target of the directive) will check the value and accept the operation only if the value matches the one expected.

3.6.1.6 Test Directive

The Test directive causes the IEEE 802.2 Test to be performed with another system. If applied to the wrong type of data link (i.e., non-LAN), the standard error Directive Not Supported is returned.

```
DIRECTIVE Test = 7 : Test;              (* Perform a Test operation with another system *)
    REQUEST
        ARGUMENT Circuit = 1 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Count = 7 : Integer;
        ARGUMENT Length = 8 : Integer;
```

```
        ARGUMENT SAP = 10 : SAPAddress;
        ARGUMENT Format = 11 : Octet;
    END;
    RESPONSE Success = 0 :              (* No output arguments *)
    END Success;
    RESPONSE Invalid Response = 1 :
        ARGUMENT Count = 7 : Integer;   (* Count of messages successfully looped *)
    END Invalid Response;
    EXCEPTION Unrecognized Circuit = 1 : (* There is no circuit with the specified identification *)
    END Unrecognized Circuit;
    EXCEPTION Data Link Error = 2 :     (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Timeout = 7 :             (* No response was received in the timeout period *)
    END Timeout;
END Test;

PROCEDURE Test (Client, Circuit, Address, Count, Length, SAP, Format)
    RAISES {InvalidResponse,
        InsufficientResources,          (* CMIP common error *)
        UnrecognizedCircuit,
        DataLinkError,
        UnrecognizedClient,             (* Does not apply here *)
        RequiredArgumentOmitted,        (* CMIP common error *)
        DirectiveNotSupported           (* CMIP common error *)
        InvalidArgumentValue,           (* CMIP common error *)
        Timeout);
```

The required parameters are:

- Circuit
- Address

The defaults values for the optional parameters are:

- Count — 1
- Length — 40
- SAP — 0
- Format — %x55

3.6.1.7 Query Directive

The Query directive causes an IEEE 802.2 XID exchange to be performed with another system. If applied to the wrong type of data link (i.e., non-LAN), the standard error Directive Not Supported is returned.

```
DIRECTIVE Query = 8 : Query;            (* Perform an XID exchange with another system *)
    REQUEST
        ARGUMENT Circuit = 1 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT SAP = 10 : SAPAddress;
    END;
    RESPONSE Success = 0 :
        ARGUMENT LLC Types = 10 : Set of Integer; (* LLC types supported by system *)
```

```
        ARGUMENT Window = 11 : Integer  (* Window size for Type 2 LLC *)
    END Success;
    RESPONSE Invalid Response = 1 :         (* Protocol error in response *)
    END Invalid Response;
    EXCEPTION Unrecognized Circuit = 1 : (* There is no circuit with the specified identification *)
    END Unrecognized Circuit;
    EXCEPTION Data Link Error = 2 :         (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Timeout = 7 :                 (* No response was received in the timeout period *)
    END Timeout;
END Query;

PROCEDURE Query (Client, Circuit, Address, SAP, LLCTypes, Window)
    RAISES {InvalidResponse,
        InsufficientResources,              (* CMIP common error *)
        UnrecognizedCircuit,
        DataLinkError,
        UnrecognizedClient,                 (* Does not apply here *)
        RequiredArgumentOmitted,            (* CMIP common error *)
        DirectiveNotSupported,              (* CMIP common error *)
        InvalidArgumentValue,               (* CMIP common error *)
        Timeout);
```

The required parameters are:

- Circuit
- Address

The defaults values for the optional parameters are:

- SAP — 0

3.6.2 Identifier Attribute

The identifier attributes for each Client subentity are shown in Table 3.

Table 3: Client entity identifier attribute

Keyword	Code	Syntax	Description
Name	0	SimpleName	A string which is the Identifier for the Client (target system) and which is unique among the set of Clients defined for the MOP module.

3.6.3 Characteristic Attributes

The characteristics attributes for each Client subentity are shown in Tables 4 and 5.

Table 4: Client entity characteristics attributes (part 1)

Keyword	Code	Note	Syntax	Default Value	Description
Circuit	1		SimpleName	“”	The name of the Circuit subentity (of the MOP module) that corresponds to the data link circuit to be used for communicating with this client.
Addresses	2	LAN	Set of LANAddress	{ }	The Set of LAN addresses for this client on the circuit specified by the Circuit characteristic. These addresses may not be multicast addresses (see note below).
Secondary Loader	3		Sequence of FileSpec	( )	The files to be loaded when the client requests “Secondary Loader” during a down line load operation. The file identification is interpreted depending on the file system of the local system.
Tertiary Loader	4		Sequence of FileSpec	( )	The files to be loaded when the client requests “Tertiary Loader” during a down line load operation. The file identification is interpreted depending on the file system of the local system.
System Image	5		Sequence of FileSpec	( )	The files to be loaded when the client requests “Operating System” (i.e., Program Type = System Image, Software ID = Standard Operating System) during a down line load operation. The file identification is interpreted depending on the file system of the local system.
Diagnostic Image	6		Sequence of FileSpec	( )	The files to be loaded when the client requests “Maintenance System” (i.e., Program Type = System Image, Software ID = Maintenance System) during a down line load operation. The file identification is interpreted depending on the file system of the local system.

Table 5: Client entity characteristics attributes (part 2)

Keyword	Code	Note	Syntax	Default Value	Description
Management Image	7		Sequence of FileSpec	( )	The files to be loaded when the client requests “Management Image” during a down line load operation. The file identification is interpreted depending on the file system of the local system.
Script File	8		Sequence of FileSpec	( )	The files to be loaded when the client requests “CMIP Initialization Script” during a down line load operation. The file identification is interpreted depending on the file system of the local system.
Phase IV Host Name	9		P4Name	“”	The host name that the client receives when it is down line loaded.
Phase IV Host Address	10		P4Address	0.0	The host address that the client receives when it is down line loaded.
Phase IV Client Name	11		P4Name	“”	The client name that the client receives when it is down line loaded.
Phase IV Client Address	12		P4Address	0.0	The client address that the client receives when it is down line loaded.
Dump File	13		Sequence of FileSpec	( )	The identification of the files to write to when the client is up line dumped. The file identification is interpreted depending on the file system of the local system.
Dump Address	14		Integer	0	The memory address at which to begin an up line dump
Verification	16		OctetString	%x0000000000000000	The verification string to be sent in a Boot message to this client. See note below.
Device Types	17		SET OF DevType	{ }	The Set of Device Types for this client on the circuit specified by the Circuit characteristic.

Note:

The address values in the Addresses parameters must be individual addresses, not group (multicast) addresses. The length of the Verification attribute must be ≤8. Attempts to set invalid values in these attribute are rejected with the CMIP standard error Invalid Attribute Value.

3.6.4 Status Attributes

The Client subentity does not have any status attributes.

3.6.5 Counter Attributes

The Client subentity does not have any counter attributes.

3.6.6 Event Reports

The Client subentity does not generate any event reports.

3.7 Circuit subentity

The Circuit subentity is registered in the Distributed Systems Management Registry as shown below.

Class Name: Circuit
Code: 1
Superior Class: MOP

3.7.1 Action Directives

The Circuit subentity provides action directives for creating and deleting a subentity, and for enabling and disabling the various MOP functions. These directives and the procedures they call are described below.

3.7.1.1 Create Directive

The Create directive creates a Circuit subentity with the specified identifier. The Characteristics initially take their default values. The Modula-2+ description follows:

```
DIRECTIVE Create = 0 : Create;              (* Create a new Circuit subentity *)
    REQUEST
        ARGUMENT Type = 0 : CircuitType;
    END;
    RESPONSE Success = 0 :                  (* No Output Arguments *)
    END Success;
    EXCEPTION Unsupported Circuit Type = 1 : (* The specified Type argument value is not supported in
                                            this implementation *)
    END Unsupported Circuit Type;
    EXCEPTION Already Exists = 2 :          (* Circuit subentity already exists *)
    END Already Exists;
END Create;

PROCEDURE Create (Identifier: SimpleName, Type: CircuitType)
    RAISES {InsufficientResources,          (* CMIP common error *)
        UnsupportedCircuitType};
```

3.7.1.2 Delete Directive

The Delete directive deletes a circuit subentity and reclaims any associated resources. The Modula-2+ description follows:

```
DIRECTIVE Delete = 1 : Delete;          (* Delete an Circuit subentity *)
    REQUEST                             (* No Input Arguments *)
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;                        (* No Entity-specific Exceptions *)
    EXCEPTION Has Children = 1 :        (* Subentities must be deleted first *)
    END Has Children;
END Delete;

PROCEDURE Delete ( Identifier )
    RAISES {};
```

3.7.1.3 Enable Directive

The Enable directive enables a particular MOP function for the specified circuit. The function to be enabled is supplied as an argument. The Modula-2+ description follows:

```
DIRECTIVE Enable = 2 : Enable;          (* Enable a Circuit subentity function *)
    REQUEST
        ARGUMENT Functions = 0 : FunType;
    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;
    EXCEPTION Unsupported Function = 0 :        (* The specified function is not supported, either not
                                        at all by the implementation, or not for this particular data
                                        link type. *)
    END Unsupported Function;
    EXCEPTION Nonexistent Data Link = 1 :       (* The specified Data Link entity does not exist *)
    END Nonexistent Data Link;
    EXCEPTION Open Port Failed = 2 :    (* The Open Port operation failed *)
    END Open Port Failed;
END Enable;

PROCEDURE Enable (Function: FunType)
    RAISES {Unsupported, NonexistentDataLink, OpenPortFailed};
```

3.7.1.4 Disable Directive

The Disable directive disables a particular MOP function for the specified circuit. The function to be disabled is supplied as an argument. The argument is optional; the default is to disable all functions (i.e., all that are currently enabled). The Modula-2+ description follows:

```
DIRECTIVE Disable = 3 : Disable;        (* Disable a Circuit subentity function *)
    REQUEST
        ARGUMENT Functions = 0 : FunType;
            DEFAULT = { Loop Requester, Console Requester, Console Carrier, Load Requester,
                                        Load Server, Dump Requester, Dump Server,
                                        Configuration Monitor, Test Requester, Query Requester }

    END;
    RESPONSE Success = 0 :              (* No Output Arguments *)
    END Success;                        (* No Entity-specific Exceptions *)
END Disable;

PROCEDURE Disable (Function: FunType)
    RAISES {};
```

3.7.1.5 Loop Directive

The Loop directive causes a loop test to be performed with another system.

```
DIRECTIVE Loop = 4 : Loop;                (* Perform a Loop operation with another system *)
    REQUEST
        ARGUMENT Client = 0 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Assistant System = 4 : SimpleName;
        ARGUMENT Assistant Address = 5 : LANAddress;
        ARGUMENT Assistance Type = 6 : HelpType;
        ARGUMENT Count = 7 : Integer;
        ARGUMENT Length = 8 : Integer;
        ARGUMENT Format = 11 : Octet
    END;
    RESPONSE Success = 0 :                (* No output arguments *)
    END Success;
    RESPONSE Invalid Response = 1 :
        ARGUMENT Count = 7 : Integer;     (* Count of messages successfully looped *)
    END Invalid Response;
    EXCEPTION Data Link Error = 2 :       (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Client = 3 :   (* There is no client with the specified identification *)
    END Unrecognized Client;
    EXCEPTION Unrecognized Assistant = 5 : (* No assistant with the specified identification *)
    END Unrecognized Assistant;
    EXCEPTION Invalid Assistant = 6 :     (* The Assistant Address is either a multicast address, or
                                          Assistant System was specified but the corresponding Client
                                          subentity has an empty Addresses list *)
    END Invalid Assistant;
    EXCEPTION Timeout = 7 :               (* No response was received in the timeout period *)
    END Timeout;
END Loop;

PROCEDURE Loop (Client, Circuit, Address, AssistantSystem,
        AssistantAddress, AssistanceType, Count, Length, Format)
    RAISES {InvalidResponse,
        InsufficientResources,            (* CMIP common error *)
        UnrecognizedCircuit,              (* Does not apply here *)
        DataLinkError,
        UnrecognizedClient,
        RequiredArgumentOmitted,          (* CMIP common error *)
        UnrecognizedAssistant,
        InvalidAssistant,
        InvalidArgumentValue,             (* CMIP common error *)
        Timeout};
```

For LAN circuits, if Assistance Type is None, the operation is a Loop-direct; If Assistance Type is some other value, the operation is Loop-assisted. For Loop-assisted, the default assistant address is the Loop Assistant multicast address. A different address may be specified by including the Assistant Address parameter, which specifies the address directly, or by specifying Assistant System, which specifies a name of a Client subentity. The Addresses characteristic of the named Client subentity is then used as the assistant address. On non-LAN circuits, the Assistant Address, Assistant System, and Assistance Type arguments are ignored.

The required parameters are:

- Address (for LAN circuits)

The defaults values for the optional parameters are:

- Assistance Type — None
- Assistant Address — Obtained dynamically: an assistant is found by sending a direct (non-assisted) Loop request to the Loop Assistance multicast address. If a response is received, the responding station is then used as the assistant for the rest of the operation.
- Count — 1
- Length — 40
- Format — %x55

3.7.1.6 Load Directive

The Load directive causes a down line load to be performed to another system. A Boot is done first to force the specified system to load.

```
DIRECTIVE Load = 5 : Load;                (* Force load an adjacent system *)
    REQUEST
        ARGUMENT Client = 0 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT System Image = 3 : FileSpec;
        ARGUMENT Script File = 4 : FileSpec;
        ARGUMENT Secondary Loader = 5 : FileSpec;
        ARGUMENT Tertiary Loader = 6 : FileSpec;
        ARGUMENT Verification = 7 : OctetString;
        ARGUMENT Management Image = 8 : FileSpec;
    END;
    RESPONSE Success = 0 :                (* No Output Arguments *)
    END Success;
    EXCEPTION Data Link Error = 2 :       (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Client = 3 :   (* There is no client with the specified identification *)
    END Unrecognized Client;
    EXCEPTION Protocol Error = 5;         (* A protocol error occurred during the load *)
    END Protocol Error;
    EXCEPTION Timeout = 7 :               (* No response was received in the timeout period *)
    END Timeout;
END Load;

PROCEDURE Load (Client, Circuit, Address, LoadFile, ScriptFile, ManagementImage,
        SecondaryLoader, TertiaryLoader, Verification)
    RAISES {InsufficientResources,        (* CMIP common error *)
        UnrecognizedCircuit,              (* Does not apply here *)
        DataLinkError,
        UnrecognizedClient,
        RequiredArgumentOmitted,          (* CMIP common error *)
        ProtocolError,
        InvalidArgumentValue,             (* CMIP common error *)
        Timeout};
```

Required parameters are:

- Address (for LAN circuits)
- System Image
- Script File (if requested by target system)
- Secondary Loader (if requested by target system)
- Tertiary Loader (if requested by target system)
- Management Image (if requested by target system)

The default values for the optional parameters are:

- Verification — %x0000000000000000

3.7.1.7 Boot directive

The Boot directive causes a Boot message to be sent to another system.

```
DIRECTIVE Boot = 6 : Boot;                 (* Send a Boot trigger to an adjacent system *)
    REQUEST
        ARGUMENT Client = 0 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Verification = 7 : OctetString;
        ARGUMENT Device = 8 : String;
        ARGUMENT Software ID = 9 : SoftwareID;
        ARGUMENT Script ID = 10 : SoftwareID;
    END;
    RESPONSE Success = 0 :                 (* No Output Arguments *)
    END Success;
    EXCEPTION Data Link Error = 2 :        (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Client = 3 :    (* There is no client with the specified identification *)
    END Unrecognized Client;
END Boot;

PROCEDURE Boot (Client, Circuit, Address, Verification, Device, SoftwareID,  ScriptID)
    RAISES {InsufficientResources,         (* CMIP common error *)
        UnrecognizedCircuit,               (* Does not apply here *)
        DataLinkError,
        UnrecognizedClient,
        InvalidArgumentValue,              (* CMIP common error *)
        RequiredArgumentOmitted};          (* CMIP common error *)
```

Required parameters are:

- Address (for LAN circuits)

The default values for the optional parameters are:

- Verification — %x0000000000000000

Note:

The default value for the Verification parameter is a default only. It is not a “wild card” value. For any value of the Verification parameter including the default value, the Console Server (target of the directive) will check the value and accept the operation only if the value matches the one expected.

3.7.1.8 Test Directive

The Test directive causes the IEEE 802.2 Test to be performed with another system. If applied to the wrong type of data link (i.e., non-LAN), the standard error Directive Not Supported is returned.

```
DIRECTIVE Test = 7 : Test;               (* Perform a Test operation with another system *)
    REQUEST
        ARGUMENT Client = 0 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT Count = 7 : Integer;
        ARGUMENT Length = 8 : Integer;
        ARGUMENT SAP = 10 : SAPAddress;
        ARGUMENT Format = 11 : Octet;
    END;
    RESPONSE Success = 0 :               (* No output arguments *)
    END Success;
    RESPONSE Invalid Response = 1 :
        ARGUMENT Count = 7 : Integer;    (* Count of messages successfully looped *)
    END Invalid Response;
    EXCEPTION Data Link Error = 2 :      (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Client = 3 :  (* There is no client with the specified identification *)
    END Unrecognized Client;
    EXCEPTION Timeout = 7 :              (* No response was received in the timeout period *)
    END Timeout;
END Test;

PROCEDURE Test (Client, Circuit, Address, Count, Length, SAP, Format)
    RAISES {InvalidResponse,
        InsufficientResources,           (* CMIP common error *)
        UnrecognizedCircuit,             (* Does not apply here *)
        DataLinkError,
        UnrecognizedClient,
        RequiredArgumentOmitted,         (* CMIP common error *)
        DirectiveNotSupported,           (* CMIP common error *)
        InvalidArgumentValue,            (* CMIP common error *)
        Timeout);
```

The required parameters are:

- Address

The defaults values for the optional parameters are:

- Count — 1
- Length — 40
- SAP — 0
- Format — %x55

3.7.1.9 Query Directive

The Query directive causes an IEEE 802.2 XID exchange to be performed with another system. If applied to the wrong type of data link (i.e., non-LAN), the standard error Directive Not Supported is returned.

```
DIRECTIVE Query = 8 : Query;                (* Perform an XID exchange with another system *)
    REQUEST
        ARGUMENT Client = 0 : SimpleName;
        ARGUMENT Address = 2 : LANAddress;
        ARGUMENT SAP = 10 : SAPAddress;
    END;
    RESPONSE Success = 0 :
        ARGUMENT LLC Types = 10 : Set of Integer; (* LLC types supported by system *)
        ARGUMENT Window = 11 : Integer  (* Window size for Type 2 LLC *)
    END Success;
    RESPONSE Invalid Response = 1 :         (* Protocol error in response *)
    END Invalid Response;
    EXCEPTION Data Link Error = 2 :         (* An error was reported by the Data Link layer *)
        ARGUMENT Reason = 0 : DLErrType;
    END Data Link Error;
    EXCEPTION Unrecognized Client = 3 :  (* There is no client with the specified identification *)
    END Unrecognized Client;
    EXCEPTION Timeout = 7 :                 (* No response was received in the timeout period *)
    END Timeout;
END Query;

PROCEDURE Query (Client, Circuit, Address, SAP, LLCTypes, Window)
    RAISES {InvalidResponse,
        InsufficientResources,              (* CMIP common error *)
        UnrecognizedCircuit,                (* Does not apply here *)
        DataLinkError,
        UnrecognizedClient,
        RequiredArgumentOmitted,            (* CMIP common error *)
        DirectiveNotSupported,              (* CMIP common error *)
        InvalidArgumentValue,               (* CMIP common error *)
        Timeout);
```

The required parameters are:

- Address

The defaults values for the optional parameters are:

- SAP — 0

3.7.2 Identifier Attribute

The identifier attributes for each Circuit subentity are shown in Table 6.

Table 6: Circuit entity identifier attribute

Keyword	Code	Syntax	Description
Name	0	SimpleName	A string which is the Identifier for the Circuit and which is unique among the set of Circuits defined for the MOP module.

3.7.3 Characteristic Attributes

The characteristics attributes for each Circuit subentity are shown in Table 7.

Table 7: Circuit entity characteristics attributes

Keyword	Code	Syntax	Default Value	Description
Type	1	CircuitType		The type of the Circuit, set when the Circuit subentity is created. This is a read-only characteristic.
Link Name	2	Local Entity Name	“”	The name of the Station subentity in the Data Link layer module indicated by the Type characteristic. This is passed to the Data Link layer module when MOP opens a port for the circuit.
Retransmit Timer	3	Integer [1..30]	4	The time-out value to use (in seconds) for retransmitting MOP protocol messages when no response is received.
Known Clients Only	13	Boolean	FALSE	TRUE if Load service is to be limited only to clients listed in the Client database, even if a Software ID string is supplied by the client.

3.7.4 Status Attributes

The status attributes for each Circuit subentity are shown in Table 8.

Table 8: Circuit entity status attribute

Keyword	Code	Syntax	Description
UID	4	UID	The UID of this entity, allocated when it was created.
Functions	6	FunType	The set of functions currently enabled for this circuit. Only the functions that can be enabled and disabled are shown; functions required to be active at all times are not.

3.7.5 Counter Attributes

The counter attributes for each Circuit subentity are shown in Table 9.

Table 9: Circuit entity counter attributes

Keyword	Code	Syntax	Description
Creation Time	5	Binary Absolute Time	The time at which the entity was created. This indicates the time at which the associated Counter attributes were zero.
Load Requests Completed	7	Counter	The number of Load service requests completed successfully.
Unrecognized Load Clients	8	Counter	The number of Load service requests that could not be processed because a Client database entry was required but no matching entry was found.
Failed Load Requests	9	Counter	The number of Load service requests that could not be completed.
Dump Requests Completed	10	Counter	The number of Dump service requests completed successfully.
Unrecognized Dump Clients	11	Counter	The number of Dump service requests that could not be processed because a Client database entry was required but no matching entry was found.
Failed Dump Requests	12	Counter	The number of Dump service requests that could not be completed.

3.7.6 Event Reports

The Circuit subentity generates six events. The event arguments are described in Table 10.

Table 10: Circuit entity event arguments

Keyword	Code	Syntax	Description
Reason	1	SvErrType	The reason for the operation failure.
Address	2	LANAddress	The Data Link address of the remote system. Supplied only for LAN Data Links.
Client	4	SimpleName	The name of the Client subentity corresponding to the remote system. If unavailable, this argument is omitted.
Program Type	5	ProgType	The Program Type requested by the remote system. Applicable only for Load requests; omitted otherwise.
File	6	FileSpec	The file name of the file being processed. For Load requests, this is the file being loaded. For Dump requests, this is the Dump File name.

The events definitions are as follows:

```
EVENT Load Request Completed = 0;          (* A Load service requested by a remote Dump/Load
                                           requester has completed successfully *)
    COUNTED AS Load Requests Completed;
    ARGUMENTS Address, Client, Program Type, File;
END Load Request Completed;
EVENT Unrecognized Load Client = 1;        (* A Load service requested by a remote Dump/Load
                                           requester was not accepted because a Client database entry
                                           was needed but no matching entry was found *)
    COUNTED AS Unrecognized Load Clients;
    ARGUMENTS Address, Program Type;
END Unrecognized Load Client;
EVENT Load Request Failed = 2;             (* A Load requested by a remote Dump/Load requester has
                                           failed.  Note that the case of unrecognized client (where the
                                           request was never started due to insufficient information) is
                                           reported using the event above.  *)
    COUNTED AS Failed Load Requests;
    ARGUMENTS Reason, Address, Client, Program Type, File;
END Load Request Failed;
EVENT Dump Request Completed = 3;          (* A Dump service requested by a remote Dump/Load
                                           requester has completed successfully *)
    COUNTED AS Dump Requests Completed;
    ARGUMENTS Address, Client, File;
END Dump Request Completed;
EVENT Unrecognized Dump Client = 4;        (* A Dump service requested by a remote Dump/Load
                                           requester was not accepted because a Client database entry
                                           was needed but no matching entry was found *)
    COUNTED AS Unrecognized Dump Clients;
    ARGUMENTS Address;
END Unrecognized Dump Client;
EVENT Dump Request Failed = 5;             (* A Dump requested by a remote Dump/Load requester has
                                           failed.  Note that the case of unrecognized client (where the
                                           request was never started due to insufficient information) is
                                           reported using the event above.  *)
    COUNTED AS Failed Dump Requests;
    ARGUMENTS Reason, Address, Client, File;
END Dump Request Failed;
```

3.8 Operation subentity

The Operation subentity is registered in the Distributed Systems Management Registry as shown below.

Class Name: Operation
Code: 0
Superior Class: Circuit

3.8.1 Action Directives

The Operation subentity does not have any action directives.

3.8.2 Identifier Attribute

The identifier attributes for each Operation subentity are shown in Table 11.

Table 11: Operation entity identifier attribute

Keyword	Code	Syntax	Description
Name	0	SimpleName	A string which identifies the operation and is unique among the set of Operation subentities for this circuit. The suggested way to generate this identifier is to use the external (user visible) representation of the Data Link address for the client system, with a suffix to distinguish it from other concurrent operations involving the same system.

3.8.3 Characteristic Attributes

The Operation subentity does not have any characteristic attributes.

3.8.4 Status Attributes

The status attributes for each Operation subentity are shown in Table 12.

Table 12: Operation entity status attributes

Keyword	Code	Syntax	Description
Operation	1	ServType	The operation being performed.
Address	2	LANAddress	The Data Link address for the client system.
Client	3	SimpleName	The Client name of the client system, if available. If a Client Database reference was necessary to process the operation, this attribute will be non-null. If not (i.e., all required parameters were supplied) the implementation may optionally look for a matching Client Database entry anyway and provide the Client name, if a match was found. Otherwise, this attribute will be a null Simple Name.

3.8.5 Counter Attributes

The Operation subentity does not have any counter attributes.

3.8.6 Event Reports

The Operation subentity does not generate any event reports.

3.9 Station subentity

The Station subentity is registered in the Distributed Systems Management Registry as shown below.

Class Name: Station

Code: 1
Superior Class: Circuit

The Station subentity is created by the Configuration Monitor component, to report stations observed on the network.

3.9.1 Action Directives

The Station subentity does not have any action directives.

3.9.2 Identifier Attribute

The identifier attributes for each station subentity are shown in Table 13.

Table 13: Station entity identifier attribute

Keyword	Code	Syntax	Description
Name	0	SimpleName	A string which identifies the station and is unique among the set of Station subentities for this circuit. The suggested way to generate this identifier is to use the external (user visible) representation of the Data Link address for this station, i.e., the source address of the System ID message from which this Station subentity was created.

3.9.3 Characteristic Attributes

The Station subentity does not have any characteristic attributes.

3.9.4 Status Attributes

The status attributes for each Station subentity are shown in Table 14.

Table 14: Station entity status attributes

Keyword	Code	Syntax	Description
Last Report	1	Binary Absolute Time	The time at which the most recent System ID message was received from this system.
MOP Version	2	Version	The MOP Protocol version received from the system.
Functions	3	SIDFunType	The set of functions implemented by this system. Note that the "Console Carrier Reserved" bit from the System ID message is not reflected here, but rather in the Console User status.
Console User	4	LANAddress	The Data Link address of the system that currently has the console reserved, or the Null address if the console is not reserved.
Reservation Timer	5	Integer	The Console reservation timer, in seconds, or zero if not applicable.
Command Size	6	Integer	The maximum Console command size, in bytes, or zero if not applicable.
Response Size	7	Integer	The maximum Console response size, in bytes, or zero if not applicable.
Hardware Address	8	LANAddress	The hardware address (default data link address) for the circuit on which the System ID message was sent by the system.
Device Type	9	DevType	The communication device type for the circuit on which the System ID was sent by the system. Refer to Appendix A.1 for the current list of codes assigned for this data type.
Data Link	10	CircuitType	The data link protocol for the circuit on which the System ID was sent by the system. The code assignments for type data type are shown in Section 3.4.
DSDU Size	11	Integer	The DSDU size for the circuit on which the System ID was sent by the system. That is, the maximum length of the message passed between MOP and the Data Link layer module, excluding any Data Link envelope.
Node Name	13	FullName	The node name of the node. Null if not reported or if reported as null by the node.
Node ID	14	LANAddress	The node ID of the node, or the null address (00–00–00–00–00–00) if not reported by the remote node.

3.9.5 Counter Attributes

The Station subentity does not have any counter attributes.

3.9.6 Event Reports

The Station subentity does not generate any event reports.

Chapter 4

Operation

This chapter presents the MOP algorithms in detail. The description is mostly pseudo-code in Modula-2-Plus, with some portions described in English where this is clearer. The structure of the algorithms is intended to facilitate understanding; there is no requirement that actual implementations structure their algorithms in a similar way, only that the externally visible behavior (i.e., protocol messages, network management output) is the same.

The algorithms shown include support for compatibility with MOP V3. Refer to Appendix G for more information.

The definitions and algorithms in this chapter are organized as follows:

- Common definitions: constants, datatypes, etc.
- Top level management directive processing
- Top level algorithms for the MOP client and server functions
- Common low level protocol primitives
- Data link dependent protocol primitives
- Miscellaneous low level procedures

4.1 Notes on the operational model

As can be seen from the MOP component model (as shown in Chapter 2), MOP consists of a number of fairly independent protocol entities. In addition, many of these components may be engaged in more than one operation at a time.

In a practical implementation, MOP will need to have a number of threads for its various functions (and, in cases such as load/dump servers, multiple threads for multiple instances of the same function).

The algorithmic description in this chapter glosses over some detail in an attempt to reduce the amount of unnecessary clutter. These include:

- Thread creation, scheduling, and deletion is not shown. However, the `Operation` record type declaration does show (approximately) the per-thread state needed to describe each operation thread.

- The description of each thread assumes that there is a receive dispatcher that sorts out the received messages and passes to each thread only those messages that concern it (see Section 4.12).

- Parsing and validation of incoming messages is generally not shown. The general rule is that any unexpected or invalid message should be treated as if it had not been received at all. Note that this means that timers and retry counters should not be reset when such a message is received. In a few cases this rule is called out explicitly with the pesudo-statement « quietly ignore the message ».

  Note:

  Note that this means that unexpected, erroneous, malformed, or otherwise illegal messages are normally *not* cause for aborting the operation with a "Protocol error" result. The few exceptions to this rule are called out explicitly in the algorithms below.

