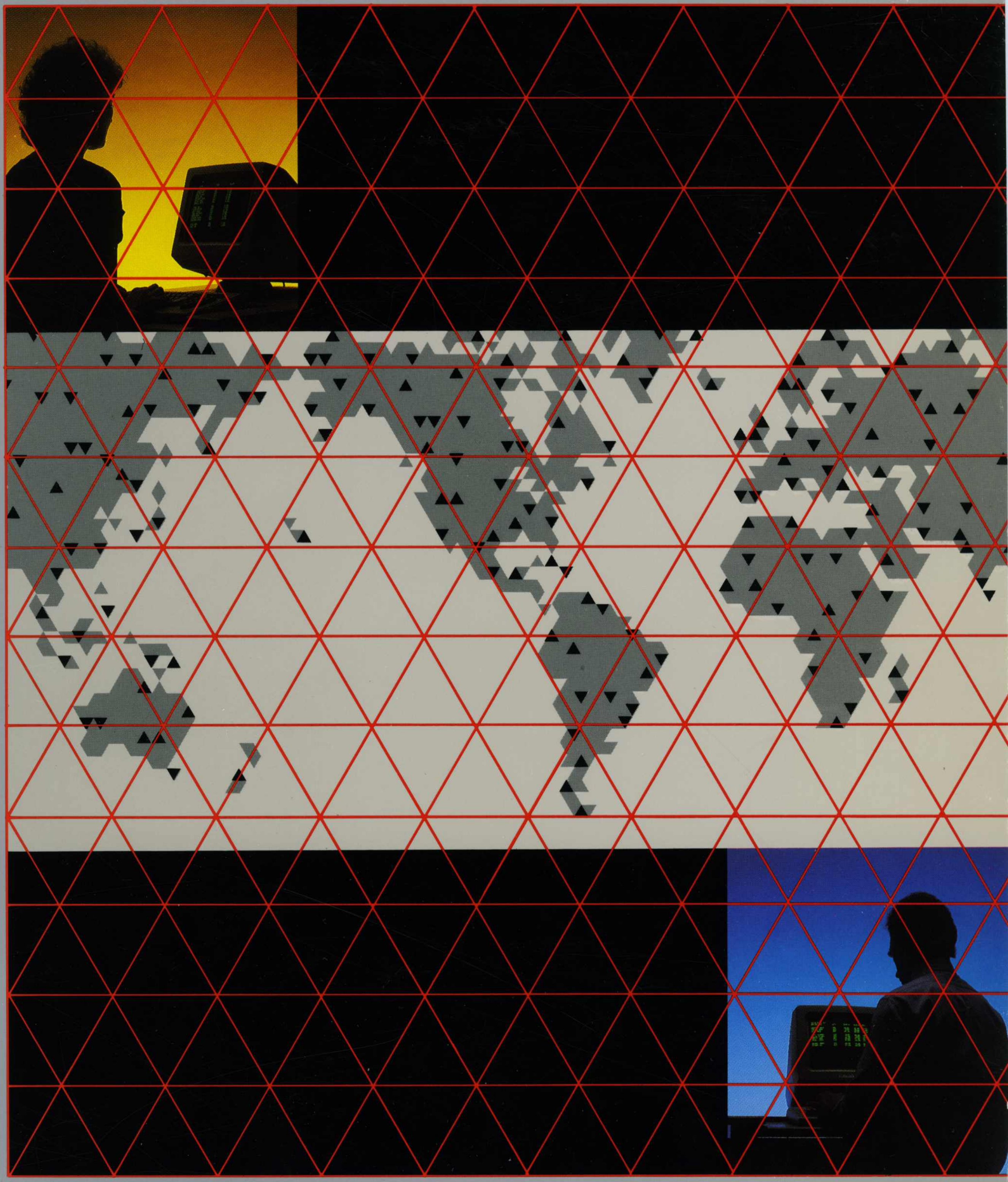
CDCNET Base System Software

Systems Programmer's Reference Manual, Volume 1



60462410

CONTROL DATA

CDCNET Base System Software

Systems Programmer's Reference Manual, Volume 1

Publication Number 60462410

Manual History

This manual is revision A, printed in September 1986. It documents CDCNET Software applicable to NOS and NOS/VE environments.