- In a number of cases, the algorithms have to attempt request/response exchanges with several possible addresses and/or version numbers. The algorithms below try each combination sequentially. An implementation may instead do some or all of the possible combination in parallel for efficiency. For some protocol exchanges (e.g., Loop) care must be taken not to confuse the replies; the receipt number in the message can be used to do this.

4.2 Common definitions

4.2.1 Architectural constants

Table 15 below lists the MOP architectural constants with the name by which they are referred to in this chapter.

Table 15: MOP Architectural Constants

Retransmit Max	15	Maximum number of retries before giving up, for the "Transact" operation
Burst Size	4	Number of retries per "burst" in the BackoffTransact operation
Max Backoff	300	Maximum value of the backoff timer, in seconds (i.e., 5 minutes)
Load Protocol Type	60–01	Ethernet Protocol Type for Load and Dump protocol
Console Protocol Type	60–02	Ethernet Protocol Type for Remote Console protocol
Loopback Protocol Type	90–00	Ethernet Protocol Type for Loopback protocol
Load Protocol ID	08–00–2B–60–01	IEEE 802.2 SNAP Protocol ID for Load and Dump protocol
Console Protocol ID	08–00–2B–60–02	IEEE 802.2 SNAP Protocol ID for Remote Console protocol
Loopback Protocol ID	08–00–2B–90–00	IEEE 802.2 SNAP Protocol ID for Loopback protocol
Load Assistance	AB–00–00–01–00–00	Assistance multicast address for Load and Dump protocol
Console Multicast	AB–00–00–02–00–00	Multicast address for Remote Console protocol (for periodic SYSID announcements)
Loopback Assistance	CF–00–00–00–00–00	Assistance multicast address for Loopback protocol
MOP Protocol ID	01–01	HDLC Protocol ID for MOP messages

4.2.2 MOP module data type definitions

The following are definitions of the data types used in this chapter. Since Modula-2-Plus does not have a "Sequence of" type constructor (as in CMIP), the ARRAY OF constructor is used instead when needed.

```
FROM System IMPORT Port, Timer, Time, Byte;
FROM System IMPORT Random;          (* Function to return a pseudo-random number 0<x<1 *)
FROM System IMPORT Wait;            (* Procedure to wait for some amount of time *)
FROM System IMPORT StartTimer, StopTimer, Expired;
FROM Management IMPORT BinaryAbsoluteTime, LocalEntityName, P4Name, P4Address,
    FileSpec, LANAddress, SAPAddress;
FROM Names IMPORT SimpleName, FullName;
FROM CSMACD IMPORT FormatAndMux, LLCService;

TYPE MOPVersion = (V3, V4, Unknown);
CONST Infinite = « a time-out value to wait indefinitely »;
    FiveMinutes = « a time-out value equal to 5 minutes »;
    OneMinute = « a time-out value equal to 1 minute »;
    NullAddress = « the null (48 bits of zero) LANAddress »;
    LANTypes = { CSMACD, FDDI };    (* The set of Circuit Data Link types that are LANs *)
    LLCTEST = 0E3H;                 (* LLC Ctrl field code for TEST message *)
```

```
     LLCXID = 0AFH;                           (* LLC Ctrl field code for XID message *)

TYPE Buffer = RECORD
     LANFormat: FormatAndMux;                 (* LLC format and SAP *)
     Destination, Source: LANAddress;         (* LAN address fields *)
     Length: INTEGER;
     Data: ARRAY [0..length–1] OF Byte
END;

TYPE Circuit = RECORD
     Name: SimpleName;                        (* Identifier of this Circuit entity *)
     DataLink: CircuitType;                   (* Type of data link *)
     LinkName: LocalEntityName;               (* Name of data link entity used *)
     DataLinkAddress: LANAddress;             (* This circuit's LAN address *)
     SDUSize: INTEGER;                        (* Datalink SDU size (MOP PDU size)[1] *)
     RetransmitTimer: Integer;
     KnownClientsOnly: Boolean;               (* TRUE if only serving registered clients *)
     CircUID: UID.UID;
     CreateTime: BinaryAbsoluteTime;          (* Creation time of this Circuit entity *)
     Functions: Funtype;                      (* Enabled MOP functions *)
     ConsoleUser: LANAddress;                 (* Console user — if Console Carrier supported *)
     ReservationTimer: Timer;                 (* Console reservation timer — if C.C. supported *)
     ConsoleSeq: INTEGER;                     (* Console Command sequence number *)
     LoadRequestsCompleted, UnrecognizedLoadClients,
          FailedLoadRequests, DumpRequestsCompleted,
          UnrecognizedDumpRequests, FailedDumpRequests: Counter
     mopport, loopport, LLCport: POINTER TO Port;      (* Data link ports *)
END;

TYPE Client = RECORD
     Name: SimpleName;                        (* Identifier of this client entity *)
     CircPtr: POINTER TO Circuit;             (* Circuit used for this client[2] *)
     Addresses: SET OF LANAddress;            (* Set of client data link addresses *)
     DevTypes: SET OF DevType;                (* Set of client device types *)
     SecondaryLoader, TertiaryLoader,
          SystemImage, DiagnosticImage,
          ManagementImage, ScriptFile: ARRAY OF FileSpec;
                                              (* Actually, SEQUENCE *)
     PhaseIVHostName: P4Name;                 (* Four MOP V3 style parameter values *)
     PhaseIVHostAddress: P4Address;
     PhaseIVClientName: P4Name;
     PhaseIVClientAddress: P4Address;
     DumpFile: ARRAY OF FileSpec;             (* Actually, SEQUENCE OF FileSpec *)
     DumpAddress: INTEGER;
     Verification: OctetString;
END;

TYPE Operation = RECORD
     Name: SimpleName;                        (* Identifier of this Operation entity *)
     Addresses: SET OF LANAddress;            (* Set of client data link addresses to try *)
     Address: LANAddress;                     (* Address of remote station, if on LAN *)
     ClientName, CircName: SimpleName;        (* Name of Client and Circuit to use *)
     CircPtr: POINTER TO Circuit;             (* Circuit being used for this operation *)
```

1 In data links where the maximum SDU sizes differ for transmit and receive, the smaller of the two values is used.

2 The POINTER type is used here for descriptive convenience in the pseudocode. Note that the network management interface uses a SimpleName to refer to the Circuit; this allows Client and Circuit entities to be created in any order.

```
    ClientPtr: POINTER TO Client;          (* Client entity for this operation, if known *)
    Version: MOPVersion;                   (* Protocol version used by target *)
    Timeout: Timer;                        (* Receive timer for this operation *)
    CASE Op: OpType OF                     (* Operation being performed *)
        LoopRequester:
            AssistantSystem: SimpleName;   (* Name of assistant (i.e., of its Client entity) *)
            AssistantAddresses: SET OF LANAddress; (* Set of assistant data link addresses to try *)
            AsisstantAddress: LANAddress;  (* Selected data link address of assistant *)
            AssistanceType: HelpType;      (* Type of assistance to use, if any *)
            Count, Length: INTEGER;        (* Number of messages and length of each *)
            Data: [0..255] |               (* Byte value to load buffer with *)
        LoopServer: |                      (* Nothing special *)
        ConsoleRequester:                  (* For "Boot" directive *)
            Verification: OctetString;     (* Boot verification string ("password") *)
            Device: String;                (* Device name, if booting "specified device" *)
            SoftwareID,
                ScriptID: SoftwareID |     (* Value to use for Software ID of System and Script *)
        ConsoleServer: |                   (* Nothing special *)
        LoadRequester:
            ProgramType: ProgType;         (* Type of program being loaded *)
            Sequence: [0..255] |           (* Next load sequence number wanted *)
        LoadServer:
            ProgramType: ProgType;         (* Type of program requested by client *)
            SecondaryLoader, TertiaryLoader,
                SystemImage, DiagnosticImage,
                ManagementImage, ScriptFile: ARRAY OF FileSpec;
                                           (* Actually, SEQUENCE *)
            Verification: OctetString;     (* Boot verification string ("password") *)
            SDUSize: Integer |             (* MOP receive buffer size supported by station *)
        DumpRequester: |                   (* Nothing specific to dump client *)
        DumpServer:
            DumpFile: ARRAY OF FileSpec;   (* Actually, SEQUENCE *)
            DumpAddress: INTEGER;          (* Address to dump next *)
            DumpCount: INTEGER;            (* Amount left to dump (memory size) *)
            SDUSize: Integer |             (* MOP receive buffer size supported by station *)
        ConfigurationMonitor: |            (* Nothing specific to the configuration monitor *)
        TestRequester:
            Count, Length: INTEGER;        (* Number of messages and length of each *)
            Data: [0..255];                (* Byte value to load buffer with *)
            SAP: SAPAddress |              (* Destination SAP address to send to *)
        QueryRequester:
            SAP: SAPAddress                (* Destination SAP address to send to *)
    END
END;

TYPE Station = RECORD
    Name: SimpleName;                      (* Identifier of this Station entity *)
    LastReport: BinaryAbsoluteTime;        (* Time last heard from station *)
    Version: MOPVersion;                   (* Version of the station *)
    Functions: SIDFunType;                 (* Supported functions *)
    ConsoleUser: LANAddress;               (* Console user, or null address if none *)
    ReservationTimer, CommandSize,
        ResponseSize: INTEGER
    HardwareAddress: LANAddress;           (* ROM Address for the station *)
    Device: DevType;                       (* Device type of the station *)
    DataLink: CircuitType;                 (* Type of data link *)
    DSDUSize: Integer;                     (* MOP receive buffer size supported by station *)
    NodeName: FullName;                    (* Name of the node, if known *)
```

```
    NodeID: LANAddress;              (* Node ID of station, if known *)
    CircPtr: POINTER TO Circuit;     (* Circuit to which this Station record applies *)
END;
```

4.2.3 Message type definitions

```
CONST MemLoadTA = 0; DumpComplete = 1;
    MemLoad = 2; Assistance = 3;
    RequestDump = 4; RequestID = 5; Boot = 6; SysID = 7;
    RequestProgram = 8; RequestCounters = 9;
    RequestLoad = 10; CSMACDCounters = 11;
    RequestDumpService = 12; ReserveConsole = 13;
    DumpData = 14; ReleaseConsole = 15;
    ConsoleCommand = 17; ConsoleResponse = 19;
    ParameterLoad = 20; OtherCounters = 21;
    LoopData = 24; LoopedData = 26;      (* These two codes are for point to point circuits *)
    Reply = 1; ForwardData = 2;          (* These two codes are for LAN loop protocol *)
```

4.3 Use of the Client database

Client database records contain parameters that MOP will use to satisfy requests related to a particular client. In general, request parameters can be supplied with the request. In the case of requests that are management directives, it is possible to supply all the needed parameters as directive arguments. In the case of requests that arrive via MOP protocol messages (e.g., Dump requests) the protocol message may not be able to encode all the necessary information. In either case, the requester may omit some of the parameters; such omitted parameters are filled in from the client database.

Management directives may refer to a Client by name. In that case, the Client subentity with the supplied name is looked up, and its attributes are used as defaults for the directive parameters. Any parameters explicitly supplied with the directive override the defaults from the Client entry.

If a management directive does not specify a Client name, no reference to the Client database is made and all required parameters must be included with the directive.

Dump protocol requests always require a search of the Client database; Load protocol requests also require a search, unless the request includes a Software ID string and the Circuit attribute Known Clients Only is False. This is a search by circuit (see Section 4.18.3). If a matching Client entry is found, its parameters are used as defaults for the processing of the request. A Software ID string in a Request Program message, if present, overrides the file names in the client entry. If no entry is found, the result depends on the request:

- A Dump request from an unknown client generates an Unrecognized Dump Client event, and is otherwise ignored.
- For a Load request, the result also depends on the Known Clients Only attribute of the Circuit:
    - If Known Clients Only is True, then a Load request from an unknown client generates an Unrecognized Load Client event, and is otherwise ignored.
    - Otherwise, if the Request Program message contains a Software ID string, the load server maps that string to a file name and attempts to open the file. If the open fails due to “File not found”, this generates an Unrecognized Load Client event, and the load request is ignored.

- Otherwise (no Software ID string included in the Request Program message), this generates an Unrecognized Load Client event, and the load request is ignored.

4.4 Processing of management directives

This section defines the operation of the action directives which are invoked via the management interface defined in Chapter 3.

In the algorithms below, directive parameters that are not supplied with the directive are defaulted from the Client database, if a Client name was supplied with the directive. If the Client name supplied with the directive does not match the name of an entry in the Client database, the exception UnrecognizedClient is generated.

4.4.1 Loop directive

The Loop directive is processed as follows:

- Get defaulted Circuit and Address from the Client database, if a Client name was given.
- Get defaulted Assistant Address from the Client database, if an Assistant Name was given.
- On a LAN, if assistance was specified, select an assistant address by trying a direct loop to each of the assistant addresses. (If no specific assistant was specified, the loop is done to the Loopback Assistance multicast address.) The source address from the first received reply is the assistant address.
- Perform the Loop exchange Count times. If the response has the wrong data, or the exchange times out, abort the Loop operation.

  Special case: if the first exchange times out on a LAN, try the next target address. If all addresses have been tried, switch to the other MOP version format, and repeat the entire process (including assistant selection). If both versions time out for all addresses, the Loop fails with a Timeout error.

Since the purpose of a Loop operation is to look for network problems, all the request/response exchanges are done as single attempts, *not* with the usual “Transact” multiple retries. This ensures that intermittent errors are not masked.

```
PROCEDURE DLI.Loop (clientsn, circuitsn: Names.SimpleName; target: LANAddress;
        assistantsn: Names.SimpleName; assistant: LANAddress; help: HelpType;
        loopcount, looplength: INTEGER; loopdata: INTEGER)
    RAISES {InvalidResponse, InsufficientResources, UnrecognizedCircuit, DataLinkError,
        UnrecognizedClient, RequiredArgumentOmitted,
        UnrecognizedAssistant, InvalidAssistant, InvalidArgumentValue, Timeout);
VAR op: POINTER TO Operation;
    assistantptr: POINTER TO Client;
    buff: POINTER TO Buffer;
BEGIN
    DLI.Alloc (op, buff);                  (* Allocate necessary resources, if available *)
    TRY
        op^.Version := Unknown;
        op^.Op := LoopRequester;
        op^.Count := 1; op^.Length := 40; op^.Data := 55H;      (* Default values *)
        op^.AssistantAddresses := { LoopbackAssistance };       (* Default value *)
        op^.ClientPtr := NIL;                                   (* Default value *)
        IF « clientsn supplied »
```

```
THEN
    op^.ClientName := clientsn;
    op^.ClientPtr := FindClientByName (op^.ClientName)
END;
op^.CircuitName := circuitsn;          (* if supplied as directive argument *)
op^.Addresses := target;               (* if supplied as directive argument *)
IF target[0] = 1                       (* Multicast bit set in supplied target address? *)
THEN
    RAISE (InvalidArgumentValue)
END;
op^.AssistantSystem := assistantsn;    (* if supplied as directive argument *)
op^.AssistantAddresses := assistant;   (* if supplied as directive argument *)
IF assistant[0] = 1                    (* Multicast bit set in supplied assistant address? *)
THEN
    RAISE (InvalidAssistant)
END;
op^.AssistanceType := help;            (* if supplied as directive argument *)
op^.Count := loopcount;                (* if supplied as directive argument *)
op^.Length := looplength;              (* if supplied as directive argument *)
op^.Data := loopdata;                  (* if supplied as directive argument *)
IF « circuitsn not supplied »
THEN
    IF op^.ClientPtr ≠ NIL
    THEN op^.CircuitName := op^.ClientPtr^.CircPtr^.Name
    END
END;
IF op^.CircuitName = ""
THEN
    RAISE (RequiredArgumentOmitted)
END;
op^.CircPtr := FindCircuitByName (op^.CircuitName)
IF « target not supplied » AND op^.CircPtr^.DataLink IN LANTypes
THEN
    IF op^.ClientPtr ≠ NIL
    THEN op^.Addresses := op^.ClientPtr^.Addresses;
    END;
    IF op^.Addresses = { }             (* Check for no addresses in Client database *)
    THEN RAISE (RequiredArgumentOmitted)
    END
END;
IF op^.AssistantSystem ≠ ""
THEN
    TRY
        assistantptr := FindClientByName (op^.AssistantSystem)
    EXCEPT
    | UnrecognizedClient: RAISE (UnrecognizedAssistant)
    END;
    IF « assistant not supplied »
    THEN op^.AssistantAddresses := assistantptr^.Addresses;
        IF op^.AssistantAddresses = { }        (* Check for no addresses in Client database *)
        THEN RAISE (InvalidAssistant)
    END
END;
IF op^.CircPtr^.DataLink IN LANTypes
THEN
    LAN.LoopRequester (op, buff)
ELSE
    PTP.LoopRequester (op, buff)
```

```
        END
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.Loop;
```

4.4.2 Load directive

The Load directive is processed as follows:

- Get defaulted parameters from the Client database, if a Client name was given.
- Send Boot messages to the target. For a point to point circuit, one message is sent. For a LAN, two messages are sent to each address, one in each version format. The messages have Control = 1, indicating "Load server = requesting system". This instructs the target to send its Request Program message back to the node sending the Boot messages.
- Wait for a Request Program message from the target. On a LAN, the Request Program may come from any of the target addresses, in either version format. The recommended timeout is 60 seconds.
- If a Request Program message is received, proceed with load service requested by that message.

  Note: if a System Image and/or Script File argument was included in the directive, that file name overrides any Software ID string in a Request Program message for the corresponding Program Type..

```
PROCEDURE DLI.Load (clientsn, circuitsn: Names.SimpleName; target: LANAddress;
        loadfile, script, management, secondary, tertiary: FileSpec; ver: Octetstring)
    RAISES {InsufficientResources, UnrecognizedCircuit, DataLinkError, UnrecognizedClient,
        RequiredArgumentOmitted, ProtocolError, InvalidArgumentValue, Timeout};
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
BEGIN
    DLI.Alloc (op, buff);                   (* Allocate necessary resources, if available *)
    TRY
        op^.Version := Unknown;
        op^.Op := LoadServer;
        op^.ClientName := clientsn;         (* if supplied as directive argument *)
        op^.CircuitName := circuitsn;       (* if supplied as directive argument *)
        op^.Addresses := target;            (* if supplied as directive argument *)
        IF target[0] = 1                    (* Multicast bit set in supplied target address? *)
        THEN
            RAISE (InvalidArgumentValue)
        END;
        op^.SystemImage := loadfile;                (* if supplied as directive argument *)
        op^.DiagnosticImage := op^.SystemImage;     (* Use it also if target asks for diagnostic image *)
        op^.ScriptFile := script;                   (* if supplied as directive argument *)
        op^.ManagementImage := management;          (* if supplied as directive argument *)
        op^.SecondaryLoader := secondary;           (* if supplied as directive argument *)
        op^.TertiaryLoader := tertiary;     (* if supplied as directive argument *)
        op^.Verification := ver;            (* if supplied as directive argument *)
        op^.ClientPtr := NIL;                               (* Default value *)
        IF « clientsn supplied »
        THEN
            op^.ClientName := clientsn;
```

```
    op^.ClientPtr := FindClientByName (op^.ClientName)
END;
IF « circuitsn not supplied »
THEN
    IF op^.ClientPtr ≠ NIL
    THEN op^.CircuitName := op^.ClientPtr^.CircPtr^.Name
    END
END;
IF op^.CircuitName = ""
THEN
    RAISE (RequiredArgumentOmitted)
END;
op^.CircPtr := FindCircuitByName (op^.CircuitName)
IF « target not supplied » AND op^.CircPtr^.DataLink IN LANTypes
THEN
    IF op^.ClientPtr ≠ NIL
    THEN op^.Addresses := op^.ClientPtr^.Addresses;
    END;
    IF op^.Addresses = { }              (* Check for no addresses in Client database *)
    THEN RAISE (RequiredArgumentOmitted)
    END
END;
IF « loadfile not supplied »
THEN
    IF op^.ClientPtr ≠ NIL
    THEN op^.SystemImage := op^.ClientPtr^.SystemImage
        op^.DiagnosticImage := op^.ClientPtr^.SystemImage
    END;
    IF op^.SystemImage = { }            (* Check for no System Image in Client database *)
    THEN RAISE (RequiredArgumentOmitted)
    END
END;
IF « script not supplied » AND op^.ClientPtr ≠ NIL
THEN op^.ScriptFile := op^.ClientPtr^.ScriptFile
END;
IF « management not supplied » AND op^.ClientPtr ≠ NIL
THEN op^.ManagementImage := op^.ClientPtr^.ManagementImage
END;
IF « secondary not supplied » AND op^.ClientPtr ≠ NIL
THEN op^.SecondaryLoader := op^.ClientPtr^.SecondaryLoader
END;
IF « tertiary not supplied » AND op^.ClientPtr ≠ NIL
THEN op^.TertiaryLoader := op^.ClientPtr^.TertiaryLoader
END;
IF « ver not supplied » AND op^.ClientPtr ≠ NIL
THEN op^.Verification := op^.ClientPtr^.Verification
END;
DLI.SendBoot (op, buff, 1);             (* Send Boot msg with Boot Server = Requesting System *)
DLI.Receive (op, buff, OneMinute, Unknown);  (* Wait 1 minute for a Request Program message *)
op^.ProgramType := buff^.Data[3];       (* get program type from Request Program message *)
IF op^.ProgramType = System
THEN
    IF « Request Program message indicates request for Diagnostic Image »
    THEN op^.ProgramType := Diagnostic
    END
END
IF « Data Link Buffer Size present in message »
THEN op^.SDUSize := « Data Link Buffer Size from message »
```

```
        ELSE op^.SDUSize := 262
        END;
        DLI.PerformLoad (op, buff)
            (* Note that Software ID string is overridden by directive arguments if specified *)
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.Load;
```

4.4.3 Boot directive

The Boot directive is processed as follows:

- Get defaulted parameters from the Client database, if a Client name was given.
- Send Boot messages to the target. For a point to point circuit, one message is sent. For a LAN, two messages are sent to each address, one in each version format. The messages have Control = 2, indicating "Load device = specified device" if the Device argument was present on the directive, and Control = 0, indicating "Load server = system default" otherwise.

```
PROCEDURE DLI.Boot (clientsn, circuitsn: Names.SimpleName; target: LANAddress;
    ver: Octetstring; dev, swid, scrid: SoftwareID)
    RAISES {InsufficientResources, UnrecognizedCircuit, DataLinkError,
    UnrecognizedClient, RequiredArgumentOmitted, InvalidArgumentValue);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    ctrl: INTEGER;
BEGIN
    DLI.Alloc (op, buff);              (* Allocate necessary resources, if available *)
    TRY
        op^.Version := Unknown;
        op^.Op := ConsoleRequester;
        op^.ClientName := clientsn;    (* if supplied as directive argument *)
        op^.CircuitName := circuitsn;  (* if supplied as directive argument *)
        op^.Addresses := target;       (* if supplied as directive argument *)
        IF target[0] = 1               (* Multicast bit set in supplied target address? *)
        THEN
            RAISE (InvalidArgumentValue)
        END;
        op^.Verification := ver;       (* if supplied as directive argument *)
        op^.Device := dev;             (* if supplied as directive argument *)
        IF « dev argument supplied »
        THEN ctrl := 2                 (* Control = "Boot device = specified device" *)
        ELSE ctrl := 0                 (* Control = "Boot server = system default" *)
        END;
        op^.SoftwareID := swid;        (* if supplied as directive argument *)
        op^.ScriptID := scrid;         (* if supplied as directive argument *)
        IF NOT (DLI.ValidID (swid) AND DLI.ValidID (scrid))
        THEN RAISE (InvalidArgumentValue)
        END;
        IF « circuitsn not supplied »
        THEN
            IF op^.ClientPtr ≠ NIL
            THEN op^.CircuitName := op^.ClientPtr^.CircPtr^.Name
            END
        END;
        IF op^.CircuitName = ""
```

```
        THEN
            RAISE (RequiredArgumentOmitted)
        END;
        op^.CircPtr := FindCircuitByName (op^.CircuitName)
        IF « target not supplied » AND op^.CircPtr^.DataLink IN LANTypes
        THEN
            IF op^.ClientPtr ≠ NIL
            THEN op^.Addresses := op^.ClientPtr^.Addresses;
            END;
            IF op^.Addresses = { }          (* Check for no addresses in Client database *)
            THEN RAISE (RequiredArgumentOmitted)
            END
        END;
        IF « ver not supplied » AND op^.ClientPtr ≠ NIL
        THEN op^.Verification := op^.ClientPtr^.Verification
        END;
        IF LENGTH(op^.Verification) > 8
        THEN RAISE (InvalidArgumentValue)
        END;
        DLI.SendBoot (op, buff, ctrl)       (* Send Boot messages *)
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.Boot;

PROCEDURE DLI.ValidID (id: SoftwareID) : Boolean
BEGIN
    « return TRUE if id meets the Software ID validity criteria described in  Section 5.5.1 »
END DLI.ValidID;

PROCEDURE DLI.SendBoot (op: POINTER TO Operation; buff: POINTER TO Buffer;  ctrl: INTEGER);
VAR target: LANAddress;
BEGIN
    buff^.Data[0] := Boot;
    « store op^.Verification in buffer data bytes 1 through 8, pad with trailing  0 bytes if necessary »;
    buff^.Data[9] := 0;                     (* System processor *)
    buff^.Data[10] := ctrl;                 (* Control byte as specified *)
    IF ctrl = 2
    THEN « store op^.Device in buffer starting at byte 11 »
    END;
    IF op^.SoftwareID = ""
    THEN « store byte of 0 in buffer »      (* No software ID string supplied *)
    ELSE « store op^.SoftwareID string in buffer »
    END;
    IF op^.ScriptID ≠ ""
    THEN « store op^.ScriptID string in buffer »
    END;
    IF op^.CircPtr^.DataLink IN LANTypes
    THEN
        LOOP
            IF op^.Addresses = { }
            THEN EXIT
            END;                            (* Done if no more addresses *)
            target := « select and remove an address from op^.Addresses »;
            buff^.Destination := target;
            DLI.Send (op, buff, V4);        (*Send V4 format Boot message *)
            IF op^.ScriptID = ""
            THEN                            (* Skip V3 if asking for a V4-only function *)
```

```
            DLI.Send (op, buff, V3)         (*Send V3 format Boot message *)
         END
      END
   ELSE
      DLI.Send (op, buff, V4);              (*Send Boot message on point to point link *)
   END
END DLI.SendBoot;
```

4.4.4 Test directive

The Test directive is processed as follows:

- Get defaulted parameters from the Client database, if a Client name was given.
- Select a suitable Source SAP address (see Section 4.18.5 below).
- Perform the TEST exchange Count times. If the response has the wrong data, or the exchange times out, abort the Test operation.

  Special case: if the first exchange times out, try the next target address. If all addresses time out for all addresses, the Test fails with a Timeout error.

Since the purpose of a Test operation is to look for network problems, all the request/response exchanges are done as single attempts, *not* with the usual “Transact” multiple retries. This ensures that intermittent errors are not masked.

```
PROCEDURE DLI.Test (clientsn, circuitsn: Names.SimpleName; target: LANAddress;
      loopcount, looplength: INTEGER; loopdata: INTEGER; dsap: SAPAddress)
   RAISES {InvalidResponse, InsufficientResources, UnrecognizedCircuit, DataLinkError,
      UnrecognizedClient, RequiredArgumentOmitted, DirectiveNotSupported,
      InvalidArgumentValue, Timeout);
VAR op: POINTER TO Operation;
   buff: POINTER TO Buffer;
BEGIN
   DLI.Alloc (op, buff);                    (* Allocate necessary resources, if available *)
   TRY
      op^.Version := Unknown;
      op^.Op := TestRequester;
      op^.Count := 1; op^.Length := 40;     (* Default values *)
      op^.Data := 55H; op^.SAP := 0;        (* Default values *)
      op^.ClientName := clientsn;           (* if supplied as directive argument *)
      op^.CircuitName := circuitsn;         (* if supplied as directive argument *)
      op^.Addresses := target;              (* if supplied as directive argument *)
      IF target[0] = 1                      (* Multicast bit set in supplied target address? *)
      THEN
         RAISE (InvalidArgumentValue)
      END;
      op^.Count := loopcount;               (* if supplied as directive argument *)
      op^.Length := looplength;             (* if supplied as directive argument *)
      op^.Data := loopdata;                 (* if supplied as directive argument *)
      op^.SAP := dsap;                      (* if supplied as directive argument *)
      IF op^.SAP[0] = 1
      THEN                                  (* Group SAP is invalid *)
         RAISE (InvalidArgumentValue)
      END;
      IF « circuitsn not supplied »
      THEN
         IF op^.ClientPtr ≠ NIL
```

```
            THEN op^.CircuitName := op^.ClientPtr^.CircPtr^.Name
            END
        END;
        IF op^.CircuitName = ""
        THEN
            RAISE (RequiredArgumentOmitted)
        END;
        op^.CircPtr := FindCircuitByName (op^.CircuitName)
        IF « target not supplied »
        THEN
            IF op^.ClientPtr ≠ NIL
            THEN op^.Addresses := op^.ClientPtr^.Addresses;
            END;
            IF op^.Addresses = { }          (* Check for no addresses in Client database *)
            THEN RAISE (RequiredArgumentOmitted)
            END
        END;
        IF op^.CircPtr^.DataLink IN LANTypes
        THEN
            LAN.TestRequester (op, buff)
        ELSE
            RAISE (DirectiveNotSupported)
        END
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.Test;
```

4.4.5 Query directive

The Test directive is processed as follows:

- Get defaulted parameters from the Client database, if a Client name was given.
- Select a suitable Source SAP address (see Section 4.18.5 below).
- Perform the XID exchange. If it times out, try the next target address until all have been tried. If all addresses time out, the Query fails with a Timeout error.

```
PROCEDURE DLI.Query (clientsn, circuitsn: Names.SimpleName; target:  LANAddress;
        dsap: SAPAddress; VAR LLCTypes; SET OF INTEGER; VAR Window: INTEGER)
    RAISES {InvalidResponse, InsufficientResources, UnrecognizedCircuit,  DataLinkError,
        UnrecognizedClient, RequiredArgumentOmitted, DirectiveNotSupported,
        InvalidArgumentValue, Timeout);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
BEGIN
    DLI.Alloc (op, buff);                   (* Allocate necessary resources, if available *)
    TRY
        op^.Version := Unknown;
        op^.Op := QueryRequester;
        op^.SAP := 0;                       (* Default value *)
        op^.ClientName := clientsn;         (* if supplied as directive argument *)
        op^.CircuitName := circuitsn;       (* if supplied as directive argument *)
        op^.Addresses := target;            (* if supplied as directive argument *)
        IF target[0] = 1                    (* Multicast bit set in supplied target address? *)
        THEN
            RAISE (InvalidArgumentValue)
```

```
        END;
        op^.SAP := dsap;                    (* if supplied as directive argument *)
        IF op^.SAP[0] = 1
        THEN                                (* Group SAP is invalid *)
            RAISE (InvalidArgumentValue)
        END;
        IF « circuitsn not supplied »
        THEN
            IF op^.ClientPtr ≠ NIL
            THEN op^.CircuitName := op^.ClientPtr^.CircPtr^.Name
            END
        END;
        IF op^.CircuitName = ""
        THEN
            RAISE (RequiredArgumentOmitted)
        END;
        op^.CircPtr := FindCircuitByName (op^.CircuitName)
        IF « target not supplied »
        THEN
            IF op^.ClientPtr ≠ NIL
            THEN op^.Addresses := op^.ClientPtr^.Addresses;
            END;
            IF op^.Addresses = { }          (* Check for no addresses in Client database *)
            THEN RAISE (RequiredArgumentOmitted)
            END
        END;
        IF op^.CircPtr^.DataLink IN LANTypes
        THEN
            LAN.QueryRequester (op, buff, LLCTypes, Window)
        ELSE
            RAISE (DirectiveNotSupported)
        END
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.Query;
```

4.5 Downline load algorithms

This section defines the major algorithms used for down line load.

4.5.1 Downline load client

This section shows the Load Client algorithm.

In the case of a LAN load, the server address is normally not known. In that case, the first step is a Request Program message to the Load Assistance multicast address. These initial requests are sent both in V4 and V3 format, unless the request is one that is defined only in MOP V4. The first server that responds is used as the server for the rest of the load. However, if the load was initiated by a Boot message with Control = 1 (“Load server = Requesting system”), or if for some other reason the client knows which server should be used, then that server is used for the load and the multicast to the Load Assistance address is skipped.

The algorithm shows a timeout of one minute on each transaction once the load is underway. The load client does not have to do retransmissions at that stage, since the load server does them, but it does need some sort of timeout to handle the case where the network is partitioned or the load server has crashed. The load server does 15 retries at (by default) 4 second intervals, so a one minute load client timeout is appropriate.

There are some special rules relating to the Secondary Loader. This fits in a single Memory Data message. A request for a Secondary Loader is answered with that data, not with an Assistance Volunteer message. To ensure that subsequent load steps (for example Tertiary Loader or Operating System) do not terminate prematurely due to receiving another copy of a secondary loader, the load client ignores any Memory Data with Transfer Address messages that contain load data whenever it is loading anything other than Secondary Loader. Similarly, the load client ignores any Memory Data with Transfer address messages that contain no data when it is loading a Secondary Loader. The load sequences for other program types terminate either with a Parameter Load message or with a Memory Load with Transfer Address message that contains no data.

The amount of memory data in each Memory Load message may vary; the client must not assume that it will receive a full size message each time. The memory data may be null, for example if the load server does not have the next segment of memory data available yet but wants to keep the client from timing out. In all cases, the client transfers the memory data in the message (if any) and responds with a request for the next higher sequence number.

```
PROCEDURE DLI.LoadClient (circ: POINTER TO Circuit; server: LANAddress;
        prog: ProgType; swid, scrid: SoftwareID)
    RAISES {InsufficientResources, DataLinkError, Timeout}
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    msgtyp, seq, rseq, ldaddr: INTEGER;
BEGIN
    DLI.Alloc (op, buff);                (* Allocate necessary resources, if available *)
    op^.CircPtr := circ;
    op^.Op := LoadClient;
    op^.Version := Unknown;
    op^.Address := server;
    op^.ProgramType := prog;
    IF prog = CMIPScript
    THEN op^.Version := V4               (* CMIP Script is a V4 only function *)
    END;
    TRY
        IF op^.CircPtr^.DataLink IN LANTypes AND server = NullAddress
        THEN                             (* LAN and server not selected yet *)
            DLI.BuildReqProg (op, buff, prog, swid, scrid);
            buff^.Destination := LoadAssistance;
            DLI.BackoffTransact (op, buff, buff);
            op^.Address := buff^.Source;     (* Whoever answers is our server *)
            IF prog = SecondaryLoader        (* For Secondary the answer is the data *)
            THEN
                IF buff^.Data[0] ≠ MemLoadTA        (* If it isn't the right message type *)
                    OR buff^.Length ≤ 10            (* ...or message has null data *)
                THEN op^.Address := NullAddress;    (* No server yet after all *)
                    « quietly ignore this message and keep requesting »
                END
            ELSE                             (* for others, ask selected server again *)
                DLI.BuildReqProg (op, buff, prog, swid, scrid);
                buff^.Destination := op^.Address;
                DLI.Transact (op, buff, buff);
            END
        ELSE                             (* We know the server we want, talk to it *)
            DLI.BuildReqProg (op, buff, prog, swid, scrid);
            buff^.Destination := op^.Address;
```

```
            DLI.BackoffTransact (op, buff, buff);
        END
        seq := 0;                           (* Start with sequence number zero *)
        (* Note: on entry to the loop we have just received the first Memory Load  message *)
        LOOP
            msgtyp = buff^.Data[0];
            IF msgtyp = MemLoadTA AND buff^.Length > 10 AND prog ≠ SecondaryLoader
            THEN « quietly ignore this message »
            END;
            rseq := buff^.Data[1];          (* Get sequence number from received message *)
            IF rseq = seq                   (* If it matches the sequence number we want *)
            THEN
                ldaddr := DLI.GetLong (buff, 2);        (* Get memory address to load *)
                IF msgtyp = ParameterLoad
                THEN « store the received parameters »
                ELSEIF (msgtyp = MemLoad AND buff^.Length > 6)
                    OR (msgtyp = MemLoadTA AND buff^.Length > 10)
                THEN
                    IF prog IN { Management, CMIPScript } AND ldaddr = 0
                    THEN « begin a new management file or CMIP script record »
                    END;
                    « store data at specified ldaddr »
                END;
                seq := (seq + 1) MOD 256;   (* Look for the next consecutive sequence number *)
                IF msgtyp IN { MemLoadTA, ParameterLoad }
                THEN
                    EXIT                    (* Leave the loop, done with the load *)
                END
            END;
            DLI.BuildReqLoad (buff, seq);
            DLI.ReqResp (op, buff, buff, OneMinute, op^.Version);
        END;
        IF msgtyp = ParameterLoad           (* If system or diagnostic image load *)
        THEN
            DLI.BuildReqLoad (buff, seq);
            DLI.Send (op, buff, op^.Version); (*Send one more request to ACK the last message *)
        END;
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.LoadClient;
```

The procedures below show how the Load protocol messages are constructed. Note that the Format Version byte (third byte) of the Request Program message should be set to reflect what protocol version is being sent; the V3 requests have the value 1 in Format Version, and the V4 requests have the value 4.

```
PROCEDURE DLI.BuildReqProg (op: POINTER TO Operation;
    buff: POINTER TO Buffer; prog: ProgType; swid, scrid: SoftwareID);
VAR idlen: INTEGER;
    idstring: SoftwareID;
BEGIN
    buff^.Data[0] := RequestProgram;
    buff^.Data[1] := « Device Type code of data link being used »;
    buff^.Data[2] := « Format Version, see above »;
    IF prog = Diagnostic
    THEN buff^.Data[3] := System
    ELSE buff^.Data[3] := prog              (* Program Type *)
```

```
    END;
    IF prog = System THEN idstring := swid
    ELSEIF prog = CMIPScript THEN idstring := scrid
    ELSE idstring := ""
    END;
    idlen := LENGTH (idstring);
    IF idlen > 0
    THEN
        buff^.Data[4] := idlen;
        « store idstring starting at buff^.Data[5] »
    ELSEIF prog = System
    THEN buff^.Data[4] := –1              (* Standard operating system *)
    ELSEIF prog = Diagnostic
    THEN buff^.Data[4] := –2              (* Diagnostic system *)
    ELSE buff^.Data[4] := 0               (* No ID string *)
    END;
    buff^.Data[5 + idlen] := 0            (* System processor *)
    (* Store TLV entry for Data Link Buffer Size *)
    DLI.PutInteger (buff, 6 + idlen, 401);
    DLI.PutInteger (buff, 8 + idlen, 2);
    DLI.PutInteger (buff, 10 + idlen, op^.CircPtr^.SDUSize);
    buff^.Length := 12 + idlen
END DLI.BuildReqProg;

PROCEDURE DLI.BuildReqLoad (buff: POINTER TO Buffer; seq: INTEGER);
BEGIN
    buff^.Data[0] := RequestLoad;
    buff^.Data[1] := seq;                 (* Next sequence number we want *)
    buff^.Data[2] := 0;                   (* obsolete/reserved *)
    buff^.Length := 3
END DLI.BuildReqLoad;
```

4.5.2 Downline load server

The following sections show the load server algorithms.

4.5.2.1 Downline load server listener

This section shows the operation of the load server listener. It listens for load requests initiated by load clients, and uses the Request Program message and information in the Client database to decide whether and how to process the request. This process is active only if Load service is enabled for the circuit. The outline of the algorithm is as follows:

- A client database lookup is performed first.
  - First look for a client entry by circuit and (if a LAN) data link address.
  - If no match is found and this is a LAN, look for a client entry by circuit and device type.
- If Known Clients Only is False, and a syntactically valid Software ID string was included in the request, then the load can proceed whether or not an entry is found in the client lookup. Otherwise, if no entry is found, this is an Unrecognized Client.

  If an entry is found, the file names it specifies are used as the load files for this load operation unless the load client supplied a Software ID string.

- If the request specified a Software ID string, map the ID string to a file name and see if that file exists. If not, this is a File Open Error if a client entry was found previously, and an Unrecognized Client otherwise. (Note that the Software ID string overrides client database load file information.)

- If this is a LAN and the original request was addressed to the Load Assistance multicast address, send an Assistance Volunteer message and wait for a directed Request Program message (indicating this node was selected). If none is received, quietly go back to listening for other requests. (This condition is *not* an error and no event should be logged in this case.)

  Note: as shown here the Assistance Volunteer is sent before any files are opened, except where Software ID strings are involved. An implementation may instead open files first, and not volunteer until it has sucessfully done the open. This will avoid volunteering when the client database points to non-existent or inaccessible files.

  The secondary loader is a special case: the entire secondary loader is sent in a single message. No Assistance Volunteer is sent in this case; the response to the Request Program message is simply a message containing the secondary loader.

- Load the client according to the received request, with load file as determined above.

```
PROCEDURE DLI.LoadServer (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    device: DevType;
    prog: ProgType;
    idstring: SoftwareID;
    loadok: Boolean;                  (* True if we can load this client *)
    i: INTEGER;
EXCEPTION Quit;
BEGIN
    NEW (op);                         (* Get an Operation record *)
    op^.CircPtr := circ;
    op^.Op := LoadServer;
    LOOP
        TRY
            op^.Address := NullAddress;       (* We'll listen to any remote station address *)
            op^.Version := Unknown;           (* Accept any version *)
            DLI.Receive (op, buff, Infinite); (* Wait for a message, however long it takes *)
            IF buff^.Data[1] = 31             (* Special case *)
            THEN
                device := 37                  (* 37 is LQA, 31 is alternate code for LQA *)
            ELSE
                device := buff^.Data[1]       (* Get requester's device type *)
            END
            IF buff^.Length < 3
            THEN prog := SecondaryLoader      (* Default if no program type in message *)
            ELSE prog := buff^.Data[3];       (* Get program type *)
                IF prog = System
                THEN
                    IF « Request Program message indicates request for Diagnostic Image »
                    THEN prog := Diagnostic
                    END
                END
            END;
            op^.ProgramType := prog;
            idstring := « ID string from Request Program message, or null if no string supplied »;
```

```
IF NOT DLI.ValidID (idstring)
THEN idstring := ""                 (* If supplied ID isn't valid, pretend none was supplied *)
END;
IF « Data Link Buffer Size present in message »
THEN op^.SDUSize := « Data Link Buffer Size from message »
ELSE op^.SDUSize := 262
END;
loadok := FALSE;
op^.Client := DLI.FindClientByCircAddr (circ, buff^.Source);
IF circ^.DataLink IN LANTypes AND op^.Client = NIL
THEN
    op^.Client := DLI.FindClientByCircDev (circ, device)
END;
IF op^.Client ≠ NIL
THEN
    loadok := TRUE;
    op^.SecondaryLoader := op^.Client^.SecondaryLoader;
    op^.TertiaryLoader := op^.Client^.TertiaryLoader;
    op^.SystemImage := op^.Client^.SystemImage;
    op^.DiagnosticImage := op^.Client^.DiagnosticImage;
    op^.ManagementImage := op^.Client^.ManagementImage;
    op^.ScriptFile := op^.Client^.ScriptFile;
    IF idstring ≠ ""
    THEN                            (* ID string supplied, override database filename *)
        loadok := DLI.MapIDString (op, idstring)
    END
ELSEIF NOT KnownClientsOnly  (* No client entry found *)
    IF idstring ≠ ""
    THEN                            (* ID string supplied, map to filename *)
        loadok := DLI.MapIDString (op, idstring)
    END
END;
IF loadok
THEN
    IF circ^.DataLink IN LANTypes
        AND buff^.Destination[0] = 1      (* multicast request requires volunteer response *)
        AND op^.ProgramType ≠ SecondaryLoader          (* except secondary loaders *)
    THEN
        FOR i := 1 TO RetransmitMax
        DO
            LAN.SendVolunteer (op, buff);          (* Volunteer to serve *)
            TRY
                DLI.Receive (op, buff, OneMinute);
            EXCEPT
            | Timeout: RAISE (Quit)
            END;
            « received Request Program message should be the same as the multicast  request
              that initiated the lookup.  If not, quit and report a Load Request Failed  event,
              Reason = Protocol Error. »
            IF buff^.Destination[0] = 0
            THEN EXIT                   (* Request Program direct to us, carry on *)
            ELSEIF i = RetransmitMax
            THEN RAISE (Quit)           (* Volunteer isn't being heard, quit *)
            END
        END
    END;
    DLI.PerformLoad (op,buff);   (* Go perform the actual load with these parameters *)
    « report a Load Request Completed event »
```

```
            ELSE
                IF op^.Client = NIL
                THEN                    (* Couldn't load because no matching client entry *)
                    « report an Unrecognized Load Client event »
                ELSE                    (* Couldn't load because of bad Software ID *)
                    « report a Load Request Failed event, reason = File Open Error »
                END
        EXCEPT
        | Quit:                         (* Quit load attempt quietly *)
        | ELSE « report a Load Request Failed event, Reason as indicated by the  exception »
        END;
        DLI.ReceiveDone (buff)
    END
END DLI.LoadServer;

PROCEDURE DLI.MapIDString (op: POINTER TO Operation; idstring: SoftwareID) :  Boolean
VAR filename: FileSpec;
    exists: Boolean;
BEGIN
    filename := « map idstring to file name »;
    CASE op^.ProgramType OF
        SecondaryLoader:            op^.SecondaryLoader := filename |
        TertiaryLoader:             op^.TertiaryLoader := filename |
        System:                     op^.SystemImage := filename |
        Diagnostic:                 op^.DiagnosticImage := filename |
        Management:                 op^.ManagementImage := filename |
        ScriptFile:                 op^.ScriptFile := filename
    END;
    exists := « True if file named by filename exists »;
    RETURN (exists)
END DLI.MapIDString;

PROCEDURE LAN.SendVolunteer (op: POINTER TO Operation, buff: POINTER TO  Buffer);
BEGIN
    buff^.Destination := op^.Address;
    buff^.Length := 1;
    buff^.Data[0] := AssistanceVolunteer;
    DLI.Send (op, buff, op^.Version)
END LAN.SendVolunteer;
```

4.5.2.2 Downline load data transfer phase

This section shows the algorithm for the actual data transfer phase of the downline load process.

This procedure is given as input an Operation record, which specifies the address and MOP version of the client, as well as a load file name or possibly sequence of file names. This information was determined either from management directive parameters (in the case of a Load directive) or from a Request Program message (in the case of a load initiated by a load client); in either case the Client database may have been used to supply additional information.

The general flow of the load is

- Attempt to open load file; try next in sequence if failure, until end of list or success. If failure, abort the load.

- If program type is Secondary Loader, transfer the entire loader in a Memory Data with Transfer Address message.

Otherwise, transfer the data in multiple messages:

- Send each piece of load data in a Memory Data message. Increment the load address by the amount sent.
- If program type is Management Image or Script File, the data is organized in records. Each new record begins in a new Memory Data message, and for each new record the load address is reset to zero.
- After all data was sent, a load of System Image or Diagnostic Image is terminated with a Parameter Load message. Other program types are terminated with a Memory Data with Transfer Address message that has no load data in it. For the Management Image or Script file, the transfer address is –1.

- Once the load is complete, an implementation should wait for another Request Program message from the same client. In this way the server will handle a multi-step load without having to do multiple client database lookups. In the case of a load initiated by the Load directive, waiting for another request ensures that all the steps of a multi-step load are done in the context of that directive, i.e., with load parameters as supplied with the directive.

  The recommended timeout when waiting for another Request Program is 60 seconds.

```
PROCEDURE DLI.PerformLoad (op: POINTER TO Operation; buff; POINTER TO Buffer)
    RAISES { DataLinkError, Timeout, FileOpenError }
VAR prog: ProgType;
    files: ARRAY OF FileSpec;
    ldaddr, segsiz, seq, rseq, maxdatasize: INTEGER;
BEGIN
    CASE op^.ProgramType OF
        SecondaryLoader:            files := op^.SecondaryLoader |
        TertiaryLoader:             files := op^.TertiaryLoader |
        System:                     files := op^.SystemImage |
        Diagnostic:                 files := op^.DiagnosticImage |
        Management:                 files := op^.ManagementImage |
        ScriptFile:                 files := op^.ScriptFile
    END;
    LOOP
        IF files = { }
        THEN RAISE (FileOpenError)  (* Ran out of files to try *)
        END;
        « select and remove a filespec from files »;
        TRY
            « open the selected filespec »;
            EXIT                    (* Got the file we wanted *)
        EXCEPT
        ELSE                        (* Keep going on any error *)
        END
    END;
    seq := 0;
    IF op^.Version = V3
    THEN                            (* Check and correct erroneous buffer size values *)
        IF op^.SDUSize > 1498
        THEN op^.SDUSize := 1498
        ELSEIF op^.SDUSize = 1030
        THEN op^.SDUSize := 1010
        END
    END;
    IF op^.ProgramType = SecondaryLoader
```

```
THEN
    maxdatasize := MIN (op^.SDUSize, op^.CircPtr^.SDUSize) – 10;
                                    (* Max image data in a Memory Load with TA message *)
ELSE
    maxdatasize := MIN (op^.SDUSize, op^.CircPtr^.SDUSize) – 6;
                                    (* Max image data in a Memory Load message *)
END;
    ldaddr := « base load address of image »;
IF op^.ProgramType = SecondaryLoader
THEN
    « read entire secondary loader image, must fit in maxdatasize »;
    DLI.BuildMemLoadTA (op, buff, 0, ldaddr, « secondary loader data »)
ELSE
    LOOP
        « read another segment of load data, ≤ maxdatasize bytes long »;
        IF « no more data »
        THEN EXIT
        END;
        segsiz := « length of load data segment »;
        IF op^.ProgramType IN { Management, CMIPScript }
            AND « segment is the start of a new record »
        THEN ldaddr := 0
        END;
        DLI.BuildMemLoad (op, buff, seq, ldaddr, « load data segment »);
        LOOP
            DLI.Transact (op, buff, buff);
            IF buff^.Data[0] = RequestProgram    (* Another RequestProgram from client? *)
            THEN « quietly ignore this message »
            ELSE
                rseq := buff^.Data[1];      (* Get sequence number requested by client *)
                IF rseq = ((seq + 1) MOD 256)
                THEN EXIT                   (* asking next segment, go on *)
                ELSEIF rseq ≠ seq
                THEN « quietly ignore this message »
                END
            END
        END;
        seq := (seq + 1) MOD 256;
        ldaddr := ldaddr + segsiz;
    END;
    IF op^.ProgramType IN { System, Diagnostic }
    THEN DLI.BuildParameterLoad (op, buff, seq)
    ELSE DLI.BuildMemLoadTA (op, buff, seq, ldaddr, "")
    END
END;
(* Send the closing message of the load *)
LOOP
    TRY
        DLI.Transact (op, buff, buff)
    EXCEPT
    | Timeout: EXIT                     (* Timeout means we're done but not an error here *)
    END;
    IF buff^.Data[0] = RequestProgram   (* RequestProgram (for next load phase)? *)
    THEN EXIT                           (* done — and go process that request *)
    END;
    rseq := buff^.Data[1];              (* Get sequence number requested by client *)
    IF rseq = ((seq + 1) MOD 256)
    THEN EXIT                           (* asking next segment (ACK), done *)
```

```
        ELSEIF rseq ≠ seq
        THEN « quietly ignore this message »
        END
    END;
    IF buff^.Data[0] ≠ RequestProgram          (* If we didn't already receive the next Req Prog *)
        AND (op^.ProgramType IN { SecondaryLoader, TertiaryLoader }
        OR (op^.ProgramType IN { System, Diagnostic }
            AND (op^.ManagementImage OR op^.ScriptFile ≠ { })))
    THEN                                       (* Another request from this client is expected *)
        « wait for another Request Program from this client »
    END;
    IF buff^.Data[0] = RequestProgram
    THEN
        « repeat to process new received Request Program »
    END
END DLI.PerformLoad;

PROCEDURE DLI.BuildMemLoad (op: POINTER TO Operation, buff: POINTER TO  Buffer;
    seq, ldaddr: INTEGER; data: STRING);
BEGIN
    buff^.Destination := op^.Address;
    buff^.Data[0] := MemLoad;
    buff^.Data[1] := seq;
    DLI.PutLong (buff, 2, ldaddr);
    « move data into buffer starting at byte 6 »;
    buff^.Length := 6 + LENGTH(data)
END DLI.BuildMemLoad;

PROCEDURE DLI.BuildMemLoadTA (op: POINTER TO Operation, buff: POINTER TO  Buffer;
    seq, ldaddr: INTEGER; data: STRING);
BEGIN
    buff^.Destination := op^.Address;
    buff^.Data[0] := MemLoadTA;
    buff^.Data[1] := seq;
    DLI.PutLong (buff, 2, ldaddr);
    « move data into buffer starting at byte 6 »;
    IF op^.ProgramType IN { Management, CMIPScript }
    THEN DLI.PutLong (buff, 6 + LENGTH(data), –1)
    ELSE DLI.PutLong (buff, 6 + LENGTH(data), « transfer address from load  file »)
    END;
    buff^.Length := 10 + LENGTH(data)
END DLI.BuildMemLoadTA;

PROCEDURE DLI.BuildParameterLoad (op: POINTER TO Operation, buff: POINTER TO  Buffer;
    seq: INTEGER);
BEGIN
    buff^.Destination := op^.Address;
    buff^.Data[0] := ParameterLoad;
    buff^.Data[1] := seq;
    IF op^.ClientPtr ≠ NIL
    THEN
        IF op^.ClientPtr^.PhaseIVHostName ≠ ""
        THEN « move Phase IV host name into buffer »
        END;
        IF op^.ClientPtr^.PhaseIVHostAddress ≠ 0.0
        THEN « move Phase IV host address into buffer »
        END;
        IF op^.ClientPtr^.PhaseIVClientName ≠ ""
```

```
        THEN « move Phase IV client name into buffer »
        END;
        IF op^.ClientPtr^.PhaseIVClientAddress ≠ 0.0
        THEN « move Phase IV client address into buffer »
        END;
        (* optionally: *)
            IF op^.ClientPtr^.PhaseIVHostName ≠ ""
                OR op^.ClientPtr^.PhaseIVHostAddress ≠ ""
                OR op^.ClientPtr^.PhaseIVClientName ≠ ""
                OR op^.ClientPtr^.PhaseIVClientAddress ≠ ""
            THEN « move V3 format host system time into buffer »
            END
        (* End option *)
    END;
    (* optionally: *)
        IF op^.Version = V4
        THEN « move V4 format host system time into buffer »
        END;
    (* End option *)
    buff^.Data[2 + «length of parameters»] := 0;        (* end marker *)
    DLI.PutLong (buff, 3 + «length of parameters», « transfer address  from load file »)
    END;
    buff^.Length := 7 + «length of parameters»
END DLI.BuildParameterLoad;
```

4.6 Upline dump algorithms

This section defines the major algorithms used for upline dump.

4.6.1 Upline dump client

This section shows the Dump Client algorithm.

In the case of a LAN upline dump, the server address is often already known; for example, it may be the load server used to load the client, or the “host” identified in the Parameter Load message from the load. If no dump server address is known, the first step is a Request Dump Service message to the Load Assistance multicast address. The first server that responds is used as the server for the rest of the dump. However, if the address of the dump server has already been chosen, then that server is used for the dump and the multicast to the Load Assistance address is skipped. For example, an implementation may elect always to dump to its “Supporting Host”.

Unlike downline load, it is usually not desirable to wait indefinitely for dump service. Instead, the Request Dump Service is tried for some period of time; if no response is received, the dump is aborted and the system proceeds (typically with a restart). The algorithm below shows a DLI.Transact for the Request Dump Service message (15 tries, 60 seconds total); other similar approaches are acceptable. If the dump client does want to wait indefinitely for a dump server, it should use the DLI.BackoffTransact procedure to transmit the Request Dump Service message.

The algorithm shows a timeout of one minute on each transaction once the dump is underway. The dump client does not have to do retransmissions at that stage, since the dump server does them, but it does need some sort of timeout to handle the case where the network is partitioned or the dump server has crashed. The dump server does 15 retries at 4 second intervals, so a one minute dump client timeout is recommended.

The dump algorithm resembles the load algorithm in its general form, but there are some significant differences:

- There are no sequence numbers; each Request Dump Data message specifies a start address, and the client responds with data from that address.

- There is no acknowledgement to the Dump Complete message.

```
PROCEDURE DLI.DumpClient (circ: POINTER TO Circuit; server: LANAddress)
    RAISES {InsufficientResources, UnrecognizedCircuit, DataLinkError, Timeout}
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    msgtyp, memadr: INTEGER;
BEGIN
    DLI.Alloc (op, buff);                   (* Allocate necessary resources, if available *)
    op^.CircPtr := circ;
    op^.Op := DumpClient;
    op^.Version := Unknown;
    op^.Address := server;
    TRY
        IF op^.CircPtr^.DataLink IN LANTypes AND server = NullAddress
        THEN                                (* LAN and server not selected yet *)
            DLI.BuildReqDumpServ (op, buff);
            buff^.Destination := LoadAssistance;
            DLI.Transact (op, buff, buff);
            op^.Address := buff^.Source;    (* Whoever answers is our server *)
        END;
        DLI.BuildReqDumpServ (op, buff);    (* Ask the selected server for dump service *)
        buff^.Destination := op^.Address;
        DLI.Transact (op, buff, buff);
        (* Note: on entry to the loop we have just received the first Request Memory  Dump message *)
        LOOP
            msgtyp = buff^.Data[0];
            IF msgtyp = DumpComplete
            THEN EXIT                       (* Dump complete message — all finished *)
            END;
            memadr := DLI.GetLong (buff, 1);  (* address to be dumped *)
            buff^.Data[0] := DumpData;      (* Respond with Dump Data message *)
            DLI.PutLong (buff, 1, memadr);  (* Indicate base address of data *)
            « store data (if available) starting at byte 5 »;
            buf^.Length := 5 + « length of memory data »;
            DLI.ReqResp (op, buff, buff, OneMinute, op^.Version);
        END
    FINALLY
        DLI.Free (op, buff)
    END
END DLI.DumpClient;
```

The procedures below show how the Dump protocol messages are constructed. Note that the Format Version byte (third byte) of the Request Dump Service message should be set to reflect what protocol version is being sent; the V3 requests have the value 1 in Format Version, and the V4 requests have the value 4.

```
PROCEDURE DLI.BuildReqDumpServ (op: POINTER TO Operation; buff: POINTER TO  Buffer);
BEGIN
    buff^.Data[0] := RequestDumpService;
    buff^.Data[1] := « Device Type code of data link being used »;
    buff^.Data[2] := « Format Version, see above »;
```

```
    DLI.PutLong (buff, 3, « size of memory to be dumped »);
    buff^.Data[7] := 2;
    (* Store TLV entry for Data Link Buffer Size *)
    DLI.PutInteger (buff, 8, 401);
    DLI.PutInteger (buff, 10, 2);
    DLI.PutInteger (buff, 12, op^.CircPtr^.SDUSize);
    buff^.Length := 14
END DLI.BuildReqDumpServ;
```

4.6.2 Upline dump server

The following sections show the dump server algorithms.

4.6.2.1 Upline dump server listener

This section shows the operation of the dump server listener. It listens for dump service requests initiated by dump clients, and uses the Request Dump Service message and information in the Client database to decide whether and how to process the request. This process is active only if Dump service is enabled for the circuit. The outline of the algorithm is as follows:

- First look for a client entry by circuit and (if a LAN) data link address.
- If no match is found and this is a LAN, look for a client entry by circuit and device type.

  If no entry is found, this is an Unrecognized Client. If an entry is found, the file names it specifies are used as the dump files for this dump operation.
- If this is a LAN and the original request was addressed to the Load Assistance multicast address, send an Assistance Volunteer message and wait for a directed Request Dump Service message (indicating this node was selected). If none is received, quietly go back to listening for other requests. (This condition is *not* an error and no event should be logged in this case.)

  Note: as shown here the Assistance Volunteer is sent before any files are opened. An implementation may instead open files first, and not volunteer until it has sucessfully done the open. This will avoid volunteering when the client database points to inaccessible files.
- Dump the client according to the received request, with dump file and other parameters as determined above.