Contents

Figures

About This Manual

The CDCNET Systems Programmer's Reference Manual describes the CDC® Control Data Distributed Communications Network (CDCNET) software. The CDCNET software enables you to write new terminal interface programs (TIP) and gateway software to support terminals, networks and devices not currently supported by Control Data.

Audience

The Systems Programmer's Reference Manual is intended for anyone who will develop or integrate new software that is compatible with CDCNET. An understanding of the software described in this manual can be useful to programmers who intend to write new terminal interface programs or gateways for CDCNET. This includes writing of new applications, TIPs for terminals not already supported by CDCNET, and gateways to networks using protocols foreign to CDCNET. The reader is assumed to be familiar with CDCNET network operations.

Organization

This manual is one of a three-volume Systems Programmer's Reference Manual set that describes the CDCNET software. The following subsection contains brief descriptions of the other manuals in the set and a more detailed description of this manual.

Systems Programmer's Reference Manual Set

The three-volume manual set includes the following manuals.

- Volume 1: Base System Software
- Volume 2: Network Management Entities and Layer Interfaces
- Volume 3: Network Protocols

Base System Software, volume 1, describes the CDCNET Device Interface (DI) startup and system management software: its base system software. The volume begins with an overview of each base system software component, and continues with details of the procedures and functions provided by these software components for general use.

Network MEs and Layer Interfaces, volume 2, describes each of the network management entities (MEs) and layer interfaces for CDCNET. The volume also describes the interfaces to non-CDCNET systems. These interfaces include interfaces to NOS hosts and X.25 Packet level networks. The information described in this volume is essential for programmers who intend to write gateway software that will reside in CDCNET.

Network Protocols, volume 3, defines CDCNET network protocols. The use of protocols enforces consistent communication between software entities and the transitions that result. The description of each protocol follows a rigid documentation guideline, called the Finite State Machine (FSM). The information described in this volume will be useful to programmers implementing Control Data Network Architecture (CDNA) on a foreign host or network.

Volume 1: Base System Software

This manual begins with a general introduction of the role CDCNETS's base system software plays within the network architecture. This software is then described in detail. Base system software procedures and functions that can be called are then presented in alphabetical order with the information needed to call them. Other relevent information is presented in the appendices, including the common types used in making procedure calls.

Chapter 1 introduces the role of CDCNET's base system software within the network architecture.

Chapter 2 describes the process of initializing a CDCNET system.

Chapter 3 describes the general concepts and software management areas controlled by the executive software.

Chapter 4 describes the software providing statistics management services.

Chapter 5 describes the software providing status management services.

Chapter 6 describes the software providing table management services.

Chapter 7 describes the software providing the online loader.

Chapter 8 describes the software providing failure management.

Chapter 9 describes the Device Manager, which provides the interface between CDNA's physical and link layers.

Chapter 10 describes the common routines of the base system software that provide the services described in the previous chapters.

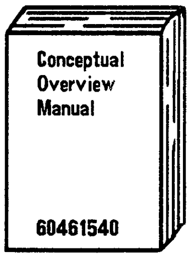
Appendix A provides a glossary of CDCNET terms and acronyms.

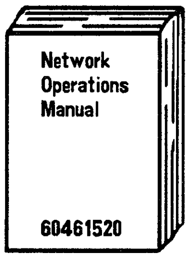
Appendix B provides definitions of descriptor and data buffers.

Appendix C provides an alphabetical list of CYBIL-defined common types discussed in this manual.

Related Manuals

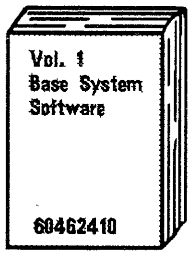
Background (access as needed):

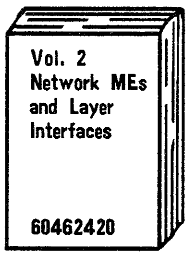


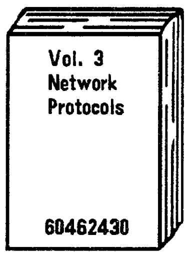


Software development manuals:

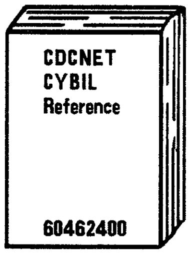
CDCNET Systems Programmer's Reference Manual

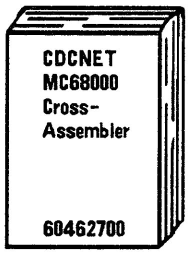


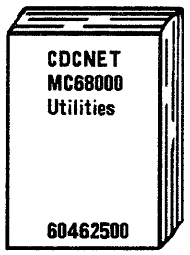




Software tools manuals:







Additional Related Publications

The following manuals are referenced in this document. These manuals should be used when more extensive information about the topics covered herein is required.

Manual Title	Publication Number
CDCNET Network Analysis Manual	60461590
CDCNET Troubleshooting Guide	60462630

Ordering Manuals

Control Data manuals are available through Control Data Sales offices or through:

Control Data Corporation
Literature Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Submitting Comments

Control Data welcomes your comments about this manual. Your comments may include your opinion of the usefulness of this manual, your suggestions for specific improvements, and the reporting of any errors you have found.

You can submit your comments on the comment sheet on the last page of this manual. If the manual has no comment sheet, mail your comments on another sheet of paper to:

Control Data Corporation
Technology and Publications Division, ARH219
4201 Lexington Avenue North
St. Paul, Minnesota 55126-6198

You can also submit your comments through SOLVER, an online facility for reporting problems. To submit a documentation comment through SOLVER, do the following:

1. Select `Report a new problem or change in existing PSR` from the main SOLVER menu.
2. Respond to the prompts for site-specific information.
3. Select `Write a comment about a manual` from the new menu.
4. Respond to the prompts.

Please indicate whether you would like a written response.

Introduction 1

Introduction 1

CDCNET Device Interfaces (DIs) manage the exchange of information between network interfaces. Information being moved from point A to point B on the network must be packaged and repackaged on its way. The data doesn't simply move horizontally across the network on the same level at which it came in. It must also move down through the network layers to the physical layer and back up again; otherwise, it cannot traverse the network.

Moving information both horizontally and vertically through the network requires the specialized services of CDCNET management entities (MEs) and layer software. These services, in turn, must rely on base system software to provide the necessary environment for their activities. Base system software creates and maintains the system environment in a CDCNET DI. Without it, the system's MEs and layer software could not function.

Base system software creates an operational environment in a DI by taking advantage of the distributed processing design of CDCNET. Working on a buddy system, DI software requesting help to initialize gets that help from an already operational DI. Software that is loaded in this way continues the initialization process from within the initializing DI. Initialization is complete when the DI is up and running normally.

While a DI is operating normally, its base system software maintains the system environment for the MEs and layer software to use. Maintaining this environment means providing the following services:

- Integer- and string-keyed table management
- Allocation and de-allocation of system resources
- Online loading
- Device status table management
- External device interrupt management
- Comprehensive failure management
- Collection and reporting of statistics

The primary purpose of this manual is to provide enough information to enable you to use common base system software routines in your TIPs, gateways, or other CDCNET-compatible software. These routines are detailed in chapter 10. Between here and chapter 10, you will read about the special concepts and activities of CDCNET base system software components. This information is provided in the context of what these components can do for you in your own programming.

System Initialization 2

System Initialization 2

The initialization process defines how the software is initially loaded and started in a CDCNET system. This process is designed to accommodate a first-time load as well as the needs of restoring a failed system to its operational state. CDCNET system initialization is a distributed process that requires the services of another, operational system. The initialization process consists of the following steps:

- After DI power is turned on or the DI is reset, the hardware quicklook diagnostics are run. If the diagnostics complete successfully, control is transferred to the DI-resident Main Bootstrap Controller.
- The Main Bootstrap Controller selects a board across which to attempt a bootstrap and moves board-specific ROM code from that board into system main memory (SMM) to begin the bootstrap process.
- The DI-resident Initialization Bootstrap is then called to assist in loading across the selected board. This subroutine requests help in loading from an operational DI or host system. If a Terminal DI (TDI) is initializing, it looks to a Mainframe DI (MDI) for help. If an MDI or Mainframe Terminal Interface (MTI) is initializing, it looks to the host.