After dump completion, the dump server simply goes back to waiting for more requests. The dump client will presumably attempt to restart, but this does not involve the dump server. In particular, the dump server should *not* send a Boot message or otherwise interfere with the restart of the client.

```
PROCEDURE DLI.DumpServer (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    client: POINTER TO Client;
    buff: POINTER TO Buffer;
    device: DevType;
    count: INTEGER;
    i: INTEGER;
EXCEPTION Quit;
BEGIN
    NEW (op);                          (* Get an Operation record *)
    op^.CircPtr := circ;
    op^.Op := DumpServer;
```

```
LOOP
    TRY
        op^.Address := NullAddress;         (* We'll listen to any remote station address *)
        op^.Version := Unknown;             (* Accept any version *)
        DLI.Receive (op, buff, Infinite);   (* Wait for a message, however long it takes *)
        IF buff^.Data[1] = 31               (* Special case *)
        THEN
            device := 37                    (* 37 is LQA, 31 is alternate code for LQA *)
        ELSE
            device := buff^.Data[1]         (* Get requester's device type *)
        END
        count := DLI.GetLong (buff, 3);     (* Memory Size field *)
        IF « Data Link Buffer Size present in message »
        THEN op^.SDUSize := « Data Link Buffer Size from message »
        ELSE op^.SDUSize := 262
        END;
        op^.Client := DLI.FindClientByCircAddr (circ, buff^.Source);
        IF circ^.DataLink IN LANTypes AND client = NIL
        THEN
            op^.Client := DLI.FindClientByCircDev (circ, device)
        END;
        IF op^.Client ≠ NIL
        THEN
            op^.DumpFile := op^.Client^.DumpFile;
            op^.DumpAddress := op^.Client^.DumpAddress;
            op^.DumpCount := count – op^.DumpAddress;
                                            (* make count = memory size – start address *)
            IF circ^.DataLink IN LANTypes
                AND buff^.Destination[0] = 1        (* multicast request requires volunteer response *)
            THEN
                FOR i := 1 TO RetransmitMax
                DO
                    LAN.SendVolunteer (op, buff);           (* Volunteer to serve *)
                    TRY
                        DLI.Receive (op, buff, OneMinute);
                    EXCEPT
                    | Timeout: RAISE (Quit)
                    END;
                    « received Request Dump Service message should be the same as the multicast
                      request that initiated the lookup.  If not, quit and report a Dump Request  Failed
                      event, Reason = Protocol Error. »
                    IF buff^.Destination[0] = 0
                    THEN EXIT                       (* Request Program direct to us, carry on *)
                    ELSEIF i = RetransmitMax
                    THEN RAISE (Quit)               (* Volunteer isn't being heard, quit *)
                    END
                END
            END;
            DLI.PerformDump (op,buff);  (* Go perform the actual dump with these parameters *)
            « report a Dump Request Completed event »
        ELSE
            « report an Unrecognized Dump Client event »
    EXCEPT
    | Quit:                                 (* Quit dump attempt quietly *)
    | ELSE « report a Dump Request Failed event, Reason as indicated by the  exception »
    END;
    DLI.ReceiveDone (buff)
END                                         (* loop until told to stop *)
```

END DLI.DumpServer;

4.6.2.2 Upline dump data transfer phase

This section shows the algorithm for the actual data transfer phase of the upline dump process.

This procedure is given as input an Operation record, which specifies the address and MOP version of the client, as well as a dump file name or possibly sequence of file names and other dump parameters. This information was determined from a Request Dump Service message received from a dump client and from information in the Client database.

The general flow of the dump is

- Attempt to open dump file; try next in sequence if failure, until end of list or success. If failure, abort the dump.
- Send Request Dump Data messages for each piece of the dump. As data is received, increment the dump address and decrement the count. Repeat until the dump count has been satisfied.
- End the dump by sending the Dump Complete message once.

The amount of data sent by the Dump Client in each Dump Data message can vary. The client may send as much data as will fit in the data link buffer size (as reported in the Request Dump Service) message. The client may also send less data. There is no requirement to send the same amount of data all the time. In particular, if a client needs to "delay" the dump server (for example because it has not finished gathering up all the data yet) it may respond to a request with a Dump Data message containing no image data. In that case the server will simply repeat the request. There is no limit to the number of times the client may use this option.

```
PROCEDURE DLI.PerformDump (op: POINTER TO Operation; buff; POINTER TO Buffer)
    RAISES { DataLinkError, Timeout, FileOpenError }
VAR memadr, count, segsiz, recadr, maxdatasize: INTEGER;
BEGIN
    LOOP
        IF op^.DumpFile = { }
        THEN RAISE (FileOpenError)          (* Ran out of files to try *)
        END;
        « select and remove a filespec from op^.DumpFile »;
        TRY
            « open the selected filespec »;
            EXIT                            (* Got the file we wanted *)
        EXCEPT
        ELSE                                (* keep going on any error *)
        END
    END;
    memadr := op^.DumpAddress;
    count := op^.DumpCount;
    IF op^.Version = V3
    THEN                                    (* Check and correct erroneous buffer size values *)
        IF op^.SDUSize > 1498
        THEN op^.SDUSize := 1498
        ELSEIF op^.SDUSize = 1030
        THEN op^.SDUSize := 1010
        END
    END;
```

```
    maxdatasize := MIN (op^.SDUSize, op^.CircPtr^.SDUSize) – 5;
                                      (* Max image data in a Memory Dump Data message *)
    LOOP
        segsiz := MIN (maxdatasize, count);    (* use what's left to dump if that's less than what fits *)
        DLI.BuildReqDump (op, buff, memadr, segsiz);
        LOOP
            DLI.Transact (op, buff, buff);
            recadr := DLI.GetLong (buff, 1);   (* Get start of data returned by client *)
            IF recadr = memadr
            THEN EXIT                          (* that's what we asked for, handle it *)
            ELSE
                « quietly ignore this message »
            END
        END;
        segsiz := buff^.Length – 5;            (* Length of received dump data *)
        « write received dump data, if any, to dump file »;
        memadr := memadr + segsiz;
        count := count – segsiz;
        IF count ≤ 0
        THEN EXIT                              (* Nothing left to dump *)
        END
    END;
    (* Send the closing message of the dump *)
    buff^.Destination := op^.Address;
    buff^.Length := 1;
    buff^.Data[0] := DumpComplete;
    DLI.Send (op, buff, op^.Version)
END DLI.PerformDump;

PROCEDURE DLI.BuildReqDump (op: POINTER TO Operation, buff: POINTER TO  Buffer;
    memadr, segsiz: INTEGER);
BEGIN
    buff^.Destination := op^.Address;
    buff^.Data[0] := ReqDumpData;
    DLI.PutLong (buff, 1, memadr);
    DLI.PutInteger (buff, 5, segsiz);
    buff^.Length := 7
END DLI.BuildReqDump;
```

4.7 Loop algorithms for point to point data links

4.7.1 Loop requester

Unlike most other MOP functions, the loop requester is intentionally *not* tolerant of problems on the link. Instead of retrying a number of times, it performs each request-response exchange once. If it succeeds, the loop continues; if it fails, the loop is finished (and in error).

```
PROCEDURE PTP.LoopRequester (op: POINTER TO Operation; buff: POINTER TO  Buffer)
    RAISES {InvalidResponse, DataLinkError, Timeout};
VAR i, j: INTEGER;
BEGIN
    FOR i := 1 TO op^.Count
    DO
        buff^.Data[0] := LoopData;             (* Function code *)
```

```
        DLI.PutInteger (buff, 1, i);        (* Use sequence counter for receipt number.  Note: any other
                                            receipt number that is different for each consecutive message
                                            may also be used. *)
        FOR j := 0 to op^.Length – 1
        DO
            buff^.Data[j + 3] := op^.Data   (* Load the test data pattern into the buffer *)
        END;
        buff^.Length := op^.Length + 3;     (* Buffer length = loop data length + 3 byte header *)
        DLI.ReqResp (op, buff, buff, op^.CircPtr^.RetransmitTimer, V4);
        IF buff^.Length ≠ op^.Length + 3    (* If wrong length response *)
        THEN
            RAISE (InvalidResponse, op^.Count – i)
        END;
        j := DLI.GetInteger (buff, 1);      (* Receipt number of response *)
        IF NOT (buff^.Data[0] IN { LoopData, LoopedData } )
            OR j ≠ i
        THEN                                (* If wrong function code or receipt number *)
            RAISE (InvalidResponse, op^.Count – i)
        END;
        FOR j := 0 TO op^.Length – 1
        DO
            IF buff^.Data[j + 3] ≠ op^.Data (* Check test data pattern *)
            THEN
                RAISE (InvalidResponse, op^.Count – i)
            END
        END
    END
END PTP.LoopRequester;
```

4.7.2 Loop server

There is always one Loop Server active for each point to point circuit. It keeps a receive pending at all times. When a message comes in (which will have function code LoopData), the message is modified appropriately and sent back to the sender.

Note that the Loop Server is required to support the full range of valid message sizes for each data link.

```
PROCEDURE PTP.LoopServer (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
BEGIN
    NEW (op);                           (* Get an Operation record *)
    op^.CircPtr := circ;
    op^.Op := LoopServer;
    LOOP
        DLI.Receive (op, buff, Infinite);   (* Wait for a message, however long it takes *)
        buff^.Data[0] := LoopedData;        (* Change the message code *)
        DLI.Send (op, buff, V4);            (* Send it back (version is not applicable, actually) *)
        DLI.ReceiveDone (buff)              (* Done using the receive buffer *)
    END                                     (* Repeat that forever *)
END PTP.LoopServer;
```

4.8 Loop algorithms for LAN data links

4.8.1 Loop requester

The procedures below define the algorithm for the Loop Requester.

The LAN loopback protocol uses two message codes, “Forward Data” and “Reply”. Forward Data specifies that the message is to be sent onwards to another station. Reply indicates that the message terminates at this station and should be delivered to a Loop Requester. The message sent by the requester in effect consists of a Reply, enveloped in a number of Forward Data envelopes. This provides the mechanism needed to do “loop assistance”.

If assisted loopback is desired, the caller will often supply the data link address of the station to use as assistant. If no assistant address is supplied, then the Loop Requester attempts to find an assistant. Stations willing to be assistant listen to the Loopback Assistance multicast address. To find an assistant, the requester sends a loop message to that multicast address, and uses the source address from the first received response as the assistant address for the remainder of the operation.

Note that all Loop messages (except for the one used to find a loop assistant) are sent as single attempts. Since the purpose of the Loop directive is to look for network problems, no retries are done during the operation. Otherwise soft errors would be obscured.

```
PROCEDURE LAN.LoopRequester (op: POINTER TO Operation; buff: POINTER TO Buffer)
    RAISES {InvalidResponse, DataLinkError, Timeout};
VAR i, j, offset, buflen: INTEGER;
    addresses: SET OF LANAddress;
    target: LANAddress;
BEGIN
    op^.Version := V4;                  (* Initially use V4 format *)
    addresses := op^.Addresses;
    FOR i := 1 TO op^.Count
    DO
        IF i = 1
        THEN                            (* For first one we have to select the address *)
            LAN.FindAssistant (op, buff);       (* Select assistant if we need one *)
            LOOP
                IF addresses = { }
                THEN                    (* No target addresses left to try *)
                    IF op^.Version = V4
                    THEN                (* V4 didn't work, try it with V3 *)
                        op^.Version := V3;
                        addresses := op^.Addresses;
                        LAN.FindAssistant (op, buff);(* Find a V3 assistant if we need one *)
                    ELSE RAISE (Timeout) (* Both versions received no response *)
                    END
                ELSE
                    target := « select and remove an address from addresses »;
                    LAN.BuildLoopMsg (buff, target, op^.AssistanceType, op^.AssistantAddress, op, i,
                                        buflen);
                    TRY
                        DLI.RequestResponse (op, buff, buff, op^.Version);
                        op^.Address := target;      (*If we get an answer, that's the address to use *)
                        EXIT            (* Leave the address search loop *)
                    EXCEPT
                    | Timeout:          (* If no answer, keep on trying addresses *)
                    END
```

```
                END
            END                                 (* end of target address search loop *)
        ELSE                                    (* Not first message *)
            LAN.BuildLoopMsg (buff, target, op^.AssistanceType, op^.AssistantAddress, op,  i, buflen);
            DLI.ReqResp (op, buff, buff, op^.Version)
        END;
        IF buff^.Length ≠ buflen                (* If wrong length response *)
        THEN
            RAISE (InvalidResponse, op^.Count – i)
        END;
        offset := DLI.GetInteger (buff, 0);     (* Get skip count *)
        j := DLI.GetInteger (buff, offset + 2); (* Receipt number of response *)
        IF j ≠ i                                        (* If wrong receipt number *)
            OR offset + 4 + op^.Length ≠ buflen         (* or wrong offset *)
        THEN
            RAISE (InvalidResponse, op^.Count – i)
        END;
        FOR j := 0 to op^.Length – 1
        DO
            IF buff^.Data[j + offset + 4] ≠ op^.Data   (* Check test data *)
            THEN
                RAISE (InvalidResponse, op^.Count – i)
            END
        END
    END
END LAN.LoopRequester;
```

The following procedure builds a LAN Loopback message. The message consists of a Reply message (containing the data), preceded by one or more Forward headers. These headers list the addresses on the path the message is to take. For example, if Transmit assistance is being used, the Loopback message destination address is the assistant address; the first Forward header contains the target station address, and the second forward header contains the station address for our circuit.

```
PROCEDURE LAN.BuildLoopMsg (buff: POINTER TO Buffer; target: LANAddress; help:  HelpType;
    assistant: LANAddress; op: POINTER TO Operation; receipt: INTEGER; VAR buflen:  INTEGER);
VAR offset: INTEGER;
BEGIN
    DLI.PutInteger (buff, 0, 0);            (* Initial skip count = 0 *)
    IF help IN {Transmit, Full}             (* If first hop is to assistant *)
    THEN
        buff^.Destination := assistant;     (* Set first hop destination address *)
        LAN.Forward (buff, 2, target);      (* First Forward header points to target *)
        offset := 10
    ELSE                                    (* Receive assistance or no assistance, first hop to target *)
        buff^.Destination := target;
        offset := 2
    END;
    IF help IN {Receive, Full}              (* If assistant used on the way back *)
    THEN
        LAN.Forward (buff, offset, assistant);
        offset := offset + 8
    END;
    LAN.Forward (buff, offset, op^.CircPtr^.DataLinkAddress);
                                            (* Last forward header points back to us *)
    offset := offset + 8;
    DLI.PutInteger (buff, offset, Reply);   (* Put in Reply header *)
    DLI.PutInteger (buff, offset + 2, receipt);
```

```
    FOR j := 0 to op^.Length – 1
    DO
        buff^.Data[j + offset + 4] := op^.Data   (* Load the test data pattern into the buffer *)
    END;
    buflen := op^.Length + offset + 4;           (* total loop message length *)
    buff^.Length := buflen
END LAN.BuildLoopMsg;

PROCEDURE LAN.Forward (buff: POINTER TO Buffer; offset: INTEGER; address;  LANAddress);
BEGIN
    DLI.PutInteger (buff, offset, ForwardData);
    DLI.PutAddress (buff, offset + 2, address)
END LAN.Forward;

PROCEDURE LAN.FindAssistant (op: POINTER TO Operation; buff: POINTER TO  Buffer);
VAR assistants: SET OF LANAddress;
BEGIN
    IF op^.AssistanceType ≠ None                 (* Need to select an assistant? *)
    THEN
        assistants := op^.AssistantAddresses;    (* Initialize the list of addresses to try *)
        LOOP
            IF assistants = { }
            THEN                                 (* No assistant addresses left to try *)
                IF op^.Version = V4
                THEN                             (* V4 didn't work, try it with V3
                    op^.Version := V3;
                    assistants := op^.AssistantAddresses
                ELSE RAISE (Timeout)             (* Both versions received no response *)
                END
            ELSE
                target := « select and remove an address from assistants »;
                LAN.BuildLoopMsg (buff, target, None, NullAddress, op, –1);
                TRY
                    DLI.Transact (op, buff, buff);        (* Do a "direct" loop to the selected address *)
                    op^.AssistantAddress := buff^.Source;        (* Whoever responds is the assistant *)
                    EXIT                         (* Leave the assistant search loop *)
                EXCEPT
                | Timeout:                       (* If no answer, keep on trying addresses *)
                END
            END
        END
    END
END LAN.FindAssistant;
```

4.8.2 Loop server

There is always one Loop Server active for each LAN circuit. It keeps a receive pending at all times. When a message comes in (which will have function code ForwardData), the message is modified appropriately and sent onwards. Invalid messages are quietly ignored.

Note that the Loop Server is required to support the full range of valid message sizes for each data link.

```
PROCEDURE LAN.LoopServer (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    offset: INTEGER;
```

```
     nexthop: LANAddress;
BEGIN
     NEW (op);                              (* Get an Operation record *)
     op^.CircPtr := circ;
     op^.Op := LoopServer;
     LOOP
          op^.Address := NullAddress;        (* We'll listen to any remote station address *)
          op^.Version := Unknown;            (* Accept any version *)
          DLI.Receive (op, buff, Infinite);  (* Wait for a message, however long it takes *)
          offset := DLI.GetInteger (buff, 0); (* Get the offset value from message *)
          IF buff^.Length ≥ offset+10        (* Must have our header plus next guy's msgtype *)
          THEN
               nexthop := GetAddress (buff, offset+2);   (* Get address to send to *)
               IF nexthop[0] = 0              (* Must not be a multicast address *)
               THEN
                    buff^.Destination := nexthop;  (* Forwarding address is our destination *)
                    DLI.PutInteger (buff, 0, offset+8);   (* Update Offset in message buffer  *)
                    DLI.Send (op, buff, op^.Version)
               END
          END;
          DLI.ReceiveDone (buff)             (* Done using the receive buffer *)
     END                                     (* Repeat that forever *)
END LAN.LoopServer;
```

4.9 XID and TEST algorithms

This section describes the operation of the Test and Query Requesters. These are functionally quite separate from the remainder of MOP, for example because they do not use the normal LLC data transfer services of the data link. Note that the corresponding server (responder) algorithms are not shown here; they are defined in the IEEE 802.2 (LLC) standard.

4.9.1 Test requester

```
PROCEDURE LAN.TestRequester (op: POINTER TO Operation)
     RAISES {InvalidResponse, DataLinkError, Timeout};
VAR i, j: INTEGER;
     buff: POINTER TO Buffer;
     target: LANAddress;
     ssap: SAPAddress;
BEGIN
     ssap := LAN.SelectaSAP (op);            (* Pick a source SAP to use *)
     LAN.EnableSAP (op^.CircPtr^.LLCport, ssap);      (* Enable that SAP for the duration *)
     buff := op^.buff;
     TRY
          FOR i := 1 TO op^.Count
          DO
               buff^.Data[0] := op^.SAP;       (* Destination SAP address *)
               buff^.Data[1] := ssap;          (* Source SAP address = system specific, Command *)
               buff^.Data[2] := LLCTEST;       (* Frame Control = TEST, P/F = 0 *)
               DLI.PutInteger (buff, 3, i);    (* Use sequence counter for receipt number.  Note: any other
                                               receipt number that is different for each consecutive message
                                               may also be used. *)
               FOR j := 0 to op^.Length – 1
               DO
                    buff^.Data[j + 5] := op^.Data  (* Load the test data pattern into the buffer *)
               END;
```

```
            buff^.Length := op^.Length + 5;    (* Buffer length = loop data length + 5 byte header *)
            IF i = 1                           (* Have to select an address *)
            THEN
                LOOP
                    IF op^.Addresses = { }
                    THEN RAISE (Timeout)           (* No response from anything *)
                    END;
                    target := « select and remove an address from op^.Addresses »;
                    buff^.Destination := target;
                    TRY
                        LAN.LLCReqResp (op, buff, LLCTEST);
                                               (* Do a request/response, looking for TEST response *)
                        op^.Address := target;     (* If it works, that's the target address *)
                        EXIT
                    EXCEPT
                    | Timeout:                 (* If no answer, keep on trying addresses *)
                    END
                END                            (* End of target address search loop *)
            ELSE                               (* Not first message *)
                LAN.LLCReqResp (op, buff, LLCTEST);
                                               (* Do a request/response, looking for TEST response *)
            END;
            IF buff^.Length ≠ op^.Length       (* If wrong length response *)
                OR buff^.Data[1] ≠ op^.SAP + 01H   (* Must be a response frame from the right SAP *)
            THEN
                RAISE (InvalidResponse, op^.Count – i)
            END;
            j := DLI.GetInteger (buff, 3);     (* Receipt number of response *)
            IF j ≠ i                           (* If wrong receipt number *)
            THEN
                RAISE (InvalidResponse, op^.Count – i)
            END;
            FOR j := 0 to op^.Length – 1
            DO
                IF buff^.Data[j + 5] ≠ op^.Data   (* Check test data *)
                THEN
                    RAISE (InvalidResponse, op^.Count – i)
                END
            END
        END
    FINALLY
        LAN.DisableSAP (op^.CircPtr^.LLCport, ssap)         (* Done with the SAP) *)
    END
END LAN.TestRequester;
```

4.9.2 Query requester

```
PROCEDURE LAN.QueryRequester (op: POINTER TO Operation;
    VAR LLCTypes: BITSET; VAR Window: INTEGER)
    RAISES {InvalidResponse, DataLinkError, Timeout};
VAR buff: POINTER TO Buffer;
    target: LANAddress;
    ssap: SAPAddress;
BEGIN
    ssap := LAN.SelectaSAP (op);               (* Pick a source SAP to use *)
    LAN.EnableSAP (op^.CircPtr^.LLCport, ssap);    (* Enable that SAP for the duration *)
    buff := op^.buff;
```

```
    buff^.Length := 6;                  (* 3 bytes LLC header, 3 bytes info *)
    buff^.Data[0] := op^.SAP;           (* Destination SAP address *)
    buff^.Data[1] := ssap;              (* Source SAP address = system specific, Command. *)
    buff^.Data[2] := LLCXID;            (* Frame Control = XID, P/F = 0 *)
    buff^.Data[3] := 81H;               (* IEEE basic format XID *)
    buff^.Data[4] := 01H;               (* We say we do LLC class 1 only *)
    buff^.Data[5] := 00H;               (* No window size *)
    TRY
        LOOP
            IF op^.Addresses = { }
            THEN RAISE (Timeout)                (* No response from anything *)
            END;
            target := « select and remove an address from op^.Addresses »;
            buff^.Destination := target;
            TRY
                LAN.LLCTransact (op, buff, LLCXID);
                                        (* Do a request/response, looking for XID response *)
                op^.Address := target;  (* If it works, that's the target address *)
                EXIT
            EXCEPT
            | Timeout:                  (* If no answer, keep on trying addresses *)
            END
        END;
        IF buff^.Length ≠ 6             (* If wrong length response *)
        THEN
            RAISE (InvalidResponse)
        END;
        IF buff^.Data[1] ≠ op^.SAP + 01H    (* Must be a response frame from the right SAP *)
            OR buff^.Data[3] ≠ 81H      (* ... and format = IEEE Basic Format *)
        THEN
            RAISE (InvalidResponse)
        END;
        LLCTypes := buff^.Data[4] MOD 20H;      (* Get bitmask for supported LLC types *)
        Window := buff^.Data[5] DIV 2   (* ... and window size *)
    FINALLY
        LAN.DisableSAP (op^.CircPtr^.LLCport, ssap)         (* Done with the SAP) *)
    END
END LAN.QueryRequester;
```

4.10 Console server algorithms

There is always one Console Server active for each LAN circuit. It keeps a receive pending at all times. The messages it can receive are Request ID, Request Counters, and, if supported and enabled, Boot and Console Carrier messages. It checks the message type, and dispatches appropriate to build the response.

```
PROCEDURE LAN.ConsoleServer (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    function: Byte;
    timeout: Time;
BEGIN
    NEW (op);                           (* Get an Operation record *)
    op^.CircPtr := circ;
    op^.Op := ConsoleServer;
    LOOP
        op^.Address := NullAddress;     (* We accept requests from any source *)
```

```
        op^.Version := Unknown;                  (* Accept any version *)
        IF circ^.ConsoleUser = NullAddress
        THEN                                     (* Console not currently reserved *)
            DLI.Receive (op, buff, Infinite)     (* Wait for a message, however long it takes *)
        ELSE
            timeout := « time remaining before circ^.ReservationTimer expires »;
            TRY
                DLI.Receive (op, buff, timeout);       (* Wait for a message, up to reservation timeout *)
            EXCEPT
            | Timeout:
                circ^.ConsoleUser := NullAddress;      (* Cancel reservation *)
                DLI.Receive (op, buff, Infinite)       (* Wait for a message, however long it takes *)
            END
        END;
        function := buff^.Data[0];               (* Get the function code from the buffer *)
        IF function = RequestID                  (* Do the right thing given the function code *)
        THEN LAN.SysIDResponse (op, buff)
        ELSEIF function = RequestCounters
        THEN LAN.CountersResponse (op, buff)
        ELSEIF function IN { ReserveConsole, ReleaseConsole, ConsoleCommand }
        THEN LAN.ConsoleCarrierResponse (op, buff)
        END
        ELSEIF function = Boot AND « Boot supported »
        THEN DLI.ReceiveBoot (op, buff);
        DLI.ReceiveDone (buff)                   (* Done using this receive buffer *)
    END
END LAN.ConsoleServer;
```

4.10.1 Periodic System ID transmission

For each LAN circuit, there is one Periodic System ID transmitter process. Strictly speaking, this is part of the Console Server, but it is simpler to show it as a separate process. It has a very simple task: every 5 minutes ± 20% it sends a System ID message. The first is sent in V4 format, the next in V3 format, and so on alternating the format.

```
PROCEDURE LAN.PeriodicSysID (circ: POINTER TO Circuit);
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    tmo: Time;
BEGIN
    NEW (op);                                (* Get an Operation record *)
    op^.CircPtr := circ;                     (* Initialize what we need *)
    NEW (buff);                              (* Allocate a buffer *)
    buff^.Destination := ConsoleMulticast;   (* Send periodic messages to this address *)
    LOOP
        op^.Version := V4;                   (* First we send a V4 format System ID *)
        LAN.SendSysID (op, buff, 0);         (* Send a System ID with receipt number of zero *)
        tmo := FiveMinutes * (0.8 + 0.4 * Random ( ) );  (* Compute the delay time *)
        System.Wait (tmo);                   (* ... and wait for that long *)
        op^.Version := V3;                   (* Next we send a V3 format System ID *)
        LAN.SendSysID (op, buff, 0);         (* Send a System ID with receipt number of zero *)
        tmo := FiveMinutes * (0.8 + 0.4 * Random ( ) );  (* Compute the delay time *)
        System.Wait (tmo)                    (* ... and wait for that long *)
    END                                      (* Repeat all this indefinitely *)
END LAN.PeriodicSysID;
```

4.10.2 Build a System ID message

This procedure builds a System ID message and sends it. Depending on the version number of the intended recipient (given by the Version field of the Operation record that is passed), certain fields are included or omitted; refer to the description of the System ID message in Sections 5.5.3 and G.6. The receipt number to be used is passed as an argument. For periodic System ID transmissions, the value is zero; for responses to requests, the value is what was sent in the Request ID message.

```
PROCEDURE LAN.SendSysID (op: POINTER TO Operation; buff: POINTER TO Buffer;
    receipt: INTEGER);
BEGIN
    buff^.Data[0] := SysID;              (* Function code *)
    buff^.Data[1] := 0;                  (* Reserved byte *)
    DLI.PutInteger (buff, 2, receipt);   (* Put in the 16-bit receipt number *)
    « now put in the other fields. »;
    DLI.Send (op, buff, op^.Version)
END LAN.SendSysID;
```

4.10.3 Response to Request ID

```
PROCEDURE LAN.SysIDResponse (op: POINTER TO Operation; buff: POINTER TO Buffer);
VAR receipt: INTEGER;
BEGIN
    IF buff^.Length ≥ 4                  (* Make sure request is long enough to be valid *)
    THEN
        receipt := DLI.GetInteger (buff, 2);    (* Get receipt number *)
        buff^.Destination := buff^.Source;      (* Response goes to sender of request *)
        LAN.SendSysID (op, buff, receipt)       (* Now construct the response and send it *)
    END
END LAN.SysIDResponse;
```

4.10.4 Response to Request Counters

The specifics of the counters response depend somewhat on which datalink is being used, but the whole algorithm is shown here anyway for simplicity. In particular, the CSMA/CD counters are sent with a different response message than for other datalinks, and only the CSMA/CD counters response can be sent to a V3 MOP implementation.

```
PROCEDURE LAN.CountersResponse (op: POINTER TO Operation; buff: POINTER TO Buffer);
VAR counterblock: ARRAY OF Counter;
    receipt: INTEGER;
BEGIN
    IF buff^.Length ≥ 3                  (* Make sure request is long enough to be valid *)
    THEN
        receipt := DLI.GetInteger (buff, 1);    (* Fetch receipt number from request *)
        buff^.Destination := buff^.Source;      (* Response goes to sender of request *)
        IF op^.CircPtr^.DataLink = CSMACD
        THEN
            buff^.Data[0] := CSMACDCounters;   (* Response message code *)
            DLI.PutInteger (buff, 1, receipt);   (* Store receipt number in response *)
            counterblock := CSMACD.Get (op^.CircPtr.LinkName, "COUNTERS");
            IF op^.Version = V4
            THEN
                « move counters into buffer starting at byte 3 »;
                buff^.Length := 203             (* Length of V4 format CSMA/CD counters *)
```

```
            ELSE
                « convert counters to V3 form »;
                « move converted counters into buffer starting at byte 3 »;
                buff^.Length := 57              (* Length of V3 format CSMA/CD counters *)
            END;
            DLI.Send (op, buff, op^.Version)
        ELSEIF op^.CircPtr^.DataLink = FDDI          (* Only other alternative currently is FDDI *)
        THEN
            IF op^.Version = V4
            THEN
                buff^.Data[0] := OtherCounters;          (* Response message code *)
                DLI.PutInteger (buff, 1, 5);     (* Store data link code (5 = FDDI) *)
                DLI.PutInteger (buff, 3, receipt);   (* Store receipt number *)
                counterblock := FDDI.Get (op^.CircPtr.LinkName, "COUNTERS");
                « move counters into buffer starting at byte 4 »;
                buff^.Length := 260;             (* Length of counter message for FDDI counters *)
                DLI.Send (op, buff, op^.Version)
            END                                  (* If V3 requester, ignore the request *)
        END
    END
END LAN.CountersResponse;
```

4.10.5 Console carrier server

This section shows the algorithms for the Console Carrier portion of the Console Server.

For Console Command messages, there is a one-bit sequence number to deal with lost messages. If a retransmitted Console Command message is received, the previously sent response is resent. Note that this is the same message as before, even if additional console response data is available at this time. New console response data is considered only when a *new* Console Command message is received.

The circuit reservation timer ensures that the console is released if no console carrier message is received from the console carrier requester for the timeout period. The timeout should be long enough to allow for message loss and recovery by retransmission, but not so long that the console remains unavailable for an extensive period of time. The recommended value for the reservation timeout is 60 seconds. (This corresponds to the 15 retries done by the requester at the default 4 second retransmit interval, so the client and server would give up at about the same time if communication is lost.)

The purpose of the console carrier is to provide the common ASCII local console function via a simple low level network protocol. Character stream data is sent by the console carrier requester and any console output is returned in a character stream by the console carrier server. The server should echo input (characters received from the requester) in the same manner as it would echo local console data.

```
PROCEDURE LAN.ConsoleCarrierResponse (op: POINTER TO Operation; buff: POINTER  TO Buffer);
VAR function, cflags, seq: INTEGER;
BEGIN
    IF NOT ( ConsoleCarrier IN op^.CircPtr^.Functions)
    THEN RETURN                          (* Ignore message if not enabled or not supported *)
    END;
    function := buff^.Data[0];           (* Get the function code from the buffer *)
    IF function = ReserveConsole
    THEN
        IF op^.CircPtr^.ConsoleUser = NullAddress
            AND « verification field in message is valid »
        THEN
```

```
            op^.CircPtr^.ConsoleUser := buff^.Source;
            StartTimer (op^.CircPtr^.ReservationTimer, « reservation timeout » );
            op^.CircPtr^.ConsoleSeq := 1        (* Next expected sequence number *)
        END
    ELSEIF function = ReleaseConsole
    THEN
        IF buff^.Source = op^.CircPtr^.ConsoleUser
        THEN
            op^.CircPtr^.ConsoleUser := NullAddress;
            StopTimer (op^.CircPtr^.ReservationTimer)
        END
    ELSE                                        (* It's a Console Command message *)
        IF buff^.Source = op^.CircPtr^.ConsoleUser
        THEN
            cflags := buff^.Data[1];            (* Get command flags *)
            seq := cflags MOD 2;                (* Sequence number *)
            IF seq = op^.CircPtr^.ConsoleSeq            (* New message *)
            THEN
                StartTimer (op^.CircPtr^.ReservationTimer, « reservation timeout » );
                IF (cflag DIV 2) MOD 2 = 1   (* Check Command Break flag *)
                THEN « issue Break to console »
                END;
                « issue command data (if any) to console »;
                op^.CircPtr^.ConsoleSeq := seq;
                cflag := seq;                   (* Sequence number goes into response message *)
                IF « any command data was lost »
                THEN cflag := cflag + 2
                END;
                IF « any response data was lost since last new Console Command message »
                THEN cflag := cflag + 4
                END;
                buff^.Data[0] := ConsoleResponse;
                buff^.Data[1] := cflag;
                « load response data, if any, into buff^.Data starting at byte 2 »;
                buff^.Length := 2 + « length of response data »
            ELSE buff^.Data := « previously sent response »
            END;
            buff^.Destination := buff^.Source; (* Response goes to sender of request *)
            DLI.Send (op, buff, op^.Version)   (* Send the response message *)
        END
    END
END LAN.ConsoleCarrierResponse;
```

4.10.6 Received Boot message processing

The purpose of the Boot message is to terminate whatever the node is doing and force it to reboot. The Boot message can contain parameters which change the normal reboot actions.

In order to be most effective, recognition of the Boot message should not depend on the correct operation of system software. This means that recognition of the Boot message should be done in the data link hardware or in the data link adapter firmware, if possible. When this is not possible, recognition of the Boot message at a low level in the system software (e.g., in the device driver at interrupt level) may still provide a useful function.

Since the Boot message aborts normal operation of the node, it is essential that there is a way to disable recognition of the Boot message. For systems that are usually managed locally, disabling Boot recognition by default is generally appropriate. In addition, all implementations must check the Verification field of the Boot message and ignore any message that does not have the correct value in the Verification field. The expected Verification value must be settable. Note that settable verification alone is no substitute for being able to disable Boot message recognition entirely.

Once a Boot message has been validated, the system should be rebooted. If possible, the implementation should preserve the information in the Boot message and pass that to the reboot process. This gives the sending node control over how the target node will restart. In particular, support for the Control field value 1 ("Boot Server – Requesting System") is important for the Load directive to work reliably. However, implementations that preserve no state and simply execute the default boot actions are permitted.

If the Control field is 1 ("Boot Server – Requesting System") the system should execute the Load Requester to reload the Secondary and/or Tertiary Loader (if required), the Operating System, and the CMIP Script or Management File (if required). Each of these loads should go directly to the node sending the Boot message, bypassing the multicast to the Load Assistance address.

If the System Software ID field is present in the Boot message, its value should be used in the Software ID field of the Request Program message when requesting the Operating System load. Similarly, if the Script Software ID field is present in the Boot message, its value should be used in the Software ID field of the Request Program message when requesting the CMIP Script load. For other load steps, or when those Software ID fields are omitted from the Boot message, the system should supply its default value in the Software ID field of the Request Program message.

4.10.7 Configuration monitor

The Configuration Monitor is started (for a particular circuit) by the Enable Circuit directive and stopped by the Disable directive. It receives all System ID messages addressed to the Console Multicast address. There can be multiple Configuration Monitors, one for each LAN circuit; these operate independently.

Received System ID messages are identified by their data link source address. If a Station subentity exists for that address, the information in that subentity is updated. If not, then a new Station subentity is created (resources permitting) and loaded with information from the System ID message.

In either case, the attributes of the Station subentity (see Table 14) are loaded, with appropriate data type conversions as needed, from the corresponding fields of the System ID message. Note that the Node ID attribute is encoded in the System ID message in two parts, which are concatenated to form the attribute. Any fields for which there is no corresponding attribute are ignored. The Last Report attribute is updated to reflect the time when the System ID message was received.

When the Configuration Monitor is stopped for a circuit, all Station subentities of that circuit are deleted.

4.10.8 Console carrier requester

The console carrier requester algorithm is shown below.

The central part of the algorithm is a polling loop in which the target station is polled periodically for new console carrier data. New console input is also sent to the target in this loop as needed.

Note:

The polling should be done frequently enough to carry console data from the target to the requester promptly and avoid console response data loss, but not so often that the target station is overloaded by the polling activity. The minimum permitted poll interval is 80 milliseconds. Somewhat longer poll intervals are generally more appropriate (several hundred milliseconds). Adaptive algorithms may be used for increased efficiency; for example, poll at 100 ms intervals until no new data is seen for some time (e.g., 10 seconds); then poll at 1 s intervals; switch back to 100 ms intervals when new data is seen in either direction.

The timeout for each poll should be substantially longer than the poll rate (the default timeout of 4 seconds is appropriate) to ensure that the timeout is longer than the likely packet lifetime in the network. This is necessary because the protocol only has a one bit sequence number.

The purpose of the console carrier is to provide the common ASCII local console function via a simple low level network protocol. Character stream data is sent by the console carrier requester and any console output is returned in a character stream by the console carrier server. The requester should assume that the server will echo input (characters received from the requester) in the same manner as local console data; therefore the requester should not echo characters locally.

```
PROCEDURE LAN.CarrierRequester (circ: POINTER TO Circuit)
    RAISES {InvalidResponse, DataLinkError, Unavailable, Timeout};
VAR op: POINTER TO Operation;
    buff: POINTER TO Buffer;
    seq, cflags, rseq: INTEGER;
    target: LANAddress;
BEGIN
    DLI.Alloc (op, buff);               (* Allocate necessary resources, if available *)
    TRY
        op^.Op := ConsoleCarrier;
        op^.CircPtr := circ;
        op^.Address := NullAddress;
        op^.Version := Unknown;
        LOOP
            IF op^.Addresses = { }
            THEN RAISE (Timeout)            (* No response from anything *)
            END;
            target := « select and remove an address from op^.Addresses »;
            LAN.BuildReqID (buff, target);
            TRY
                DLI.Transact (op, buff, buff);
                op^.Address := target;  (* If it works, that's the target address *)
                EXIT
            EXCEPT
            | Timeout:                  (* If no answer, keep on trying addresses *)
            END
        END;
        IF « console carrier not supported »
        THEN RAISE (Unavailable)
        END;
        FOR i := 1 TO RetransmitMax
        DO
            LAN.BuildReserve (buff, op^.Address);   (* Try to reserve console *)
            DLI.Send (op, buff, op^.Version);
```

```
            System.Wait ( « a few milliseconds » );
                                             (* Wait to allow target to make the reservation and update the
                                             System ID it sends to reflect the reservation *)
            LAN.BuildReqID (buff, op^.Address);      (* Check the result *)
            DLI.Transact (op, buff, buff);
            IF « console reserved by circ^.DataLinkAddress »
            THEN EXIT
            ELSEIF « console reserved »
            THEN RAISE (Unavailable)         (* Someone else got it first *)
            END;
            IF i = RetransmitMax
            THEN RAISE (Timeout)
            END
        END;
        seq := 1;                            (* Initial sequence number *)
        LOOP
            IF « user is finished with console carrier »
            THEN EXIT                        (* Leave the polling loop *)
            System.Wait (« poll delay »);    (* See note above *)
            LAN.BuildConsolePoll (buff, op^.Address, seq);
            DLI.Transact (op, buff, buff);
            cflags := buff^.Data[1];         (* Control flags from console response message *)
            rseq := cflags MOD 2;            (* Received sequence number *)
            IF rseq = seq
            THEN                             (* Valid response to our poll *)
                seq := (seq + 1) MOD 2;      (* Update the transmit sequence number *)
                « send console response data to user »;
                                             (* This queues the data, it doesn't wait for it to be handled.  If
                                             there is no room for more, discard the data *)
                IF (cflags DIV 2) MOD 2 = 1
                THEN                         (* Data lost flag(s) set *)
                    « send data lost error indication(s) to user »
                END;
                « get new console command data from user »
            END
        END;
        FOR i := 1 TO RetransmitMax
        DO
            buff^.Length := 1;
            buff^.Destination := op^.Address;
            buff^.Data[0] := ReleaseConsole;
            DLI.Send (op, buff, op^.Version)  (* Release the console *)
            System.Wait ( « a few milliseconds » );
                                             (* Wait to allow target to release the reservation and update
                                             the System ID it sends to reflect the release *)
            LAN.BuildReqID (buff, op^.Address);      (* Check the result *)
            DLI.Transact (op, buff, buff);
            IF NOT « console reserved by circ^.DataLinkAddress »
            THEN EXIT                        (* Leave loop if not reserved, or reserved by another *)
            END
        END
    FINALLY
        DLI.Free (op, buff)
    END
END LAN.CarrierRequester;

LAN.BuildReqID (buff: POINTER TO Buffer; target: LANAddress);
    buff^.Length := 4;
```

```
    buff^.Destination := target;
    buff^.Data[0] := RequestID;
    buff^.Data[1] := 0;                         (* reserved field *)
    DLI.PutInteger (buff, 2, « a suitable non-zero receipt number »)
END LAN.BuildReqID;

LAN.BuildReserve (buff: POINTER TO Buffer; target: LANAddress);
    buff^.Length := 9;
    buff^.Destination := target;
    buff^.Data[0] := ReserveConsole;
    « load verification value into bytes 1–8 »
END LAN.BuildRereserve;

LAN.BuildConsolePoll (buff: POINTER TO Buffer; target: LANAddress, seq:  INTEGER);
    buff^.Destination := target;
    buff^.Data[0] := ConsoleCommand;
    IF « break requested by user »
    THEN buff^.Data[1] := seq+2
    ELSE buff^.Data[1] := seq
    END;
    « load user console command data, if any, into bytes 2 and up »;
    buff^.Length := 2 + « length of command data »
END LAN.BuildConsolePoll;
```

4.11 MOP protocol primitives

The algorithms presented in this section perform the basic protocol message exchanges: transmit, request/response with or without retry.

4.11.1 Send message

This procedure is the common interface to the data link dependent transmit function. It transmits a single message. It returns when the data link is finished with the buffer (which may or may not imply successful transmission of the message).

```
PROCEDURE DLI.Send (op: POINTER TO Operation; msg: POINTER TO Buffer, ver:  MOPVersion)
    RAISES {DLD.CommonExceptions};
BEGIN
    DLD.Transmit (op, msg, ver) ;               (* Issue the Transmit request *)
    LOOP
        TRY
            DLD.TransmitPoll (op, msg);         (* Check if transmit has finished *)
            RETURN                              (* It has...  *)
        EXCEPT
        | NotComplete:                          (* Keep going if not done yet *)
        END
    END
END DLI.Send;
```

4.11.2 Receive message with timeout

This procedure is the common interface for receiving a message. It attempts to obtain a message from the receive dispatcher. If a message is received within the specified timeout period, that message is returned. Otherwise, an exception is generated when the timeout period has expired.

This procedure is used by a thread which is processing a particular MOP operation. In the model used here, the receive dispatcher dispatches received packets, based on data link envelope, protocol ID, remote data link address (for LANs), and MOP message code, to the appropriate operation thread. As a result, this procedure returns only “appropriate” messages. Refer to the receive dispatcher algorithms (Section 4.12) for more detail.

Note that receipt of a message establishes the remote station address and MOP version number to be used for the current operation.

```
PROCEDURE DLI.Receive (op: POINTER TO Operation; VAR msg: POINTER TO Buffer;
    tmo: Time; ver: MOPVersion)
    RAISES {Timeout};
BEGIN
    StartTimer (op^.Timeout, tmo);          (* Start the receive timeout *)
    LOOP
        IF Expired (op^.Timeout)            (* Check for timer expiry *)
        THEN
            RAISE (Timeout)
        END;
        IF « a message was received from the receive dispatcher »
        THEN
            StopTimer (op^.Timeout);        (* Cancel the timer *)
            msg := « pass the buffer pointer »;
            op^.Address := buff^.Source;    (* We’re now talking to this address *)
            op^.Version := « version of received message »;
            RETURN                          (* Return with the message *)
        END
    END
END DLI.Receive;
```

When the processing of the receive buffer is complete, it is released by a call to the ReceiveDone routine.

```
PROCEDURE DLI.ReceiveDone (buff: POINTER TO Buffer)
BEGIN
    « return the buffer to the buffer pool »
END DLI.ReceiveDone;
```

4.11.3 Perform a single request-response exchange attempt

This procedure is used to perform a single attempt at a request-response exchange. If the response to the request is received within the timeout period, that response is returned; otherwise an exception is raised.

The basic request-response exchange needs as one of its inputs the MOP version number to use, which indicates which data link envelope (Ethernet or IEEE 802.2) will be used on LANs.

```
PROCEDURE DLI.ReqResp (op: POINTER TO Operation; smsg: POINTER TO Buffer;
    VAR rmsg: POINTER TO Buffer; tmo: Time, ver: MOPVersion)
    RAISES {Timeout};
BEGIN
    DLI.Send (op, smsg, ver);               (* Send the request *)
    DLI.Receive (op, rmsg, tmo, ver)        (* Wait for response or timeout *)
END DLI.ReqResp;
```

4.11.4 "Transact" request-response exchange with retry

This procedure performs a request-response exchange with a limited number of retries. The retry limit is given by the architectural constant RetransmitMax. The time between retries is given by the Circuit characteristic Retransmit Timer. If the request does not succeed within the limit of the number of retries, an exception is raised.

This procedure also selects the appropriate MOP protocol version to use if the version has not yet been determined. This is done by performing one attempt using version 4, then an attempt using version 3, and so on alternately until either a response arrives or the retry limit is exceeded. In this case, the retry limit is in effect doubled, i.e., an attempt using version 4 plus an attempt using version 3 counts for a single retry. Version 4 is normally tried first, but if an implementation has some reason to expect that version 3 is more likely to be correct, it would make sense to begin with that version. As an optimization, implementations may try both versions simultaneously. For some protocol exchanges (e.g., Loop) care must be taken not to confuse the replies; the receipt number in the message can be used to do this.

```
PROCEDURE DLI.Transact (op: POINTER TO Operation; smsg: POINTER TO Buffer;
    VAR rmsg: POINTER TO Buffer)
    RAISES {Timeout};
VAR tries: INTEGER;
BEGIN
    FOR tries := 1 TO RetransmitMax
    DO
        IF op^.Version = Unknown            (* If we don't know the protocol version yet *)
        THEN
            (* If we don't know the version, try V4, then V3; do these two steps repeatedly *)
            TRY
                DLI.ReqResp (op, smsg, rmsg, op^.CircPtr^.RetransmitTimer, V4);
                RETURN                      (* Return to caller *)
            EXCEPT
            | Timeout:                      (* Keep going if no answer received *)
            END;
            TRY
                DLI.ReqResp (op, smsg, rmsg, op^.CircPtr^.RetransmitTimer, V3);
                RETURN                      (* Return to caller *)
            EXCEPT
            | Timeout:                      (* Keep going if no answer received *)
            END
        ELSE                                (* Version is known, use it *)
            TRY
                DLI.ReqResp (op, smsg, rmsg, op^.CircPtr^.RetransmitTimer, op^.Version);
                RETURN                      (* If it worked, return to caller *)
            EXCEPT
            | Timeout:                      (* Keep going if no answer received *)
            END
        END
    END;
    RAISE (Timeout)
END DLI.Transact;
```

4.11.5 Request-response with infinite retry for Request Program

This procedure performs a request-response exchange with unlimited retries similar to the one shown above for DLI.MustTransact. The one shown here is intended for use with the Request Program message. Instead of retrying at a fixed rate, the retries are done in short bursts; within a burst, retries are spaced by the Circuit characteristic Retransmit Timer. If the protocol version number to be used is not known, then the burst consists of a number of V4 messages first, followed by V3 messages. This ensures that the Requester will not unintentionally fall back to V3 protocol if a message is lost or the server is slow in responding. As with the other request/response exchanges, an implementation may start with V3 and try V4 next if it has reason to expect V3 to be likely to be the right version. As an optimization, implementations may try both versions simultaneously.

The bursts in turn are spaced according to a “Backoff” timer, which increases as the number of retries increases. To help reduce the problem of many clients transmitting requests in lock-step, the actual delay between bursts has a “random jitter” applied to it, which varies the delay by ± 25% from the backoff value. The random number generator used to control sending of periodic SysID messages can be used for this purpose.

The procedure returns only on successful completion of the exchange.

Note:

Since the retransmissions within the burst are separated by Retransmit Timer (4 seconds by default) and the initial value of the backoff timer is equal to that, each node requesting load service can contribute up to 0.25 packets/second of load on the load server. Under certain conditions, such as restoration of power to a significant size LAN, the number of nodes requesting load service may be substantial. Therefore it is very important for load servers to process Request Program messages efficiently.

In large networks, there typically will be a number of load servers, each of which is responsible for loading a subset of all the load clients on the LAN. This will help performance when there are many requests, but *only* if the servers can dismiss requests from clients that they are not set up to load in a very short time. Therefore, implementations should be designed to be able to dismiss requests within a few milliseconds.

```
PROCEDURE DLI.BackoffTransact (op: POINTER TO Operation; smsg: POINTER TO Buffer;
    VAR rmsg: POINTER TO Buffer);
VAR backofftime, tries: INTEGER;
    tmo: Time;
BEGIN
    backofftime := op^.CircPtr^.RetransmitTimer;
                                        (* Initialize backoff *)
    LOOP
        IF op^.Version = Unknown        (* If we don’t know the protocol version yet *)
        THEN
            (* If we don’t know the version, try a burst of V4, then a burst of V3 *)
            FOR tries := 1 TO BurstSize     (* Do a burst *)
                TRY
                    DLI.ReqResp (op, smsg, rmsg, op^.CircPtr^.RetransmitTimer, V4);
                    RETURN                  (* Return to caller *)
                EXCEPT
                | Timeout:                  (* Keep going if no answer received *)
                END
            END;
            FOR tries := 1 TO BurstSize     (* Do a burst *)
                IF tries = BurstSize
                THEN
                    tmo := backofftime * (0.75 + 0.5 * Random ( ) )
                                        (* On last try of a burst, wait the backoff time *)
                ELSE
                    tmo := op^.CircPtr^.RetransmitTimer
                END;
                TRY
```

```
                DLI.ReqResp (op, smsg, rmsg, tmo, V3);
                RETURN                  (* Return to caller *)
            EXCEPT
            | Timeout:                  (* Keep going if no answer received *)
            END
        END
    ELSE
        FOR tries := 1 TO BurstSize     (* Do a burst *)
            IF tries = BurstSize
            THEN
                tmo := backofftime * (0.75 + 0.5 * Random ( ) )
                                        (* On last try of a burst, wait the backoff time *)
            ELSE
                tmo := op^.CircPtr^.RetransmitTimer
            END;
            TRY
                DLI.ReqResp (op, smsg, rmsg, tmo, op^.Version);
                RETURN                  (* If it worked, return to caller *)
            EXCEPT
            | Timeout:                  (* Keep going if no answer received *)
            END
        END
    END;
    backofftime := MIN (backofftime * 2, MaxBackoff)
  END
END DLI.BackoffTransact;
```

4.11.6 Enable and disable SAP address for XID/TEST requester

The following procedures enables and disable a SAP address on the port used by the Test or Query Requester. The CSMA/CD and FDDI data link architectures support the sending of XID and TEST frames only on ports that are opened for User Supplied LLC. A port of that type is opened when the circuit is enabled (see Section 4.16.1). Only the CSMA/CD data link requests are shown here; the ones for FDDI are identical.

```
PROCEDURE LAN.EnableSAP (port: PortID; ssap: SAPAddress);
BEGIN
  CSMACD.EnableSAP (port,ssap);         (* Enable the specified SAP address *)
END LAN.EnableSAP;

PROCEDURE LAN.DisableSAP (port: PortID; ssap: SAPAddress);
BEGIN
  CSMACD.DisableSAP (port,ssap);        (* Disable the specified SAP address *)
END LAN.DisableSAP;
```

4.11.7 Perform an XID or TEST exchange

The procedures below handle the data link service interface calls for the Test and Query Requester. For simplicity, only the CSMA/CD calls are shown; if the data link being used is FDDI, then the corresponding FDDI service interface calls would be used (which take the same set of arguments).

These procedures are essentially the same as the common MOP Transact and ReqResp procedures. The only functional difference is that they exchange LLC control messages rather than data messages. In addition, the receive processing deals with arriving XID and TEST command frames, since that is required by the IEEE 802.2 LLC standard. (This is normally a function of the data link layer, but when a port is open in User Supplied LLC mode, those frames are delivered to the user and need to be handled. The actual processing is not shown; refer to the IEEE 802.2 standard for details.)

```
PROCEDURE LAN.LLCTransact (op: POINTER TO Operation; buff: POINTER TO Buffer;  LLCFC: Byte)
    RAISES {Timeout};
VAR tries: INTEGER;
BEGIN
    FOR tries := 1 TO RetransmitMax
    DO
        TRY
            LAN.LLCReqResp (op, buff, LLCFC);
            RETURN                          (* If it worked, return to caller *)
        EXCEPT
        | Timeout:                          (* Keep going if no answer received *)
        END
    END;
    RAISE (Timeout)
END LAN.LLCTransact;

PROCEDURE LAN.LLCReqResp (op: POINTER TO Operation;
    VAR msg: POINTER TO Buffer; LLCFC: Byte);
    RAISES {Timeout};
VAR i: INTEGER;
    detail: CSMACD.ReceiveErrorDetail;
    src, dest: LANAddress;
    form: CSMACD.FormatAndMux;
    b: BOOLEAN;
    fcs: CSMACD.FCSValue;
BEGIN
    LAN.LLCSend (op, msg);                  (* Send the request *)
    StartTimer (op^.Timeout, op^RetransmitTimer);
    LOOP
        CSMACD.Receive (op^.CircPtr^.LLCport, FALSE, msg, i)
        LOOP
            IF Expired (op^.Timeout)
            THEN
                CSMACD.ReceiveAbort (op^.CircPtr^.LLCport)
                RAISE (Timeout)
            END;
            TRY
                CSMACD.ReceivePoll (op^.CircPtr^.LLCport, detail, dest, src, form, msg, i, b,  fcs);
                EXIT                        (* Receive completed, handle it *)
            EXCEPT
            | NotComplete:                  (* Keep going if nothing received yet *)
            END
        END;
        IF msg^.Data[1] MOD 2 = 0           (* If it's a Command frame *)
        THEN
            CASE msg^.Data[2] OF            (* Dispatch on the LLC FC field *)
            LLCTEST:                        (* TEST *)
                CSMACD.TestResponse (op^.CircPtr^.LLCport, msg)
            | LLCXID:                       (* XID *)
                CSMACD.XIDResponse (op^.CircPtr^.LLCport, msg)
            | ELSE
                (* Ignore any other frames *)
            END
        ELSEIF msg^.Data[2] = LLCFC
        THEN                                (* If a response, is this the one we're waiting for? *)
```

```
            StopTimer (op^.Timeout);        (* Cancel the receive timeout *)
            RETURN                          (* ... and we're done *)
        END
    END
END LAN.LLCReqResp;

PROCEDURE LAN.LLCSend (op: POINTER TO Operation; msg: POINTER TO Buffer);
VAR form: CSMACD.FormatAndMux;
BEGIN
    form.format := IEEE;
    CSMACD.Transmit (op^.CircPtr^.LLC, op^.Address, op^.CircPtr^.DataLinkAddress, form, msg);
    LOOP
        TRY
            CSMACD.TransmitPoll (op^.CircPtr^.LLC, msg);
            EXIT
        EXCEPT
        | NotComplete:                      (* Keep going if not done yet *)
        END
    END
END LAN.LLCSend;
```

4.12 Receive dispatchers

The receive dispatcher examines packets received from the data link layer and dispatches them to the appropriate processing thread. The rules by which the correct thread is chosen depend on the data link type. The sections below define the receive dispatcher algorithms in detail.

4.12.1 Receive dispatcher for point to point data links

In the case of point to point data links, the dispatching decision is based simply on the MOP message function code (the first byte of the MOP message). If there is a processing thread waiting for the message (i.e., it is in the DLI.Receive routine) then it is given the message. If not, then the message is ignored.

The dispatching rules are as follows:

1. If the message is a Looped Data message, it goes to the loop requester.

2. If the message is a Loop Data message, and the loop requester is active on the circuit, the message goes to the loop requester. If not, it goes to the loop server. (The reason for this peculiar rule is that Loop Data messages may be sent out by a loop requester and be looped back by a passive loopback connector. For that case they have to be treated as responses.)

3. If the message is a Memory Load, Memory Load with Transfer Address, or Parameter Load with Transfer Address message, it goes to the load requester.

4. If the message is a Request Program or Request Memory Load message, it goes to the load server.

5. If the message is a Boot message, it goes to the console server.

6. If the message is a Request Dump Data or Dump Complete message, it goes to the dump requester.

7. If the message is a Request Dump Service or Memory Dump Data message, it goes to the dump server.

8. Any other message, including a message with undefined message function codes, is quietly ignored.

4.12.2 Receive dispatcher for LAN data links

In the case of LAN data links, the dispatching decision can be based on a number of factors:

- MOP message function code (the first byte of the MOP message)
- Protocol Identifier (Protocol Type for Ethernet format messages), or DSAP address in the case of XID and TEST responses
- Source address
- MOP version (as implied by the data link envelope: Ethernet format packets are V3, 802.2 format packets are V4)

If there is a processing thread waiting for the message (i.e., it is in the DLI.Receive routine) then it is given the message. If not, then the message is ignored.

When the message pertains to an operation that is in progress (as opposed to one just being initiated), the search for a matching processing thread includes a comparison of the message source address with the operation's remote station address. In those cases the version number of the message is also checked against what was expected; if the two do not match the message is quietly ignored.

When the message pertains to an operation that is just being initiated, reception of a response establishes the remote station address and version for this operation. (See DLI.Receive, Section 4.11.2).

The dispatching rules are as follows:

1. If the Protocol Identifier is the Loop Protocol ID:

   a. If the message is a Reply message, it goes to the loop requester.

      Note: it is not correct to check the message source address against the address of the loop target in this case, since the last hop of the loop may have been an assistant. One solution is to check against either the target address or the assistant address depending on the assistance type; another approach is not to permit multiple concurrent loop requester operations on the same circuit.

   b. If the message is a Forward Data message, it goes to the loop server.

2. If the Protocol Identifier is the Load Protocol ID:

   a. If the message is an Assistance Volunteer message, it goes to the load requester or the dump requester.

      (Presumably the load requester and dump requester are never simultaneously active, otherwise it would be impossible to decide which of the two receives the Assistance Volunteer message.)

   b. If the message is a Memory Load, Memory Load with Transfer Address, or Parameter Load with Transfer Address message and the message source address matches, the message goes to the load requester.

Special case: if a load of the Secondary Loader is being requested from the Load Assistance multicast address, then the address check is omitted. (For the Secondary Loader, the server sends the data immediately rather than sending an Assistance Volunteer message first.)

c. If the message is a Request Program, it goes to the load server. If there is a load server thread waiting for a Request Program message from this remote station (i.e., because it is processing a Load directive addressed to that station) the message is delivered to that thread.

d. If the message is a Request Memory Load message and the message source address matches, the message goes to the corresponding load server.

e. If the message is a Request Dump Data or Dump Complete message and the message source address matches, the message goes to the dump requester.

f. If the message is a Request Dump Service message, it goes to the dump server.

g. If the message is a Memory Dump Data message and the message source address matches, it goes to the corresponding dump server.

3. If the Protocol Identifier is the Console Protocol ID:

   a. If the message is a Request System ID, Request Counters, Reserve Console, or Boot message, it goes to the console server.

   b. If the message is a Console Command or Release Console message and the message source address matches the circuit console user address, the message goes to the console server.

   c. If the message is a System ID, CSMA/CD Counters, Other Counters, or Console Response message and the message source address matches, the message goes to the corresponding console requester.

   d. If the message is a System ID message, it goes to the configuration monitor.

Note:

The two rules above are somewhat in conflict since a System ID message could pass both checks (if it comes from the console carrier target and the configuration monitor function is active). In that case, the message could be passed to both functions, or alternatively the periodic System ID message (which is sent to the Console multicast address and has a receipt number of zero) could go to the configuration monitor and the other to the Console Requester.