  The operational system determines if memory is to be dumped from the initializing system, based on information contained in the help request. If a dump is required, it is written to a file on the host before the loading is attempted. The boot file is then loaded into the initializing DI.
- Once the boot file has been loaded into DI memory, the Initialization Bootstrap transfers control to the transfer address provided by the board-specific routine. This corresponds to the Initial Loader, which is the first section of the boot image; it arrives prelinked and ready for execution.

  The Initial Loader (described in more detail later) allocates memory and links the remaining software modules in the boot image. It then initializes the Executive and starts the System Ancestor as a task.
- The System Ancestor, in turn, starts the basic set of tasks common to all DI variants, plus the software required to drive the network interface that was used in the initial load. When the System Ancestor completes its initialization function, the minimum set of system tasks required to support the DI environment has been started.
- Among the tasks started by the System Ancestor is the Configuration Procurer. The Configuration Procurer oversees the starting and execution of configuraton commands that are specific to the DI being initialized.

The Initial Loader

The Initial Loader is the first section of the boot file to be loaded into the DI during initialization. It is loaded as a prelinked, absolute record that is ready for execution. The Initial Loader is immediately followed in the boot image by the other DI software modules.

The Initial Loader loads the software modules from the boot image into the memory space of the DI hardware configuration (making the necessary linkages) and initiates software execution. Initiating software execution requires initialization of the Executive and startup of the System Ancestor task.

Program Naming Conventions

There are three hardware boards that serve as storage locations for programs that will run on the MPB processor: the MPB itself, the Private Memory Module (PMM), and the System Main Memory (SMM). The following naming convention is used to ensure proper software module loading in the available hardware configuration.

program_name_MPB	Preferred residence is in MPB memory.
program_name_PMM	Preferred residence is in PMM memory.

Allocation of MPB, PMM, and SMM is hierarchical in that order. If there is not enough PMM, for example, a program with a PMM suffix will be placed in the SMM. If a program does not have a recognizable suffix to its name, the loader assumes that the program is to be stored in the SMM.

NOTE

Because the modules are examined and loaded sequentially from the boot image, modules that have a critical need for residence in the limited space of the MPB or the PMM should be placed early in the boot file so that they will be loaded early. For example, the Executive (EXEC_MPB) is the first record in the boot image after the Initial Loader.

The Executive 3

The Executive 3

The Executive offers procedures that let users share the system's available processing and memory resources efficiently. It is the kernel of the DI software.

This chapter presents some general concepts that apply to the Executive and goes on to give brief descriptions of the five specialized services that the Executive provides to its users. They are:

- Task Management
- Memory Management
- Message Management
- Queue Management
- Timer Management

General Concepts Relating to the Executive

Interrupt Service Routines

To achieve needed flexibility, the Executive distinguishes between two types of processing requests: Interrupt Service Routines and tasks.

Interrupt Service Routines execute in response to the demands of external interfaces that the system does not itself control, such as intelligent peripheral (IP) communications and Mainframe Channel Interface (MCI) I/O completion. These requests execute in supervisor state, the higher of two privilege states of CPU operation.

Interrupt Service Routines communicate with other code using intertask messages (ITMs). ITMs are specific in their form and content and are identified by number. They are sent to the normal or express queue of the destination task.

The Executive uses firewalls to facilitate its error management for Interrupt Service Routines. Firewalls are special structures in execution stacks that allow code to return to a known checkpoint on demand. They are constructed within Interrupt Service Routines by means of supervisor calls that save the stack address, registers, and hardware status. By using firewalls, failures within Interrupt Service Routines are less serious; the system stack is recovered and the associated routine is notified of the error.

Tasks

System tasks perform the system's ordinary network layer and management functions. Tasks execute in the user state, the lower of two privilege states of CPU operation. Because task code responds in a less reactive fashion than Interrupt Service Routines, it may be dispatched and execute at different priority levels within the user state, depending on realtime response requirements.

The Executive schedules task execution according to the task's priority level and preemptibility.

Each task has a unique User Stack for keeping track of variables, parameters, and registers during the chaining of calls required in its execution. The stack maintains information specific to the task to which it belongs. Unused stack length may be checked using the UNUSED_STACK_ function described in chapter 10.