4. If the message is an 802.2 TEST response message and the message source address matches, the message goes to the corresponding Test requester.

5. If the message is an 802.2 XID response message and the message source address matches, the message goes to the corresponding Query requester.

6. Any other message, including a message with an undefined message type code, is quietly ignored.

Note that MOP has two open data link ports per circuit (or more when doing a Test or Query directive), due to the way the data link layer abstract service interface is defined. The distinction of which port carried the message is ignored in the rules above.

4.13 DDCMP specific algorithms

The following procedures specify the data link dependent details of using DDCMP data links with MOP.

The DDCMP architecture does not have a formal service interface definition as the other data link architectures do. The code below is written as if such interfaces did exist. The intent of each interface call is clarified by context and comments.

4.13.1 Exclusive maintenance mode

The DDCMP data link is unique among the data links supported by MOP in that it has "exclusive maintenance". This means that the data link service needed by MOP is available from DDCMP only when the normal (sequential, connection oriented) service is not active. All other data links allow both services to be used concurrently. Because of this property, additional steps are needed in the DDCMP specific algorithms to force the data link into the proper state.

Since the DDCMP spec does not have a formally defined service interface, the algorithms below use an assumed set of services. In particular:

- The type of service desired (in this case, maintenance mode) is stated on the Open call.
- Open and Close work independent of the current mode.
- The receive calls work independent of the current mode, but the receive will not complete until the data link is in maintenance mode and a maintenance mode message is received. Any normal mode traffic is invisible.

The DDCMP data link layer enters maintenance mode for two reasons:

1. A request is made to enter maintenance mode via the client interface. Subsequent transmit requests by that client are done as DDCMP Maintenance format messages.
2. A DDCMP Maintenance Mode message is received.

When DDCMP enters maintenance mode, normal (connection oriented) service is terminated. To restore normal mode service, a client using that service (e.g., the Routing layer) must close and reopen its port.

Synchronization between MOP and any normal mode DDCMP client is not shown here in detail. In outline, the necessary steps are:

1. When initiating a request (e.g., from the Dump Client), MOP must tell DDCMP to enter maintenance mode. The circuit remains in maintenance mode until the operation is finished, at which time it can be restarted in normal mode.
2. When responding to a requests (e.g., in the Load Server), DDCMP has entered maintenance mode due to the receipt of a Maintenance format message. The circuit remains in maintenance mode until the operation is finished. It may then be restarted in normal mode.

4.13.2 Open

The Open procedure opens a port for MOP on the circuit, and sets up all the necessary operating parameters.

PROCEDURE DLD.DDCMP_Open (circ: POINTER TO Circuit)

```
    RAISES {DLD.CommonExceptions}
VAR portname: LocalEntityName;          (* Holds returned PortName *)
BEGIN
    DDCMP.OPEN (circ^.Name,             (* Use our identifier for “name” *)
        circ^.LinkName,                 (* “Station” to open *)
        Maintenance,                    (* Make this a “Maintenance” type port *)
        262,                            (* Minimum acceptable buffer size *)
        portname,                       (* Scratch variable to receive port name *)
        circ^.mopport,                  (* Return port ID here *)
        circ^.SDUsize)                  (* Buffer size available on this port *)
END DLD.DDCMP_Open;
```

4.13.3 Close

The Close procedures closes the MOP port on the circuit that was previously opened by the Open procedure.

```
PROCEDURE DLD.DDCMP_Close (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
BEGIN
    DDCMP.CLOSE (circ^.mopport);        (* Call data link to close the port *)
    circ^.mopport := NIL
END DLD.DDCMP_Close;
```

4.13.4 Transmit functions

The following are the transmit related data link specific functions for the DDCMP data link.

```
PROCEDURE DLD.DDCMP_Transmit (op: POINTER TO Operation; msg: POINTER TO Buffer;
    ver: MOPVersion) RAISES {DLD.CommonExceptions}
BEGIN
    DDCMP.Transmit (op^.CircPtr^.mopport, msg)
END DLD.DDCMP_Transmit;

PROCEDURE DLD.DDCMP_TransmitPoll (op: POINTER TO Operation; msg: POINTER TO Buffer)
    RAISES {DLD.CommonExceptions}
BEGIN
    DDCMP.TransmitPoll (op^.CircPtr^.mopport, msg)
END DLD.DDCMP_TransmitPoll;
```

4.14 HDLC specific algorithms

The following procedures specify the data link dependent details of the operation of MOP over the HDLC data link. Unlike DDCMP, the HDLC data link supports “concurrent” maintenance operation, i.e., MOP can use the data link independent of other users of the data link. This is done by using the Unsequenced Data service of HDLC, which offers a datagram communication mechanism independent of the sequenced (virtual circuit) data service. (If a node needs only MOP, for example in primitive state, it suffices to implement just the unsequenced service of the HDLC data link, and omit the sequenced service support entirely.)

Detailed descriptions of the HDLC data link services used in this section may be found in the HDLC Data Link architecture specification.

4.14.1 Open

The Open procedure opens a port for MOP on the circuit, and sets up all the necessary operating parameters.

```
PROCEDURE DLD.HDLC_Open (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
VAR portname: LocalEntityName;          (* Holds returned PortName *)
BEGIN
    HDLC.OPEN (circ^.Name,              (* Use our identifier for “name” *)
        circ^.LinkName,                 (* “Station” to open *)
        Unsequenced,                    (* Ask for unsequenced service *)
        MOPProtocolID,                  (* Use the HDLC MOP protocol ID *)
        262,                            (* Minimum acceptable buffer size *)
        portname,                       (* Scratch variable to receive port name *)
        circ^.mopport,                  (* Return port ID here *)
        circ^.SDUsize)                  (* Buffer size available on this port *)
END DLD.HDLC_Open;
```

4.14.2 Close

The Close procedures closes the MOP port on the circuit that was previously opened by the Open procedure.

```
PROCEDURE DLD.HDLC_Close (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
BEGIN
    HDLC.CLOSE (circ^.mopport);         (* Call data link to close the port *)
    circ^.mopport := NIL
END DLD.HDLC_Close;
```

4.14.3 Transmit functions

The following are the transmit related data link specific functions for the HDLC data link. Note that the Transmit is done using the Unsequenced mode of HDLC.

```
PROCEDURE DLD.HDLC_Transmit (op: POINTER TO Operation; msg: POINTER TO Buffer;
    ver: MOPVersion) RAISES {DLD.CommonExceptions}
BEGIN
    HDLC.TransmitUnsequenced (op^.CircPtr^.mopport, msg)
END DLD.HDLC_Transmit;

PROCEDURE DLD.HDLC_TransmitPoll (op: POINTER TO Operation; msg: POINTER TO Buffer)
    RAISES {DLD.CommonExceptions}
BEGIN
    HDLC.TransmitPollUnsequenced (op^.CircPtr^.mopport, msg)
END DLD.HDLC_TransmitPoll;
```

4.15 LAPB specific algorithms

This section describes the data link specific aspects of using the LAPB data link. Since LAPB is a variant of HDLC, there is a lot of similarity with the HDLC section. There are some differences to account for the different service interfaces of the two data links.

LAPB is normally used as the data link protocol in connections to the X.25 public packet switched network. MOP obviously has no place in such connections. Therefore, on LAPB circuits MOP is only used for loop testing; this should only be done when the line has previously been set into some form of loopback mode or otherwise has been disconnected from the public network.

4.15.1 Open

The Open procedure opens a port for MOP on the circuit, and sets up all the necessary operating parameters.

```
PROCEDURE DLD.LAPB_Open (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
VAR portname: LocalEntityName;          (* Holds returned PortName *)
BEGIN
    LAPB.OPEN (circ^.Name,              (* Use our identifier for "name" *)
        circ^.LinkName,                 (* "Station" to open *)
        Unsequenced,                    (* Ask for unsequenced service *)
        262,                            (* Minimum acceptable buffer size *)
        portname,                       (* Scratch variable to receive port name *)
        circ^.mopport,                  (* Return port ID here *)
        circ^.SDUsize)                  (* Buffer size available on this port *)
END DLD.LAPB_Open;
```

4.15.2 Close

The Close procedures closes the MOP port on the circuit that was previously opened by the Open procedure.

```
PROCEDURE DLD.LAPB_Close (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
BEGIN
    LAPB.CLOSE (circ^.mopport);         (* Call data link to close the port *)
    circ^.mopport := NIL
END DLD.LAPB_Close;
```

4.15.3 Transmit functions

The following are the transmit related data link specific functions for the LAPB data link. Note that the Transmit is done using the Unsequenced mode of LAPB.

```
PROCEDURE DLD.LAPB_Transmit (op: POINTER TO Operation; msg: POINTER TO Buffer;
    ver: MOPVersion) RAISES {DLD.CommonExceptions}
BEGIN
    LAPB.TransmitUnsequenced (op^.CircPtr^.mopport, msg)
END DLD.LAPB_Transmit;

PROCEDURE DLD.LAPB_TransmitPoll (op: POINTER TO Operation; msg: POINTER TO Buffer;
    ver: MOPVersion) RAISES {DLD.CommonExceptions}
BEGIN
    LAPB.TransmitPollUnsequenced (op^.CircPtr^.mopport, msg)
END DLD.LAPB_TransmitPoll;
```

4.16 CSMA/CD specific algorithms

The following procedures specify the data link dependent details of the operation of MOP over the CSMA/CD (Ethernet and IEEE 802.3) data link.

The CSMA/CD service procedures called (module name CSMACD) are defined in the services chapter of the CSMA/CD Data Link architecture. For convenience, we will treat them as procedures that signal any errors (rather than as functions that return a value indicating success or failure).

4.16.1 Open

This procedure shows how MOP sets up the data link ports for its use. This is done when MOP begins to perform its tasks. The equivalent setup, such as protocol and multicast address setup, is also necessary whenever the set of enabled functions is changed (via the Enable and Disable management directives); that detail is not shown here.

```
PROCEDURE DLD.CSMACD_Open (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
VAR portname: LocalEntityName;              (* Holds returned PortName *)
BEGIN
    CSMACD.OpenPort (circ^.Name,            (* Use our identifier for “name” *)
        circ^.LinkName,                     (* “Station” to open *)
        Class1,                             (* Use Class 1 LLC service *)
        TRUE,                               (* Use “padded” form for Ethernet frames *)
        circ^.mopport,                      (* Return port ID here *)
        portname);                          (* Scratch variable to receive port name *)
    CSMACD.EnableProtocolType (circ^.mopport, ConsoleProtocolType);
                                            (* Enable protocol type *)
    CSMACD.EnableProtocolID (circ^.mopport, ConsoleProtocolID);
                                            (* and corresponding Protocol ID *)
    IF LoadServer IN circ^.Functions OR     (* If the Load/Dump functions are enabled...  *)
        LoadClient IN circ^.Functions OR
        DumpServer IN circ^.Functions OR
        DumpClient IN circ^.Functions       (* ...  then its Protocol Type is enabled *)
    THEN
        CSMACD.EnableProtocolType (circ^.mopport, LoadProtocolType);
        CSMACD.EnableProtocolID (circ^.mopport, LoadProtocolID)
    END;
    IF LoadServer IN circ^.Functions OR     (* If the Load/Dump server is enabled...  *)
        DumpServer IN circ^.Functions       (* ...  then enable the assistant multicast address *)
    THEN
        CSMACD.EnableMACAddress (circ^.mopport, LoadAssistant)
    END;
    CSMACD.OpenPort (circ^.Name,            (* Use our identifier for “name” *)
        circ^.LinkName,                     (* “Station” to open *)
        Class1,                             (* Use Class 1 LLC service *)
        FALSE,                              (* Do not use “padded” form for Ethernet frames *)
        circ^.loopport,                     (* Return port ID here *)
        portname);                          (* Scratch variable to receive port name *)
    CSMACD.EnableProtocolType (circ^.loopport, LoopProtocolType);
                                            (* Enable protocol type *)
    CSMACD.EnableProtocolID (circ^.loopport, LoopProtocolID);
                                            (* and corresponding procol ID *)
    (* optionally: *)
        CSMACD.EnableMACAddress (circ^.loopport, LoopAssistant)
    (* end *)
```

```
    CSMACD.OpenPort (circ^.Name,         (* Use our identifier for “name” *)
        circ^.LinkName,                  (* “Station” to open *)
        UserSupplied,                    (* Need user LLC to issue XID or TEST frames *)
        FALSE,                           (* "padded" is irrelevant for this port *)
        circ^.LLCport,                   (* Return port ID here *)
        portname);                       (* Scratch variable to receive port name *)
    (* Note that we enable no SAP (or anything else) at this time, so this port  is for the moment unused *)
    circ^.DataLinkAddress := « the MAC address for this data link »;
    circ^.SDUSize := 1492;               (* 802.2 SNAP PDU size *)
END DLD.CSMACD_Open;
```

4.16.2 Close

The Close procedures closes the ports on the circuit that were previously opened by the Open procedure.

```
PROCEDURE DLD.CSMACD_Close (circ: POINTER TO Circuit)
    RAISES {DLD.CommonExceptions}
BEGIN
    CSMACD.Close (circ^.loopport);       (* Call data link to close the loop port *)
    circ^.loopport := NIL;
    CSMACD.Close (circ^.mopport);        (* ...  and the other port *)
    circ^.mopport := NIL
END DLD.CSMACD_Close;
```

4.16.3 Transmit functions

The following are the transmit related data link specific functions for the CSMA/CD data link. The major issue here is the selection of the correct SNAP Protocol ID or Ethernet Protocol Type, according to the protocol version being used and the specific message being transmitted.

```
PROCEDURE DLD.CSMACD_Transmit (op: POINTER TO Operation; msg: POINTER TO  Buffer;
    ver: MOPVersion) RAISES {DLD.CommonExceptions}
BEGIN
    IF ver = V4
    THEN                                 (* Set up the correct Protocol ID *)
        msg^.LANFormat.format := SNAP;
        IF op^.OpType IN {LoopRequester, LoopServer}
        THEN msg^.LANFormat.ProtID := LoopbackProtocolID
        ELSEIF msg^.Data[0] IN {MemLoadTA, DumpComplete, MemLoad, Assistance,  RequestDump,
                                 RequestProgram, RequestLoad, RequestDumpService,
                                 DumpData, ParameterLoad}
        THEN msg^.LanFormat.ProtID := LoadProtocolID
        ELSE msg^.LanFormat.ProtID := ConsoleProtocolID
        END
    ELSE                                 (* V3 format, set up correct Protocol Type *)
        msg^.LANFormat.format := Ethernet;
        IF op^.OpType IN {LoopRequester, LoopServer}
        THEN msg^.LANFormat.type := LoopbackProtocolType
        ELSEIF msg^.Data[0] IN {MemLoadTA, DumpComplete, MemLoad, Assistance,  RequestDump,
                                 RequestProgram, RequestLoad, RequestDumpService,
                                 DumpData, ParameterLoad}
        THEN msg^.LanFormat.type := LoadProtocolType
        ELSE msg^.LanFormat.type := ConsoleProtocolType
        END
    END;
    msg^.Source := op^.CircPtr^.DataLinkAddress;
```

```
    IF op^.OpType IN {LoopRequester, LoopServer}
    THEN
        CSMACD.Transmit (op^.CircPtr^.loopport, msg^.Destination, msg^.Source,
            msg^.LANFormat, msg)
    ELSE
        CSMACD.Transmit (op^.CircPtr^.mopport, msg^.Destination, msg^.Source,
            msg^.LANFormat, msg)
    END
END DLD.CSMACD_Transmit;

PROCEDURE DLD.CSMACD_TransmitPoll (op: POINTER TO Operation; msg: POINTER TO  Buffer)
    RAISES {DLD.CommonExceptions}
BEGIN
    IF op^.OpType IN {LoopRequester, LoopServer}
    THEN
        CSMACD.TransmitPoll (op^.CircPtr^.loopport, msg)
    ELSE
        CSMACD.TransmitPoll (op^.CircPtr^.mopport, msg)
    END
END DLD.CSMACD_TransmitPoll;
```

4.17 FDDI specific algorithms

The FDDI specific algorithms are identical to those specified for CSMA/CD, except that the service interface calls of the FDDI Data Link architecture specification is used. These offer the same semantics as the service interface calls of the CSMA/CD Data Link. Note that the maximum data link SDU size is 1492, even though FDDI allows larger frames. This ensures that communication across LAN bridges will work.

4.18 Miscellaneous data link independent procedures

The procedures in this section are miscellaneous utility procedures used in the Modula-2-Plus descriptions of the MOP algorithms. These are independent of the data link being used.

4.18.1 Resource allocation/deallocation

These procedures allocate and deallocate resources used by MOP operation threads.

```
PROCEDURE DLI.Alloc (VAR op: POINTER TO Operation; VAR buff: POINTER TO  Buffer)
    RAISES {InsufficientResources};
BEGIN
    NEW (buff);
    NEW (op);
    « insert the new Operation record in the list of active operations »
END DLI.Alloc;

PROCEDURE DLI.Free (op: POINTER TO Operation; buff: POINTER TO Buffer);
BEGIN
    « remove Operation record from the list of active operations »;
    FREE (buff);
    FREE (op)
END DLI.Free;
```

4.18.2 Manipulate fields in the buffer contents

The procedures below are used in the Modula-2-Plus algorithm descriptions to examine or load fields in message buffers.

PROCEDURE DLI.GetInteger (buff: POINTER TO Buffer; off: INTEGER) : INTEGER;
« return the value of the 16-bit integer whose low order byte is at the specified offset, and high order byte at the byte following that in the buffer. »
END DLI.GetInteger;

PROCEDURE DLI.PutInteger (buff: POINTER TO Buffer; off: INTEGER; value: INTEGER);
« store the 16-bit integer (*value*), with low order byte at the specified offset and high order byte at the byte following that, into the buffer. »
END DLI.PutInteger;

PROCEDURE DLI.GetLong (buff: POINTER TO Buffer; off: INTEGER) : INTEGER;
« return the value of the 32-bit integer whose low order byte is at the specified offset, and higher order bytes at the 3 bytes following that in the buffer. »
END DLI.GetLong;

PROCEDURE DLI.PutLong (buff: POINTER TO Buffer; off: INTEGER; value: INTEGER);
« store the 32-bit integer (*value*), with low order byte at the specified offset and higher order bytes at the 3 bytes following that, into the buffer. »
END DLI.PutLong;

PROCEDURE DLI.GetAddress (buff: POINTER TO Buffer; off: INTEGER) : LANAddress;
« return the value of the 6-byte LAN address whost first byte is at the specified offset in the buffer. »
END DLI.GetAddress;

PROCEDURE DLI.PutAddress (buff: POINTER TO Buffer; off: INTEGER; value: LANAddress);
« store the 6-byte LAN address (*value*), with its first byte at the specified offset, into the buffer. »
END DLI.PutAddress;

4.18.3 Find a Client record

These are the procedures that find a Client record, i.e., an entry in the Client database. There are three ways to get an entry:

1. By name — this is simply a search for a Client record whose Name attribute matches the supplied name. This case applies for management requests where the directive specifies the name of the Client record to use.

2. By circuit — here we know what circuit is to be used, and we need to find the Client record to use. This case applies to server functions, which use the Client record to find out how to respond to a received request message.

   Note that it is possible to have multiple matches (more than one Client record satisfies the search criteria). In that case any of the matching Client records may be used. This is usually undesirable; the network manager should take care to avoid such client database setups.

   There are two variants of lookup by circuit:

   a. Lookup by circuit and address — this searches for a Client record whose Circuit attribute points to the circuit in question and, for a LAN, with an Addresses attribute value that contains the specified address. (For point to point circuits the match is by Circuit only.)

b. Lookup by circuit and device type — this searches for a Client record whose Circuit attribute points to the specified circuit and a Device Types attribute that contains the specified device type. (This variant is not used for point to point circuits.)

```
PROCEDURE DLI.FindClientByName(cl: SimpleName) : POINTER TO Client
    RAISES {UnrecognizedClient}
BEGIN
    « return a pointer to the Client entity whose Name field equals the name in the cl argument »
END DLI.FindClientByName;

PROCEDURE DLI.FindClientByCircAddr (cir: POINTER TO Circuit; addr: LANAddress)
    : POINTER TO Client;
BEGIN
    « return a pointer to a matching Client entity, or NIL if no such entity exists. »
END DLI.FindClientByCircAddr;

PROCEDURE DLI.FindClientByCircDev (cir: POINTER TO Circuit; dev: DevType) : POINTER TO Client;
BEGIN
    « return a pointer to a matching Client entity, or NIL if no such entity exists. »
END DLI.FindClientByCircDev;
```

4.18.4 Find a Circuit record

```
PROCEDURE DLI.FindCircuitByName(cir: SimpleName) : POINTER TO Circuit
    RAISES {UnrecognizedCircuit}
BEGIN
    « return a pointer to the Circuit entity whose Name field equals the name in the cir argument »
END DLI.FindCircuitByName;
```

4.18.5 Pick a Source SAP address

The Source SAP used in Test and XID requests is selected in an implementation specific fashion. Unfortunately, there is no standard SAP address assigned for the purpose we need here. Therefore the SSAP has to be a "locally administered" value. We can't pick one architecturally, since locally administered values are selected by the local network manager, not by Digital architects. Therefore the only way out is for the implementation to select a value dynamically.

Any value is acceptable provided that it is:

1. Not currently in use on the circuit, and
2. An individual SAP (low order bit is zero), and
3. A locally administered SAP (second bit zero), and
4. Not the null SAP (i.e., not the all-zero SAP address)
5. It may also be a good idea to avoid SAP address values that other vendors have architecturally assigned to their proprietary protocols

```
PROCEDURE LAN.SelectaSAP (op: POINTER TO Operation) : SAPAddress;
BEGIN
    RETURN « a suitable SAP address »
END LAN.SelectaSAP;
```

Chapter 5

Messages

This chapter defines the binary formats of the protocol messages that support the operation described in the operation chapter. In order to operate correctly on exclusive maintenance channels, message identification codes are taken from a single space. Values 16 and 18 are reserved obsolete values. Some data links have a minimum message size and many of the maintenance protocol messages are quite small. Padding must be requested from the particular data link on the assumption that such a service is provided if needed. The actual size of received messages is also provided by the Data Link Layer, so that messages with a single variable length data field need not include a size field.

The following notation is used to describe the messages:

FIELD (LENGTH) : CODING =

Description of field

Where:

FIELD is the name of the field being described.

LENGTH is the length of the field expressed as one of the following:

- The number of 8-bit bytes.
- The notation "C–*n*" meaning counted image field with *n* being the maximum length in 8-bit bytes of the image. The first byte of the field specifies the number of bytes in the remainder of the field. Therefore, the minimum total length of the field is one byte. The first byte of the field may contain information in addition to the count.

  Note that the maximum length *includes* the one byte field that contains the count. Therefore, the count byte of a C-*n* field can have a value in the range 0 to *n*–1, indicating 0 to *n*–1 additional bytes of data follow the count byte.
- The notation "J–*n*" meaning image field with *n* being the maximum length in 8-bit bytes of the image. The image is preceded by sufficient information to compute the length of the field. Image fields are variable length and may be null (length=0). All 8 bits of each byte may be used as information bits.
- An asterisk (*), indicating that the field consists of the remainder of the message, i.e., the total message length less the length of all of the other fields.

CODING is the representation type used, as follows:

- B — Binary
- BM — Bit map (each bit has independent meaning)
- A — ASCII
- Null — Interpretation depends on data representation

Notes:

- All numeric values are shown in decimal unless otherwise noted.
- Fields are transmitted in the order shown, left to right.
- All fields are transmitted low-order or least significant byte first unless otherwise specified.
- Bits in a field are numbered from 0 to n where 0 is the low-order or least-significant bit.
- In the packet formats shown below, fields shown in dashed boxes are optional or are present only in certain cases.

A summary of the defined message codes and the associated MOP functional component and NI Protocol Type (last two bytes of Protocol ID) is shown in Table 16 below.

Table 16: Summary of MOP message codes

Message code	Message Name	Protocol Type	Protocol Name	Section
0	Memory Load with Transfer Address	60–01	Dump/Load	5.1.4
1	Dump Complete	60–01	Dump/Load	5.2.4
2	Memory Load	60–01	Dump/Load	5.1.5
3	Assistance Volunteer	60–01	Dump/Load	5.1.2
4	Request Memory Dump	60–01	Dump/Load	5.2.2
5	Request ID	60–02	Console	5.5.2
6	Boot	60–02	Console	5.5.1
7	System ID	60–02	Console	5.5.3
8	Request Program	60–01	Dump/Load	5.1.1
9	Request Counters	60–02	Console	5.5.4
10	Request Memory Load	60–01	Dump/Load	5.1.3
11	Counters (CSMA/CD only)	60–02	Console	5.5.5
12	Request Dump Service	60–01	Dump/Load	5.2.1
13	Reserve Console	60–02	Console	5.5.7
14	Memory Dump Data	60–01	Dump/Load	5.2.3
15	Release Console	60–02	Console	5.5.10
16	Reserved			
17	Console Command and Poll	60–02	Console	5.5.8
18	Reserved			
19	Console Response and Acknowledge	60–02	Console	5.5.9
20	Parameter Load with Transfer Address	60–01	Dump/Load	5.1.6
21	Counters (other than CSMA/CD)	60–02	Console	5.5.6
24	Point to Point Loop Data	N/A	Loop Test	5.3.1
26	Point to Point Looped Data	N/A	Loop Test	5.3.2
1	LAN Loop Reply	90–00	Loop Test	5.4.1
2	LAN Loop Forward Data	90–00	Loop Test	5.4.2

5.1 Downline Load messages

The messages described in this section relate to the MOP Load/Dump components, for the Downline Load service. On LANs, they are sent using the Load/Dump Protocol ID (08–00–2B–60–01).

5.1.1 Request Program

The Request Program message consists of:

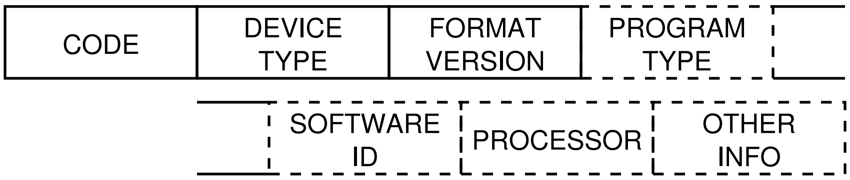

Figure 5: Request Program message format

Where:

CODE (1) : B = The number 8.

DEVICE TYPE (1) : B = The device type at the requesting system. Used by the load server to search for a client database entry by device type. Defined device types are found in Appendix A.1.

FORMAT VERSION (1) : B = The protocol format version. For version 4.0, the value is 4.

PROGRAM TYPE (1) : B = The generic type of program being requested. The defined values are as follows:

Value	Meaning
0	Secondary loader
1	Tertiary loader
2	System Image
3	Management image
4	CMIP Script file

This field and all the following can be omitted, in which case the default for this field is 0. A system image is whatever is to end up in the requesting system, and could be any type of program. A management image indicates information that is specific to an individual requesting system, while the system program in this context refers to a generic software product that can run in any system of a particular type.

SOFTWARE ID (C–128) : = Identification of the software being requested. Omitted if PROGRAM TYPE is omitted. The format is the same as defined for the Boot message (Section 5.5.1). Note that Software ID codes of –1 (standard operating system) and –2 (maintenance system) are meaningful only for Program Type = 2 (System Image)

PROCESSOR (1) : B = The processor to be loaded. This field is used to distinguish a communications “front end” processor on a system where it is independent of the main CPU.

Value	Meaning
0	System processor
1	Communication processor

OTHER INFO (*) : = Further information to identify the requesting system. Definition is as described for the Remote Console System ID message (Section 5.5.3).

The primary use of this field is for the DATA LINK BUFFER SIZE parameter. Implementations are strongly encouraged to supply this field, since the load server will otherwise assume a buffer size of 262, resulting in much reduced performance.

Note: some implementations include the Device Type item in this field. Since that information is provided already in the DEVICE TYPE field, including it here is redundant. For compatibility with certain existing implementations that have *different* values in the two fields, a Load Server shall use only the DEVICE TYPE field and not any Device Type item included in the Other Info field. However, conforming implementations shall send consistent requests. (The recommended practice is to send the Device Type only in the Device Type field; optionally, the Device Type may additionally be sent in the OTHER INFO field, but it must then be the same value in both places.)

5.1.2 Assistance Volunteer

The Assistance Volunteer message consists of:

CODE

Figure 6: Assistance Volunteer message format

Where:

CODE (1) : B = The number 3.

5.1.3 Request Memory Load

The Request Memory Load message consists of:

CODE	LOAD NUMBER	RESERVED

Figure 7: Request Memory Load message format

Where:

CODE (1) : B = The number 10.

LOAD NUMBER (1) : B = The number of the load segment being requested, as defined for the Memory Load with Transfer Address message (Section 5.1.4).

RESERVED (1) : B = A reserved byte. Send 0; ignore on receipt.

Note: for compatibility with certain existing implementations, load servers shall accept Request Memory Load messages where this field is missing. However, all conforming implementations shall send this field.

5.1.4 Memory Load with Transfer Address

The Memory Load with Transfer Address message consists of:

CODE	LOAD NUMBER	LOAD ADDRESS	IMAGE DATA	TRANSFER ADDRESS

Figure 8: Memory Load with Transfer Address message format

Where:

CODE (1) : B = The number 0.

LOAD NUMBER (1) : B = The load number for multi-segment loads. This message may be preceded by Memory Load without transfer address messages. The load number starts at zero and is incremented for each load message sent in a loading sequence. The load number is modulo 256. After load number 255 is load number 0.

LOAD ADDRESS (4) : B = The memory load address (physical) for storage of the data image.

IMAGE DATA (*) : = The image to be stored into computer memory. The form sent can be machine-dependent, to be defined on an as needed basis. Unless otherwise defined, each byte represents one memory byte.

TRANSFER ADDRESS (4) : B = The starting memory address of the image just loaded.

Note:

IMAGE DATA or LOAD ADDRESS and IMAGE DATA may be omitted. Valid message lengths are 6 (LOAD ADDRESS and IMAGE DATA omitted), 10 (IMAGE DATA omitted), or greater than 10. IMAGE DATA is present only when loading a Secondary Loader.

5.1.5 Memory Load

The Memory Load message consists of:

CODE	LOAD NUMBER	LOAD ADDRESS	IMAGE DATA

Figure 9: Memory Load message format

Where:

CODE (1) : B = The number 2.

LOAD NUMBER (1) : B = As described for Memory Load with Transfer Address (see Section 5.1.4).

LOAD ADDRESS (4) : B = As described for Memory Load with Transfer Address.

IMAGE DATA (*) : = As described for Memory Load with Transfer Address.

Note:

IMAGE DATA may be omitted. Valid message lengths may be 6 (IMAGE DATA omitted), or greater than 6. Messages without IMAGE DATA cause nothing to be loaded; however, the LOAD NUMBER value is still incremented for the next load.

5.1.6 Parameter Load with Transfer Address

The Parameter Load with Transfer Address message consists of:

CODE	LOAD NUMBER	PARAMETERS	END MARK	TRANSFER ADDRESS

Figure 10: Parameter Load with Transfer Address message format

Where:

CODE (1) : B = The number 20.

LOAD NUMBER (1) : B = As described for the Memory Load with Transfer Address message (Section 5.1.4).

PARAMETERS (*) : = Zero or more parameter entries.

A parameter entry consists of:

PARAMETER TYPE	PARAMETER LENGTH	PARAMETER VALUE

Figure 11: Parameter Load message Parameter field format

Where:

PARAMETER TYPE (1) : B = A type code for the parameter information. The values are:

Value	Parameter
1	PHASE IV CLIENT NAME
2	PHASE IV CLIENT ADDRESS
3	PHASE IV HOST NAME
4	PHASE IV HOST ADDRESS
5	V3 FORMAT HOST SYSTEM TIME
6	HOST SYSTEM TIME

PARAMETER LENGTH (1) : B = The number of bytes in the PARAMETER VALUE field.

PARAMETER VALUE (J–255) : = A value according to PARAMETER TYPE and PARAMETER LENGTH. Refer to Appendix G.5 for more detail on parameters 1–5.

Where:

PHASE IV CLIENT NAME (J–16) : A = ASCII system name target system is to use for itself.

PHASE IV CLIENT ADDRESS (J–2) : B = Binary system address target system is to use for itself.

PHASE IV HOST NAME (J–16) : A = ASCII system name of host assigned to system (for example, host for task loading of core only systems).

PHASE IV HOST ADDRESS (J–2) : B = Binary system address of host.

V3 FORMAT HOST SYSTEM TIME (10) : B = Segmented binary system time of host, consisting of:
CENTURY YEAR MONTH DAY HOUR MINUTE SECOND 100TH TDFH TDFM

where:

CENTURY (1) : B = the century base for reckoning the absolute year. Value is a positive integer (0 through +127).

YEAR (1) : B = the year of the base century. Value is in the range 0 through 99.

MONTH (1) : B = the month of the year, starting with January = 1. Value is in the range 1 through 12.

DAY (1) : B = the day of the month. Value is in the range 1 through 31.

HOUR (1) : B = the hour of the day. Value is in the range 0 through 23.

MINUTE (1) : B = the minute of the hour. Value is in the range 0 through 59.

SECOND (1) : B = the second of the minute. Value is in the range 0 through 59.

100TH (1) : B = the number of hundredths of a second. Value is in the range 0 through 99.

TDFH (1) : B = the hours portion of the Time Differential Factor. Value is in the range -12 through +13.

TDFM (1) : B = the minutes portion of the Time Differential Factor. Value is in the range -59 through 59. The sign of this value must be the same as the sign of the TDFH value.

HOST SYSTEM TIME (16) : B = The host system time, encoded in Binary Absolute Time format.

END MARK (1) : B = The number 0.

TRANSFER ADDRESS (4) : B = As described for the Memory Load with Transfer Address message (Section 5.1.4).

5.2 Upline Dump messages

The messages described in this section relate to the MOP Load/Dump components, for the Upline Dump service. On LANs, they are sent using the Load/Dump Protocol ID (08–00–2B–60–01).

5.2.1 Request Dump Service

The Request Dump Service message consists of:

CODE	DEVICE TYPE	FORMAT VERSION	MEMORY SIZE	BITS	OTHER INFO

Figure 12: Request Dump Service message format

Where:

CODE (1) : B = The number 12.

DEVICE TYPE (1) : B = As described for the Request Program message.

FORMAT VERSION (1) : B = As described for the Request Program message (Section 5.1.1).

MEMORY SIZE (4) : B = The size of physical machine memory. Units are as described for COUNT in the Request Memory Dump message (Section 5.2.2).

BITS (1) : B = The number 2. Present for compatibility only; ignore on receipt.

OTHER INFO (*) : = Further information to identify the requesting system. Definition is as described for the Remote Console System ID message (Section 5.5.3). The only valid INFO TYPE is DATA LINK BUFFER SIZE (401).

Implementations are strongly encouraged to supply this field, since the dump server will otherwise assume a buffer size of 262, resulting in much reduced performance.

5.2.2 Request Memory Dump

The Request Memory Dump message consists of:

CODE	MEMORY ADDRESS	COUNT

Figure 13: Request Memory Dump message format

Where:

CODE (1) : B = The number 4.

MEMORY ADDRESS (4) : B = The starting physical memory address for the dump.

COUNT (2) : B = The number of locations to dump. The meaning of the count can be machine-dependent, to be defined on an as needed basis. Unless otherwise defined, the count is in bytes.

Note:

This request results in a single Memory Dump Data message. A dump should not be requested for more data than can be reliably sent in a single reply on the link used. The maximum data link message length limits the maximum length for a given link.

5.2.3 Memory Dump Data

The Memory Dump Data message consists of:

CODE	MEMORY ADDRESS	IMAGE DATA

Figure 14: Memory Dump Data message format

Where:

CODE (1) : B = The number 14.

MEMORY ADDRESS (4) : B = As described for the Request Memory Dump message (Section 5.2.2).

IMAGE DATA (*) : = As described for the Memory Load with Transfer Address message (Section 5.1.4). The length of this field must not exceed the COUNT specified in the Request Memory Dump message. Note that the field may be shorter (and may be null).

5.2.4 Dump Complete

The Dump Complete message consists of:

CODE

Figure 15: Dump Complete message format

Where:

CODE (1) : B = The number 1.

5.3 Point to Point Loop Test messages

The messages shown in this section are used by the MOP Loop components. They apply only to Loop operations on point to point data links.

5.3.1 Loop Data Message for point to point links

The Loop Data message for point to point links consists of:

CODE	RECEIPT	DATA

Figure 16: Loop Data message format for point to point links

Where:

CODE (1) : B = The number 24.

RECEIPT (2) : B = The receipt number for the loop request.

DATA (*) : B = The data to be looped back.

5.3.2 Looped Data Message for point to point links

The Looped Data message for point to point links consists of:

CODE	RECEIPT	DATA

Figure 17: Looped Data message format for point to point links

Where:

CODE (1) : B = The number 26.

RECEIPT (2) : B = The receipt number from the Loop Data message.

DATA (*) : B = The data from the Loop Data message.

5.4 LAN Loop Test messages

The messages shown in this section are used by the MOP Loop components. They apply only to Loop operations on LAN data links. The Protocol ID used is the Loopback Protocol ID (08–00–2B–90–00).

5.4.1 Forward Data message for LANs

The Forward Data message for LAN data links consists of:

SKIP COUNT	skipped	CODE	ADDRESS	DATA

Figure 18: Forward Data message format for LAN links

Where:

SKIP COUNT (2) : B = The number of bytes to skip over preceding the CODE field.

CODE (2) : B = The number 2.

ADDRESS (6) : B = The address to which to forward the message.

DATA (*) : B = The data to be interpreted by the station specified by the ADDRESS field.

5.4.2 Looped Data message for LANs

The Looped Data message (Reply message) for LAN data links consists of:

SKIP COUNT	skipped	CODE	RECEIPT	DATA

Figure 19: Looped Data message format for LAN links

Where:

SKIP COUNT (2) : B = The number of bytes to skip over preceding the CODE field.

CODE (2) : B = The number 1.

RECEIPT (2) : B = The receipt number.

DATA (*) : B = The loop test data supplied by the Loop Requester.

5.5 Console messages

The messages described in this section relate to the MOP Console components. On LANs, they are sent using the Console Protocol ID (08–00–2B–60–02). Note that the Boot message is in the category of Console messages, even though it is often used in conjunction with a downline load operation.

5.5.1 Boot

The Boot message consists of:

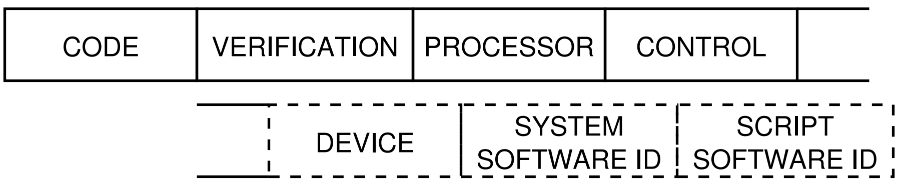

Figure 20: Boot message format

Where:

CODE (1) : B = The number 6.

VERIFICATION (8) : B = A verification code that must match before the receiving system can honor the request. Note that there is *no* wild card value (the value %x0000000000000000 is used as a default, but has no other special significance).

PROCESSOR (1) : B = As described for the Request Program message (Section 5.1.1).

Note:

The intent of this field is to specify which processor is to be booted. The state of the other processor, if any, shall not be disturbed in the process. This allows a host to boot one component without in the process forcing a complete reinitialization of other components.

CONTROL (1) : BM = Instructions to the system as to what device to use for the operation. Values are:

Bit	Meaning	Value	Meaning
0	Boot server	0	System default
		1	Requesting system
1	Boot device	0	System default
		1	Specified device

Note:

Settings “Requesting system” (bit 0 = 1) and “Specified device” (bit 1 = 1) are mutually exclusive.

DEVICE ID (C–17) : = The device to use. Present only if CONTROL<Boot-device> = Specified Device. Interpretation is specific to the receiving system.

SYSTEM SOFTWARE ID (C–128) : = The System software the system is to load (i.e., the value to send in the Software ID field of the Request Program message that requests Program Type = System). Software identification consists of:

FORM	ID

Figure 21: Software ID field format

Where:

FORM (1) : B = The general type of software. Values are:

Value	Meaning
0	No software id
>0	The length of the ID field
–1	Standard operating system
–2	Maintenance system

ID (J–127) : A = A specific software ID. Present only if FORM > 0. The interpretation of this ID is specific to the receiving system.

The ID string is required to consist only of the letters A through Z (hex values 41 through 5A), digits 0 to 9 (hex values 30 through 39), and underscore (hex value 5F). In addition, Software IDs assigned by Digital must contain one $ sign (hex 24) preceded by the Facility name for the product. Software IDs assigned by customers shall not contain a $ sign. A Load Host receiving a Request Program message with a Software ID field that does not meet this requirement shall ignore the message.

As an example, the Software ID string can be interpreted as a filename, with the device name and directory supplied by the implementation or host installation. The facility prefix (terminated by the $ sign) used in Digital-defined Software IDs could be used to identify a subdirectory. For example, the following mapping could be done:

FAC$XYZ	/sys/mop/images/digital/fac/xyz
XYZ	/sys/mop/images/other/xyz

SCRIPT SOFTWARE ID (C–128) : = The Script software the system is to load (i.e., the value to send in the Software ID field of the Request Program message that requests Program Type = CMIP Script). The format is the same as that of the System Software ID field, except that Form = –2 (“Maintenance system”) is not used.

5.5.2 Request ID

The Request ID message consists of:

CODE	RESERVED	RECEIPT

Figure 22: Request ID message format

Where:

CODE (1) : B = The number 5.

RESERVED (1) : = A one byte field reserved to Digital for future use. Send 0; ignore on receipt.

RECEIPT (2) : B = A receipt number to identify the request. The value 0 is reserved for unsolicited System ID messages (periodically transmission of System ID, as required by the Auto-configuration).

5.5.3 System ID

The System ID message consists of:

CODE	RESERVED	RECEIPT	OTHER INFO	PAD

Figure 23: System ID message format

Where:

CODE (1) : B = The number 7.

RESERVED (1) : = A one byte field reserved to Digital for future use. Send 0; ignore on receipt.

RECEIPT (2) : B = A receipt number to identify the request. The value 0 is used to indicate an unsolicited System ID message.

OTHER INFO (*) : = Further information to describe the system. Consists of zero or more entries *in any order*. Each entry consists of:

INFO TYPE	INFO LENGTH	INFO VALUE

Figure 24: System ID message Info field format

Where:

INFO TYPE (2) : B = Is the type of information. When processing the information elements of this message, implementations shall ignore any that have an unrecognized value in the INFO TYPE field. Such fields are processed by skipping the field (using the INFO LENGTH) and continuing processing with the next field, if any. The fields shown as "*xxx* Related" (101–199, 201–299, 402–499) are qualified by the field they relate to. For example, if field type 100 has the value 37 (DELQA), then the meanings of field types 101–199 (if any) is specific to the DELQA, and the code assignments are found in the specification for that device. The values are:

Value	Information	
1	*	MAINTENANCE VERSION
2	*	FUNCTIONS
3	**	CONSOLE USER
4	**	RESERVATION TIMER
5	**	CONSOLE COMMAND SIZE
6	**	CONSOLE RESPONSE SIZE

7	*	HARDWARE ADDRESS
8		*reserved*
9	***	NODE ID
10		SYSTEM TIME
11	***	NODE NAME PART 1
12	***	NODE NAME PART 2
13	§	STATION ID
100	*	COMMUNICATION DEVICE
101–199		COMMUNICATION DEVICE Related
200		SOFTWARE ID
201–299		SOFTWARE ID Related
400	¤	DATA LINK
401		DATA LINK BUFFER SIZE
402–499		DATA LINK Related

* Required field (System ID message only).
** Required field if console carrier available. (FUNCTIONS bit 5).
*** Required field when the system is not in Primitive state. Optional in Primitive state.
§ Required field if Data Link is FDDI.
¤ Required field if Data Link is other than CSMA/CD. Recommended field if Data Link is CSMA/CD.

INFO LENGTH (1) : B = The number of bytes in the INFO VALUE field. This value shall be large enough to accommodate the value, i.e., at least as large as the size shown below for each value. Note that it is possible for the field to be larger than the minimum required.

INFO VALUE : = The value according to INFO TYPE and INFO LENGTH.

Where:

MAINTENANCE VERSION (3) : B = The MOP version number. The bytes, in order from low to high, are version, ECO, and user ECO. The values are 4, 0 and 0.

FUNCTIONS (2) : BM = The maintenance functions currently available through this link. The bit meanings are:

Bit	Function
0	Loop Server (required function)
1	Dump Client
2	Primary loader Client (can only load secondary loader)
3	Multi-block loader Client (can load tertiary loader, management data, CMIP script, or system)
4	Boot (console server)
5	Console carrier Server
6	Data link counters (required function[1], console server)
7	Console carrier reserved

Note that bit 7 (Console Carrier Reserved) indicates the current state of the console reservation, while all other bits indicate whether the corresponding function is supported or not.

[1] Note: for the FDDI data link this function is required, and should be reported, only when sending System ID messages in MOP V4 format. When sending in V3 format, this bit should be zero. Refer to Appendix G for more detail.

CONSOLE USER (6) : B = System address of the system that has the console reserved. Not present if not applicable. Must be present if console carrier is available, i.e., FUNCTION bit 5 is ON. Not valid if the console carrier is not reserved, i.e., FUNCTION bit 7 is OFF.

RESERVATION TIMER (2) : B = The maximum value, in seconds, of the timer used to clear unused console reservations. Not present if not applicable. Must be present if console carrier is available, i.e., FUNCTION bit 5 is ON.

CONSOLE COMMAND SIZE (2) : B = The maximum size of the console command buffer. Not present if not applicable. Must be present if console carrier is available, i.e., FUNCTION bit 5 is ON.

CONSOLE RESPONSE SIZE (2) : B = The maximum size of the console response buffer. Not present if not applicable. Must be present if console carrier is available, i.e., FUNCTION bit 5 is ON.

HARDWARE ADDRESS (6) : B = The default data link address for the circuit on which this message is sent, i.e., the value in the MAC Address ROM of this circuit.

NODE ID (6) : B = The Node ID of this system (i.e., the value of the Node entity status attribute NodeID). Required when the system is not in primitive state.

SYSTEM TIME (16) : B = The system time, represented as a Binary Absolute Time according to the Digital Standard for Representation of Time.

NODE NAME PART 1 (J–255) : = The Node Name of this node, encoded as a Name Service FullName. If the node name has not been set yet, or the ability to set the node name is not supported in this node, this field may be sent as null, or may be omitted. If the complete node name is longer than 255 bytes, the first 255 bytes are sent in this field and the remainder is sent in NODE NAME PART 2.

NODE NAME PART 2 (J–255) : = The portion of the Node Name of this node, encoded as a Name Service FullName, beyond the first 255 bytes. If the Node Name does not exceed 255 bytes, or the node does not support the ability to set a node name, this argument may be omitted or may be sent as null.

STATION ID (8) : B = The Station ID of this system FDDI Station (i.e., the value of the FDDI Data Link entity status attribute Station ID for this circuit). Required on FDDI circuits in all node states. Not present for circuits other than FDDI.

COMMUNICATION DEVICE (1) : B = The hardware device type of the link being used. Values are in Appendix A.1.

COMMUNICATION DEVICE RELATED : = Information specific to the particular COMMUNICATION DEVICE. Not present if not applicable. Length and encoding vary according to the specific definition of these fields for a given COMMUNICATION DEVICE value.

SOFTWARE ID (C–128) : = The identification of the software the system is supposed to be running. The format is the same as defined for the Boot message (Section 5.5.1).

SOFTWARE ID RELATED : = Information specific to the particular SOFTWARE ID. Not present if not applicable. Interpretation is specific to the receiving system (e.g., file specification, which may vary depending on the type of file server). Length and encoding vary according to the specific definition of these fields for a given SOFTWARE ID value.

DATA LINK (1) : B = The type of data link protocol on the link being used. Values are in Appendix A.2.

DATA LINK BUFFER SIZE (2) : B = The size of the data link buffer. Not present if not applicable. The default value is 262. This specifies the length of the data passed from MOP to the data link layer; it includes the MOP header but not the data link envelope. A server that does not recognize this field may ignore this field, so that all requesters must support 262 byte messages. In both Ethernet and 802 (LLC SNAP) format, it includes only the length of the LLC Data, i.e., the MOP protocol header and data, but not any part of the data link envelope.

Note: the use of a buffer size of 262 will result in drastic loss of performance. The fact that 262 is the default used by (old) servers that do not recognize the Data Link Buffer Size field does not mean that any server may choose to ignore this information. Instead, servers are expected to use the value supplied by the client. Furthermore, all clients should use the largest buffer size allowed for the data link in question whenever possible.

DATA LINK RELATED : = Information specific to the particular DATA LINK. Not present if not applicable. Length and encoding vary according to the specific definition of these fields for a given DATA LINK value.

PAD (1–3) : B = Possible padding. Conforming implementation shall not pad the System ID message. For compatibility with certain existing implementations, stations receiving System ID messages shall accept up to 3 bytes of padding with 0 at the end of the message.

5.5.4 Request Counters

The Request Counters message consists of:

CODE	RECEIPT

Figure 25: Request Counters message format

Where:

CODE (1) : B = The number 9.

RECEIPT (2) : B = A receipt number to identify the request.

5.5.5 Counters (CSMA/CD only)

The Counters message in the case of the CSMA/CD data link consists of:

CODE	RECEIPT	COUNTER BLOCK

Figure 26: Counters message format for CSMA/CD link

Where:

CODE (1) : B = The number 11.

RECEIPT (2) : B = A receipt number to identify the request.

COUNTER BLOCK (*) : = A block of counters as defined for the CSMA/CD data link (see Appendix B.4).

5.5.6 Counters (other than CSMA/CD)

The Counters message for data links other than CSMA/CD consists of:

CODE	DATA LINK	RECEIPT	COUNTER BLOCK

Figure 27: Counters message format for links other than CSMA/CD

Where:

CODE (1) : B = The number 21.

DATA LINK (1) : B = The type of data link protocol on the link being used. Values are in Appendix A.2.

RECEIPT (2) : B = A receipt number to identify the request.

COUNTER BLOCK (*) : = A block of counters as defined for the particular data link (see Appendix B).

5.5.7 Reserve Console

The Reserve Console message consists of:

CODE	VERIFICATION

Figure 28: Reserve Console message format

Where:

CODE (1) : B = The number 13.

VERIFICATION (8) : B = A verification code that must match before the receiving system can honor the request.

5.5.8 Console Command and Poll

This message is issued by the Console Requester in the command system and is received by the Console Server in the target system. The Console Command and Poll message consists of:

CODE	CONTROL FLAGS	COMMAND DATA

Figure 29: Console Command and Poll message format

Where:

CODE (1) : B = The number 17.

CONTROL FLAGS (1) : BM = The control flags indicate the state of the console carrier message streams. They insure that messages are not lost.

Bit	Function
0	Message Number — indicates the current message number. This is a one bit sequence number of the current Console Requester command message.
1	Command Break Flag — indicates if the (possibly null) command data is to be preceded by a break condition in the serial byte stream. This may take on the value of zero, meaning no break, or one, meaning there is a break.

COMMAND DATA (*) : = This is a (possibly null) sequence of bytes to be provided as input to the receiving system's higher level user of the Console Server.

5.5.9 Console Response and Acknowledge

This message is issued by the Console Server in the target system in response to the receipt of a Console Command and Poll message from the Console Requester in the command system. The Console Response and Acknowledge message consists of:

CODE	CONTROL FLAGS	RESPONSE DATA

Figure 30: Console Response message format

Where:

CODE (1) : B = The number 19.

CONTROL FLAGS (1) : BM = The control flags indicate the state of the console carrier message streams. They insure that messages are not lost.

Bit	Function

0 Message Number — indicates the current message number. This is a one bit sequence number of the current command message being acknowledged.

1 Command Data Lost Flag — indicates if some console command data was lost. This may take on the value of zero, meaning acceptance of the command data, or one, meaning that some command data was lost.

Note that the requester should not resend the command data, since part of the data may have been accepted. In that case, resending will produce unpredictable results. Instead, the error should simply be reported to the user.

2 Response Data Lost Flag — indicates if remote console response data was lost due to data overrun. This may take on the value of zero, meaning no detection of lost data, or one, meaning there was lost data.

RESPONSE DATA (*) : = This is a (possibly null) sequence of bytes to be provided as input to the receiving system's higher level user of the Console Requester.

5.5.10 Release Console

The Release Console message consists of:

CODE

Figure 31: Release Console message format

Where:

CODE (1) : B = The number 15.

Appendix A

Predefined Values

This appendix contains the predefined values for various maintenance operation parameters. These values are referenced in the interfaces and in the message definitions. Each parameter has a description to be used in the interface calls and an actual value to be used in protocol messages.

The values shown here reflect the state of the MOP Registry as of the publication date of this specification. New values are defined on an as needed basis. Any values not shown are reserved for use by Digital.

A.1 Communication Devices

Value	Name	Device
1	UNA	DEUNA UNIBUS CSMA/CD communication link
2	DU	DU11-DA synchronous line interface
3	CNA	DECNA Professional CSMA/CD communication link
4	DL	DL11-C, -E or -WA asynchronous line interface
5	QNA	DEQNA Q-bus CSMA/CD communication link
7	CI	Computer Interconnect interface
8	DA	DA11-B or -AL UNIBUS link
9	PCL	PCL11-B UNIBUS multiple CPU link
10	DUP	DUP11-DA synchronous line interface
11	LUA	DELUA UNIBUS CSMA/CD communication link
12	DMC	DMC11-DA/AR, -FA/AR, -MA/AL or -MD/AL synchronous link
13	LNA	MicroServer Lance CSMA/CD communication link
14	DN	DN11-BA or -AA automatic calling unit
16	DLV	DLV11-E, -F, -J, MXV11-A or -B asynchronous line interface
17	LCS	LANCE/DECserver100 CSMA/CD communication link
18	DMP	DMP11 UNIBUS multipoint synchronous link
20	DTE	DTE20 PDP-11 to KL10 interface
21	DBT	DEBET CSMA/CD communication link
22	DV	DV11-AA/BA UNIBUS synchronous line multiplexer
23	BNA	DEBNT BI CSMA/CD communication link
24	DZ	DZ11-A, -B, -C, or -D UNIBUS asynchronous line multiplexer
25	LPC	VAXmate (LANCE) CSMA/CD communication link
26	DSV	DSV11 Q-bus synchronous link
27	CEC	3Com 3C501, IBM-PC CSMA/CD adapter
28	KDP	KMC11/DUP11-DA synchronous line multiplexer
29	IEC	Micom/Interlan 5010, IBM-PC CSMA/CD adapter

30	KDZ	KMC11/DZ11-A, -B, -C, or -D asynchronous line multiplexer
31	LQA	DELQA CSMA/CD communication link, alternate assignment. See note below.
33	DS2	LANCE/DECserver 200 CSMA/CD communication link
34	DMV	DMV11 Q-bus synchronous link
35	DS5	DECserver 500 CSMA/CD communication link
36	DPV	DPV11 Q-bus synchronous line interface
37	LQA	DELQA CSMA/CD communication link. See note below
38	DMF	DMF-32 UNIBUS synchronous line unit
39	SVA	DESVA Microvax-2000, 3100, 3300 CSMA/CD communication link
40	DMR	DMR11-AA, -AB, -AC, or -AE UNIBUS interprocessor link
41	MUX	MUXServer 100 CSMA/CD communication link
42	KMY	KMS11-PX UNIBUS synchronous line interface with X.25 level 2 microcode
43	DEP	DEPCA PCSG/IBM-PC CSMA/CD communication link
44	KMX	KMS11-BD/BE UNIBUS synchronous line interface with X.25 level 2 microcode
45	LTM	LTM (911) Ethernet monitor
46	DMB	DMB-32 BI synchronous line multiplexer
47	DES	DESNC Ethernet Encryption Module
48	KCP	KCP Professional synchronous/asynchronous comm port
49	MX3	MUXServer 300 CSMA/CD communication link
50	SYN	MicroServer Synchronous line interface
52	DSB	DSB32 BI Synchronous Line Interface
53	BAM	DEBAM LANBridge-200 Data Link
54	DST	DST-32 TEAMmate Synchronous Line Interface (DEC423)
58	3C2	3COM Etherlink II (part number 3C503)
59	3CM	3COM Etherlink/MC (part number 3C523)
60	DS3	DECServer 300 CSMA/CD communication link
61	MF2	MicroVAX 3300 CSMA/CD communication link
63	VIT	Vitalink TransLAN III/IV (NP3A) Bridge
64	VT5	Vitalink TransLAN 350 (NPC25) Bridge, TransPATH 350 BRouter
65	BNI	DEBNI BI CSMA/CD communication link
66	MNA	DEMNA XMI CSMA/CD communication link
67	PMX	DECstation-3100 CSMA/CD communication link
72	DP2	DECserver-250 (parallel printer server) CSMA/CD communication link
73	ISA	Pele SGEC-based CSMA/CD communication link
74	DIV	DIV-32 Q-bus ISDN (2B+D) adapter
75	QTA	DEQTA (DELQA-YM) CSMA/CD comm link
76	B15	LANbridge-150 CSMA/CD comm link
86	DR2	DECrouter-250 CSMA/CD comm link
87	SCC	DECrouter-250 DUSCC serial comm link (DDCMP or HDLC)
90	FBN	DECbridge-5xx CSMA/CD comm link
91	FEB	DECbridge-5xx, -6xx FDDI comm link
92	FCN	DECconcentrator-500 wiring concentrator FDDI comm link
93	MFA	DEMFA XMI — FDDI comm link
94	MXE	MIPS workstation family CSMA/CD comm links (3MAX/KN02 system board; PMAD option board)
101	DSF	DSF-32 2 line synchronous comm link for Cirrus
104	KFE	VAXft-3000 KFE52 CSMA/CD comm link
105	RT3	rtVAX-300 SGEC-based CSMA/CD comm link
112	L20	LPS20 print server CSMA/CD comm link
114	DWT	VT-1000 DECwindows terminal
115	WGB	DEWGB Work Group Bridge CSMA/CD comm link
118	MNE	3MIN (KN02-BA) integral CSMA/CD comm link
119	FZA	DEFZA TurboChannel FDDI comm link
120	90L	DS90L terminal server CSMA/CD comm link
135	TRN	DEQRA token ring (802.5) comm link

139	ISE	Network Integration Server 600 (Hastings) CSMA/CD line card
140	IST	Network Integration Server 600 (Hastings) T1 sync line card
141	ISH	Network Integration Server 600 (Hastings) 64 kb HDLC line card
142	ISF	Network Integration Server 600 (Hastings) FDDI line card
145	FB3	DECbridge-6xx CSMA/CD (3 port) comm link

Note:

Note on the assignment for the DELQA: the primary assignment is 37. Device code 31 is used in early DELQA devices when used on PDP-11 systems, in the "Request Program" message only. Later revisions of the DELQA use code 37 throughout. Software shall treat codes 31 and 37 as equivalent.

Note on the assignment for MicroVAX 3300 integral CSMA/CD comm link: there is a specific assignment for this device (61, mnemonic "MF2"). This code is used by the boot ROMs. Operating system drivers treat it as equivalent to the MicroVAX 2000 integral CSMA/CD comm link and use the code assigned to that device (39, "SVA").

A.2 Data Links

The data link type values are:

Value	**Meaning**
1	CSMA–CD
2	DDCMP
3	LAPB (frame level of X.25)
4	HDLC
5	FDDI
6	Token-passing Ring (IEEE 802.5)
7–10	*Reserved*
11	Token-passing Bus (IEEE 802.4)
12	Z–LAN 4000: Zenith 4 Megabit/second broadband CSMA/CD LAN

Note that these codes are *not* the same as the "Circuit Type" codes used in the management interface, which are defined in Section 3.4. To simplify the mapping between these two encodings, new values for Circuit Type and Data Link Type will be assigned such that the following algorithm applies:

```
IF DataLinkType ≤ 10
THEN CircuitType := LookupTable[DataLinkType]
ELSE CircuitType := DataLinkType – 5
```

Appendix B

Data Link Specific Information

This appendix contains information necessary to relate specific data link types to the maintenance operations.

B.1 DDCMP

The Digital Data Communication Message Protocol (DDCMP) Data Link is a point-to-point link and allows exclusive maintenance operation in its maintenance mode. It does not require message padding.

B.2 LAPB

The LAPB Data Link is the frame level of X.25. It is a point-to-point link and allows exclusive maintenance operation for loopback only. It does not require message padding.

B.3 HDLC

The HDLC Data Link is a point to point link. It supports concurrent maintenance operation (MOP protocol messages are carried via the Unsequenced Information service of HDLC). It does not require message padding.

The HDLC protocol ID is 01–01; it is used for all MOP protocol messages (unlike the LAN case where multiple protocol IDs or protocol types are used).

B.4 CSMA/CD

The CSMA/CD Data Link is the Digital Equipment Corporation implementation of the intercompany Ethernet Data Link and the IEEE 802.3 Data Link. It allows concurrent maintenance operation and is a LAN (multi-access) data link. As such it has specific protocol types and multicast addresses that go with it. It requires message padding when Ethernet frame format is used, except for the Loop protocol.

Refer to the the LAN Node Product Architecture Specification and the DNA CSMA/CD Data Link Architectural Specification for specific functions and requirements.

The protocol types are:

Value	Protocol
90–00	Loopback
60–01	Dump/Load
60–02	Remote Console

The IEEE 802 SNAP SAP Protocol Identification are :

Value	Protocol ID
08–00–2B–90–00	Loopback
08–00–2B–60–01	Dump/Load
08–00–2B–60–02	Remote Console

The multicast addresses are:

Address	Group
CF–00–00–00–00–00	Loopback assistance
AB–00–00–01–00–00	Dump/Load assistance
AB–00–00–02–00–00	Remote Console

CSMA/CD data link counters can be read through the Remote Console. The counters are defined in the DNA CSMA/CD Data Link specification. The counters are a fixed format block with each value as indicated below. All fields must be included in the order specified. If a counter is not implemented, the corresponding field is still included, with a value of zero. Note that MOP V3 omitted the last field (Collision detect check failure), and used counters of a different width. In V4, all counters are 64 bits (8 bytes) long. The “time since counter creation” value is a Binary Relative Time. Systems that request Data Link counters using MOP can use the length of the received counters block to distinguish the two versions.

Note:

Unlike the MOP V3 counter block, the “Time since counter creation” field is *not* in seconds. It is encoded as a Binary Relative Time according to the encoding rules specified in the Time Representation specification.

Length	Counter Value
16	Time since counter creation
8	Bytes received
8	Bytes sent
8	Frames received
8	Frames sent
8	Multicast bytes received
8	Multicast frames received
8	Frames sent, initially deferred
8	Frames sent, single collision
8	Frames sent, multiple collisions
8	Send failure — Excessive collisions
8	Send failure — Carrier check failed
8	Send failure — Short circuit
8	Send failure — Open circuit
8	Send failure — Frame too long
8	Send failure — Remote failure to defer
8	Receive failure — Block check error
8	Receive failure — Framing error
8	Receive failure — Frame too long
8	Unrecognized frame destination
8	Data overrun
8	System buffer unavailable
8	User buffer unavailable
8	Collision detect check failure

B.5 FDDI

The FDDI Data Link is the Digital Equipment Corporation implementation of the ANSI standard FDDI (Fiber Distributed Data Interface) Data Link. It allows concurrent maintenance operation and is a LAN (multi-access) data link. It supports the IEEE 802.2 LLC service.

Refer to the the LAN Node Product Architecture Specification and the DNA FDDI Data Link Architectural Specification for specific functions and requirements.

The FDDI Data Link uses the same SNAP Protocol Identification and multicast address assignments as the CSMA/CD Data Link. In addition, it also uses the same Protocol Type assignments for use with the Mapped Ethernet packet service of the FDDI Data Link architecture. (This service sends IEEE 802.2 SNAP format packets with Protocol ID values recognized by LAN Bridges. When such packets are forwarded onto CSMA/CD links, they are converted into Ethernet format. MOP uses this service to communicate with MOP V3 implementations on Ethernet links.)

FDDI Data Link counters can be read through the Remote Console. The counters are defined in the DNA FDDI Data Link specification. The counters are a fixed format block with each value as indicated below. All fields must be included in the order specified. If a counter is not implemented, the corresponding field is still included, with a value of zero. This counter block is returned in the “Counters (other than CSMA/CD)” message (Section 5.5.6). Since that message did not exist in MOP V3, MOP will ignore requests for counters in V3 format.

Length	Counter Value
16	Time since counter creation
8	Octets received
8	Octets sent
8	Frames received
8	Frames sent
8	Multicast octets received
8	Multicast octets sent
8	Multicast frames received
8	Multicast frames sent
8	Transmit underruns
8	Transmit failures
8	Block check errors
8	Frame status errors
8	Frame alignment errors
8	Frames too long
8	Unrecognized individual frame destinations
8	Unrecognized multicast frame destinations
8	Receive data overruns
8	Link buffers unavailable
8	User port buffers unavailable
8	Frame count
8	Error count
8	Lost count
8	Ring initializations initiated
8	Ring initializations received
8	Duplicate address test failures
8	Duplicate tokens detected
8	Ring purge errors
8	Bridge strip errors
8	Traces initiated
8	Link selftest failures

Appendix C

Implementation Specific Dump/Load Characteristics

This appendix documents characteristics of PDP–11 dump/load programs existing as of the date of this specification.

C.1 Secondary Loader

The secondary loader is sent as a single Memory Load with Transfer Address message as normally required. In addition to this requirement, it must be loaded and started at location 6. Current secondary loaders are between 400 and 600 bytes in length, depending upon the device type used. They use the stack address set up by the primary loader. For current loaders this will be between 17400(octal) and 17776(octal). The secondary loader assigns its buffer space below the stack. The secondary loader accepts Memory Load with and without Transfer Address messages. It is, therefore, capable of doing multi-block loads into absolute addresses without memory management. It requests a tertiary loader to be loaded.

The DMP-11 and DMV-11 do not set up the stack pointer or R1 as described. For those devices, R1 contains the device unit number.

C.2 Tertiary Loader

The tertiary loader is loaded by the secondary in a multi-block load starting at location 10000(octal). It will run with memory management on if it exists on the system. The tertiary loader moves itself to the top of physical memory and assigns its stack and buffer space just below itself. It is, therefore, capable of multi-block loads from location 0 up to its buffer address, usually the last 1–2K words of physical memory. It requests the operating system to be loaded. The current tertiary loaders do not specify any specific operating system. The choice of system to send is established by prior agreement or by command at the host system.

Appendix D

LAN Loop Test Examples

The following examples address the application of the Loop Test functions. They are intended as examples of how a higher level process can use the facilities. They are intended neither as a specification for how they must be used nor as an exhaustive test script.

In the examples, no account is taken of the fact that the Loop Test functions make only one attempt to transmit a message. To increase the reliability of the tests, each interface function that fails due to a communication error should be retried enough times to lessen the probability that an intermittent error occurred.

D.1 Local Control Test Example

In this case, a node finds itself unable to communicate with some other node that it believes is on the network and operational. The following test script can be used by the node to check out the problem.

First, LoopDirect is invoked with remoteAddress set at the correct address of the specific remote node, the suspect node. This results in a simplest, two hop frame consisting of :

1. a Forward Data message which specifies the testing node's address as the forwarding address, containing

2. a Reply message.

This frame is transmitted to the suspect node. If this test succeeds, the communication is possible and the problem may have been either intermittent, the remote node is down, or there is a problem with message length or data pattern. Different message lengths and/or data patterns could then be tried.

If the above test, LoopDirect, results in a return indicating failure, next invoke LoopAssisted, using some other node as assistantAddress. This will construct a loop test datagram as follows:

1. a Forward Data message which specifies the suspect node's address as the forwarding address, containing:

2. a Forward Data message which specifies the assistant's node address as the forwarding address, containing:

3. a Forward Data message which specifies the testing node's address as the forwarding address, containing:

4. a Reply message.

This frame is transmitted to the assistant node. This four-part datagram obtains the full assistance of the loopback assistant.

If no assistant address is known, invoke LoopDirect with no remoteAddress parameter to send a simple, two hop loop test datagram to the Loopback Assistance Multicast Address. Use the source address of any responding node as an assistant address. If no communication with the Loopback Multicast Assistance group is possible, then the last resort is to obtain a list of nodes on the network. The list can be obtained by either listening to the Remote Console Multicast Address for the Remote Console System ID messages or obtaining such information from the network manager. From the list of nodes, select a node for loopback assistant by repeating the above procedure. If this fails then repeat the process with other nodes on the list. If this fails with all the nodes on the list, then either no one else is turned on or the local node is broken.

If some loopback assistant node can communicate with the remote node but the local node cannot, the local node can then test for the direction in which communication does not work. The LoopAssisted function, using the assistant node that responded previously as the assistant, can be used to detect either transmit or receive problems. This is accomplished using various assistance level, transmit or receive or full, supported by the loopAssisted function. By repeating the above test with different remote nodes, it can be determined whether the local node or the remote node is at fault, thus isolating the problem to a particular transmitter or receiver.

When a node finds itself unable to communicate, it can report this in whatever high level way is available. An operator or control center can then respond by attempting to isolate and repair the problem.

D.2 Remote Control Test Example

When a control center receives a report that a node cannot communicate properly, it can use the Loop Test functions to investigate the problem. It can first diagnose its own ability to communicate with the node. If this communication appears to work, the control center can similarly check its ability to communicate with the remote node that the reporting node could not reach.

If the control center can communicate with both nodes, it can then use LoopAssisted, full assistance, with one of the nodes as assistant and the other as remote to see if they can communicate. Similarly it can use transmit and receive assistance to determine which direction is a problem.

Similar checks using other nodes can be used to isolate the problem to a particular transmitter or receiver.

Appendix E

Load File Formats

E.1 CMIP Script File Format

The MOP protocol specifies a way for downline loading initialization management directives in the form of a sequence of CMIP directive messages, called a CMIP Script. In order to allow initialization scripts to be manipulated on various nodes that may be different implementations than the load hosts, it is also necessary to specify the file format of initialization scripts. The following specification applies only to files used as MOP initialization scripts. Whether they can also be used in other ways, for example as scripts accepted by management directors, is beyond the scope of this architecture specification. The CMIP Script file has the structure of an RMS sequential file with variable-length records. The records of the file correspond one-to-one with the CMIP Script messages sent in the protocol. In particular, the first record contains the CMIP version number as described in the DNA CMIP architecture specification, and subsequent records contain the directive messages, one per record. The RMS FDL description of the record format is as follows:

FILE		
	...	
	ORGANIZATION	sequential
RECORD		
	BLOCK_SPAN	yes
	CARRIAGE_CONTROL	none
	FORMAT	variable
	SIZE	0

In the file, each record is preceded by a 2 byte field specifying the byte count of the record. If the byte count is odd, a pad byte (null) follows the record, thereby aligning the byte count fields on even boundaries. Block boundaries are given no special processing. End of file is indicated by a 2 byte field of hex FFFF instead of the byte count, and the remainder of the block is null-filled.

These files can be processed by RMS or directly by simple application code, and can be transferred using the DAP protocol (using Block mode if necessary).

Appendix F

LAN Frame Formats

Note:

This appendix is not part of the specification and is not under the control of the DNA review process. It is included here for information only.

The following diagrams illustrate examples of LAN frame formats for several of the LAN data links. Refer to the appropriate Data Link architecture specification for a detailed description of the messages and fields.

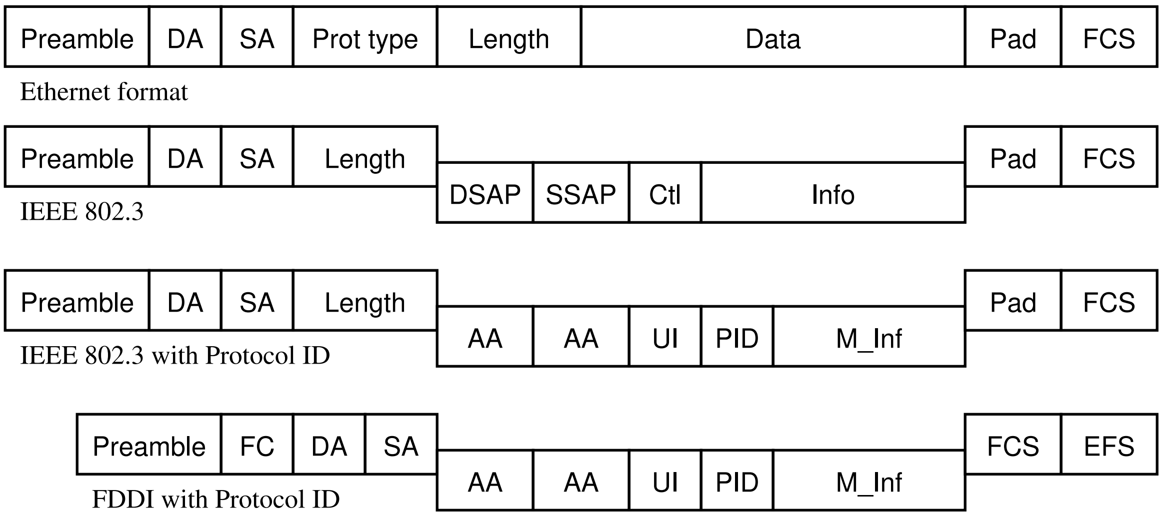

Figure 32: NI Frame Formats

Preamble: A sequence of encoded bits which the Physical Layer transmits before each frame to allow synchronization of clocks and other Physical Layer circuitry at the receivers.

FC: Frame Control field defines the type of the frame and certain MAC and management specific information frame functions.

DA: the Destination Data Link Address

SA: the Source Data Link Address

Length: the number of octets of user data (optional for Ethernet format). It is applicable for CSMA/CD LAN only.

Prot Type: the Protocol Type for Ethernet frames

PID: the Protocol Identifier for 802.2 PDUs addressed to the SNAP SAP

LLC: IEEE 802.2 Logical Link Control sublayer

PDU: Protocol Data Unit

SNAP: the 802.1 Sub-Network Access Protocol

DSAP: the LLC Destination Service Access Point Address

SSAP: the LLC Source Service Access Point Address

Ctl: the LLC PDU Control field. For connectionless service it is one of:

UI: the Unnumbered Information type LLC PDU Control field

XID: the Exchange Identification type LLC PDU Control field

TEST: the Test type LLC PDU Control field

Data: the MAC user data

Info: the LLC SAP user data

M__Inf: the 802 SNAP Protocol Identifier user data

PAD: the padding octets (optional for Ethernet format)

FCS: the Frame Check Sequence

EFS: End-of-Frame Sequence. The EFS typically contains the Ending Delimiter (ED), to mark the end of the frame, and the Frame Status.

Appendix G

Compatibility with Previous Versions

This appendix discusses what implementations of V4.0 of the MOP architecture need to do in order to communicate with implementations of earlier versions of the MOP architecture, specifically V3.0.0 and V3.1.0. Refer to the MOP V3 architecture specification for more details about message formats and algorithms for that protocol version.

G.1 Support for the previous version of MOP

In the Load Client and Dump Client components, support for MOP V3 is a product implementation choice. General purpose products should include V3 support in the Load/Dump Clients to ensure flexibility in the choice of Load/Dump Servers. Bounded (fixed function) products for which it is acceptable to limit the choice of Load/Dump Servers to only those that support MOP V4 may omit the V3 support from the Load/Dump Clients.

For all other functions, support for MOP V3 is required.

Note, however, that certain functions of MOP V4 are not defined in MOP V3. These functions are of course exempt from the above requirement. The functions specific to MOP V4 are:

- HDLC support
- Response to the Read Counters request message on FDDI data links
- CMIP Script down line load

G.2 Ethernet and 802 frame formats

On a CSMA/CD LAN, the data link allows the use of both 802 format and Ethernet format frames. MOP uses 802 format frames (using SNAP format LLC frames) in V4. MOP V3 used Ethernet format packets.

Other LANs, such as FDDI, do not have the direct equivalent of “Ethernet” format; they only support IEEE 802.2 LLC. To support communication with nodes that use Ethernet message formats across LAN Bridges, there exists a packet format mapping that maps Ethernet packets (with 2 byte Protocol Types) into IEEE 802.2 packets (using SNAP, with 5 byte Protocol IDs). The data link layer architecture for these LANs describes this mapping in detail. In the service interface, the service provided is essentially identical to that of the CSMA/CD data link with its support of Ethernet format. Consequently, the rules about the use of Ethernet format on LANs apply to all LANs; in the case of LANs other than CSMA/CD the encapsulated Ethernet packet service of the data link is used for this. In particular, where nodes are required to support MOP V3, this requirement applies to non-CSMA/CD LANs as well as to CSMA/CD, unless specifically stated otherwise.

G.2.1 Response to received requests

MOP V4 implementations listen for requests both in 802 (V4 format) and in Ethernet (V3 format) frames. The response to the request is in the same frame format as the request. If the request was in an Ethernet format frame, it is interpreted and responded to according to the algorithms specified in the MOP V3 architecture specification (except as noted below).

G.2.2 Initiation of requests

MOP V4 implementations send requests both in 802 (V4 format) and in Ethernet (V3 format) frames. In the case of the Ethernet format request, the message is formatted according to the V3 architecture rules. The 802 format request is sent out first; if no response is received in the timeout period, the Ethernet format request is then sent. If that times out as well, the 802 format request is sent again. Each iteration of 802 format, then Ethernet format frame counts as one retry. As an optimization, an implementation may send both formats at the same time. For some protocol exchanges (e.g., Loop) care must be taken not to confuse the replies; the receipt number in the message can be used to do this.

The algorithm for the Boot message is slightly different, since there is no response to that message. The Boot message is simply sent in both formats, with the 802 format frame sent first.

Note that requests that are not expressible in Ethernet format, such as a Request Program message specifying a request for a CMIP Script, are sent in 802 format only.

In MOP V3, the Verification field of the boot message was defined as being either 4 or 8 bytes long. V4 implementations may send V3 format boot messages with 8 byte verification fields independent of the value in the field. However, MOP implementations receiving a boot message in V3 format must be prepared for a 4 byte or an 8 byte verification field. If a 4 byte verification field is received, the omitted 4 bytes are treated as zero.

The Verification attribute in the Client database is an Octet String. In Phase IV, it was defined as a "Hex Integer", which is not a defined data type in CMIP. Since strings are written with the lowest numbered byte on the left, and integers with the lowest numbered byte on the right, this means that the same verification string is represented differently in the two management user interfaces. For example, the verification string written in this version of MOP as %x0102560415A69708 would appear in the Phase IV management interface as 0897A61504560201, or (omitting leading zeroes) as 897A61504560201. This difference does not affect the protocol, but it does affect documentation. If a Boot target allows setting of the verification string in Phase IV format, but the Load Server implements MOP V4, the verification strings supplied in the user interface have to be bytewise reversed.

G.3 Version handling on point to point data links

The Request Program and Request Dump Service contain a "Format Version" field which indicates the version of the MOP architecture to which the requesting system conforms. The rules for the handling of this version number are as follows:

- If the version number is higher than the highest version known to the Dump/Load server, ignore the request.
- Otherwise, process the request according to the rules for the specified version.

Note:

Note that the usual third case in version number handling does not occur here, since the only version number other than the one currently used (4) is 1, which is used by all versions of MOP preceding V4.0.0.

G.4 Request Program message

The Program Type value of CMIP Script file can only be specified in a V4 format Request Program message.

The rules given below for interpretation of the Data Link Buffer Size field in the System ID message (Appendix G.6) apply also to the interpretation of that field in the V3 format Request Program message.

G.5 Parameter Load with Transfer Address message

This message has changed substantially between MOP V3 and V4. Parameter type codes 1 through 5 were defined in previous versions of MOP; parameter type code 6 is used only in V4. The values of parameters 1 through 4 are obtained from the Client database entry used for the Load operation. The characteristics used for each parameter are as follows:

Parameter	Attribute
1	Phase IV Client Name
2	Phase IV Client Address
3	Phase IV Host Name
4	Phase IV Host Address

In the V3 format message, Host System Time is passed as parameter code 5, in the format specified in the V3 architecture specification. In the V4 format message, it is in addition passed as parameter code 6, as a Binary Absolute Time.

The MOP V3 architecture defines parameters 2 and 4 as 6-byte values, though it leaves open the issue of whether shorter values are permitted. All known implementations in fact send only two bytes, exactly as in MOP V2.1. Therefore, V4 implementations must likewise send two bytes. The parameter value sent is the value of the Phase IV Client Address or Phase IV Host Address attributes, respectively. The low order byte of the attribute is sent first.

A MOP V4 load client being loaded by a V3 load server will receive a Parameter Load message in V3 format. It uses parameter code 5 to obtain the local time of the load server, but that does not provide all of the information of the V4 format Host System Time field. Alternatively, the load client can implement the Time Service client, ignore the Host System Time field altogether, and instead use information obtained from the Time Service. (This is the preferable solution if the Time Service is available.)

G.6 System ID message

When sending a System ID message in V3 format, V4 nodes must omit those fields which are defined only in MOP V4 (Node ID, System Time, Node Name part 1 and 2, Station ID). System Time may be sent instead (if desired) using type code 8, encoded according to the V3 architecture specification. The Maintenance Version field is always sent with the value 4.0.0, indicating a Version 4 MOP implementation, whether the System ID message is sent in Ethernet format (to a V3 node) or in 802 format (to a V4 node).

In both the V3 and the V4 format messages, the value for Data Link Buffer Size specifies only the length of the MOP data (including MOP protocol header) but not any part of the Ethernet or IEEE 802.2 LLC header. Thus the largest possible value for V3 format messages on Ethernet is 1498; for 802.2 format messages on Ethernet the limit is 1492.

Note:

For compatibility with certain existing implementations, any value greater than 1498 for Data Link Buffer Size when received in a V3 format System ID, Request Program, or Request Dump Service message shall be interpreted as 1498. Furthermore, the value 1030 shall be interpreted as 1010.

G.7 Counters message

The MOP V4 CSMA/CD Counters message contains the list of counters specified in Appendix B.4. The V3 Counters message is substantially different: the counters are shorter, several counters are combined into a single counter with a "reasons" bitmap, Collision Detect Check Failure is omitted, and the time since counter initialization is encoded differently. Apart from the frame format, these two cases can also be distinguished by the length of the received counter block.

For data links other than CSMA/CD (currently only FDDI) the counter block defined in MOP V3, or its modified form for V4 described above, is not suitable, since the set of counters are different for the different data links. Such data links therefore use a different message to respond to the Request Counters message, with message code 21. That message is new in MOP V4. Since it has no MOP V3 equivalent, non-CSMA/CD data links do not respond to the Request Counters message from MOP V3 nodes. The System ID message sent on these data links reflects this rule: the V4 format System ID (sent in 802.2 format) will show Data Link Counters among the supported functions, but the V3 format System ID (sent in Ethernet format) does not.

Appendix H

Glossary

Broadcast: As applied to data links, broadcast capability refers to the ability of any one node (link) to address all other nodes (links) simultaneously.

Broadcast Address: This is a group address which is a distinguished, predefined multicast address that always denotes the set of all nodes on a given Extended LAN. The Broadcast Address is predefined to be the address with 48 bit of all 1's.

CMIP Script: An description of the information required to initialize a node, encoded in the form of a sequence of network management directives in CMIP form. (See also "Management Image".)

Datagram: An atomic unit of information exchanged by local area networks.

Flow Control: A set of rules applied to processes, beetween the nodes , which prevents a transmitting process from sending data to a receiving process that is not prepared to buffer the transmitted data.

Extended LAN: Diverse Local Area Networks interconnected, by the connecting network component called the Bridge, into an extended, logically integrated network.

Disjoint Extended LAN: A disjoint Extended LAN is defined to be separate universes where under no condition shall the separate universes ever be merged together.

Group Address: The first bit of the 48 bit address contains a 1, the first bit transmitted. This is a multi-destination address, associated with one or more stations on the Extended LAN. There are two kinds of Group address: Multicast address and Broadcast address.

Hardware Address: This is the default hardware 48 bit LAN Address. This is the universally unique hardware address which may or may not be used as the physical (link) address.

Individual Address: The first bit of the 48 bit address contains a 0, the first bit transmitted. All Physical (Link) addresses and Hardware addresses must be individual address.

LLC: Logical Link Control. The top sublayer of the data link that provides the mechanism to support multiplexing and demultiplexing of datagrams over the Medium Access Control sublayer defined by IEEE 802.2 spec.

MAC: Medium Access Control. The bottom sublayer of the data link that provides frame encapsulation/decapsulation, error checking and access control algorithms for controlling the sharing of the physical layer by the attached stations (nodes).

Management image: A load image containing initialization data for a node, encoded in a product-specific manner. (See also “CMIP Script”.)

Multicast: As applied to data links, multicast capability refers to the ability of any one node (link) to address a subset of all other nodes (links), within the logical group, with a single datagram.

Multicast Address: This is a Group Address as defined above. This is an address associated by higher-level convention with a group of logically related nodes.

Physical (Link) Address: This is the 48 bit address that identifies the link’s attachment point in the Extended LAN. This is the address recognized as specific destination in received frames and sent as source address in the transmitted frames.

SAP: Service Access Point. The LLC header contains the address fields (DSAP, SSAP), which is the ordered pair of service access point addresses at the beginning of an LLC frame which identifies the LLC’s designated to receive the frame and the LLC sending the frame. Each of the DSAP and SSAP is one octet in length.

SNAP SAP: It is the Sub-Network Access Protocol (SNAP).

Appendix I

Load/Dump Service Implementation Notes

This appendix contains a number of notes and suggestions for implementers of Load/Dump Clients and Servers.

I.1 Downline load and upline dump over multi-mode point to point lines

When load or dump service is to be provided via point to point lines (DDCMP or HDLC) it is sometimes necessary to provide the data link layer with operating parameters such as bit rate, framing mode, etc. For servers, this is in general not an issue because these parameters are set as part of the normal initialization procedure for the node.

Clients may have difficulty providing these data link parameters, since clients run in primitive state. It is generally not convenient or even desirable to have to set up line parameters prior to downline load, for reasons such as:

- Autoconfiguration: there should be no need for management setup as part of installation ("Plug and Play").
- Lack of a local console: load clients often do not have any local management interface.

For these reasons, a procedure similar to the "autobaud" mechanism used in many timesharing systems would be desirable. Ideally, the autoconfiguration process should be able to deal with all of the following parameters:

- Data Link protocol: HDLC vs. DDCMP, for lines that are capable of supporting both.
- Byte framing mode: synchronous or asynchronous.
- Line speed, for asynchronous framing mode.

Such an autoconfiguration is feasible, provided that certain requirements are met at the server end. The server must have its parameters set up via management (i.e., it must not attempt to run the autoconfiguration procedure at the same time as the client), its data link must be On, and the data link must then run its data link initialization procedure indefinitely, rather than giving up after a limited number of retries.

Given that the above requirements are met, a simple client algorithm that meets the above goals is as follows:

1. Set the framing mode for the data link to Synchronous. Generally this also requires setting the data link for externally supplied bit clocking (e.g., from the modem). Set the data link protocol to HDLC (bit stuffing mode) and attempt to receive some frames.

   If the server is indeed using HDLC, valid frames will be received within a few seconds, containing HDLC data link startup messages (SABM, SABME, or XID).

2. Set the data link protocol to DDCMP (byte count character oriented) and attempt to receive some frames. If the server is running DDCMP in synchronous mode, valid frames will be received within a few seconds, containing DDCMP data link startup messages (ISTRT). If the server is trying to perform MOP protocol operations on the data link, MOP mode DDCMP messages will be received instead.

3. Set the framing mode for the data link to Asynchronous. For each of the data link speeds to be supported, set the data link to that speed and listen for a few seconds for messages as above. Examine the messages to see if they are valid HDLC or DDCMP initialization messages. If none are heard, proceed to the next speed.

This algorithm can be repeated until it completes.

I.2 Downline load and upline dump service on diskless servers

When load or dump clients have multiple circuits, it is generally required that any of the lines can be used for downline load or upline dump. This may appear difficult when some of these lines are point to point lines connected to neighbors that have no disk storage, such as routers. This section presents a way to provide load/dump service in such cases.

MOP does *not* require that the load server have local file storage, such as disks. It may use any available means to obtain the files. In particular, remote file access protocols, such as DAP, may be used. DAP is well suited to this job, since the subset required by load servers is quite small. Only sequential read access is needed, and block mode can be used. This means that the DAP access amounts to little more than establishment of the Transport connection to the file server and receiving of the data blocks over that connection. Similarly, for upline dump only sequential write access is needed, and again block mode is sufficient.

Diskless load servers still include a normal MOP Load Server module, with the usual management interface. The necessary attributes, in particular the Client database, can be set up in any convenient manner, for example with the initialization script. The client database would contain the necessary entries to describe the requests the server must deal with. One of these would be an entry for the neighbor on the point to point line. The various attributes in the client database that specify the load file names are set to point to the files on the file server that stores the load files. These attributes are Sequence of FileSpec to allow the network manager to specify multiple locations for the files. This provides for redundant file servers.

I.3 Downline load and upline dump in clients with multiple data links

The algorithms given in the Operation chapter (Chapter 4) are for a load or dump operation on a single data link circuit. If the client has multiple circuits, the load or dump should be able to use any of them. The client needs some way to select the circuit on which the operation can be completed. Once it has done this, the remainder of the load or dump operation is executed in the usual manner.

There are several ways the initial step of the load or dump can be done on a multi link client. The most efficient way is to send the first message (Request Program in the case of downline load, Request Dump Service in the case of upline dump) simultaneously on each of the circuits.

When a response is received on any of the circuits, that circuit is selected, and the rest of the operation proceeds as usual on that circuilt.

The advantage of this “parallel” approach is speed and the fact that it requires no additional state to be kept, but it does require dealing with multiple circuits simultaneously.

Alternatively, the client can attempt to initiate the operation on the circuits one at a time in “round robin” fashion. To do this, the client would send the first message of the operation on one of the circuits, performing a small number of retries (4 retries would be suitable). If no response is received, the same is then done for the next circuit, and so on through each of the available circuits.

The “round robin” approach is potentially slower, but it may be easier in some implementations. All circuits except the one being tried at any point in time are logically Off, so the client does not need to do any processing for the other circuits. (Note in particular that the requirements of the Node Product Architecture apply only to data links that are On — in this case, only one data link circuit is On at any time.)

Index