In addition to making direct calls for services, tasks may communicate with one another using ITMs.

Parent Tasks

Task recovery is enhanced by the concept of parent tasks, in which one task oversees the execution of another. A parent task oversees execution of its child task by servicing any ITMs from the Executive regarding Interrupt Service Routines and by executing recovery mechanisms in response to these messages.

Task Control Block

The Executive may redirect CPU processing from one task to another at any point during a task's execution. When this occurs, the context of the executing task must be saved until the Executive can schedule completion of that task. The Executive saves a task's context in a task control block (TCB).

The TCB is addressed by its task_ptr (or taskid), which is created when the task is started. The TCB maintains the task state and includes, among other things:

- Stack length and pointers
- Parent/child task pointers
- Task state and transition counters
- Queue Control Blocks (QCBs)
- Register save area
- Task status register and program counter

Refer to the data structures in appendix B of this manual for the CYBIL definition of the TCB structure.

Message Queues

Each task has both a normal and an express message queue. A separate QCB is maintained in the TCB for each of these queues. Messages sent to a task's normal message queue are generally concerned with intertask services. Messages sent to the express message queue are generally concerned with task error notifications.

Each message queue is emptied in first-in, first-out (FIFO) order. The express queue takes priority; it is emptied first.

Task Attribute Flags

Two attribute flags are associated with each task and maintained in the task's TCB. They are:

Preemptible/Not Preemptible

This flag indicates if the task may be preempted by the scheduling of higher priority tasks. If processing has begun on a task that is not preemptible, exception processing must return to that task even if higher priority work is subsequently scheduled (unless the task is aborted or an immediate control task is scheduled).

If a preemptible task is preempted, the Executive reschedules that task's processing after the higher priority task completes.

Immediate Control

This flag overrides the preemptible/not preemptible flag. If a task is an immediate control task, then the Executive schedules its requests immediately, regardless of the preemptibility of the previously executing task. Only recovery tasks are immediate control tasks.

NOTE

This feature is not presently implemented. The immediate control flag will be passed, but it has no effect on task scheduling.

Task Management

The Executive provides a flexible task management environment as one of its specialized services. Tasks may start, stop, or delay their own or another task's processing. They may also change their own priority, preemptibility, and state or those of other tasks.

Besides passing work, tasks need to control access to shared resources (tables, for example). The standard method of controlling shared access is a semaphore that indicates whether a resource is in use. The Executive provides a semaphore service (SIGNAL(n)/ACQUIRE(n)).

Following are brief descriptions of the Executive's task management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
ABORT_TASK	Aborts the execution of a task
DELAY_PROCESSING	Temporarily delays task processing
NEW_INTERRUPT	Announces an interrupt service routine
NEW_PRIORITY	Changes a task's priority
NOPREMPT	Supresses task preemption
OKPREMPT	Restores task preemption
RESTORE_TASK	Restores task from a suspended state
SIGNAL(n)/ACQUIRE(n)	Tests for, and sets, semaphores
START_SYSTEM_TASK	Starts a task with the system ancestor as its parent
START_TASK	Starts task at procedure entry point
STOP_TASK	Stops task execution
SUSPEND	Suspends task execution
WAIT	Puts task in a wait state
WAKE_UP	Wakes a waiting task
YIELD	Yields control of the CPU

Memory Management

The Executive manages the allocation of a configurable amount of system memory for use by tasks. During times of memory congestion, the Executive is still able to service critical memory requests by accessing other memory known to be free at the time of the request. This section describes some of the basic memory management concepts associated with these services.

Data and Descriptor Buffers

The Executive allocates memory efficiently by distinguishing between data and descriptor buffers (also referred to as long and short buffers, respectively). CYBIL definitions for buffers are given in appendix B of this manual.

Data buffers hold user data. Associated with data buffers are descriptor buffers, which keep track of data buffer specifics such as chaining, usage counts, data offsets, data counts, and pointers. A configurable percentage of data and descriptor buffers is allocated for management by the Executive when the system is initialized. The relationship between descriptor and data buffers is illustrated in figure 3-1.

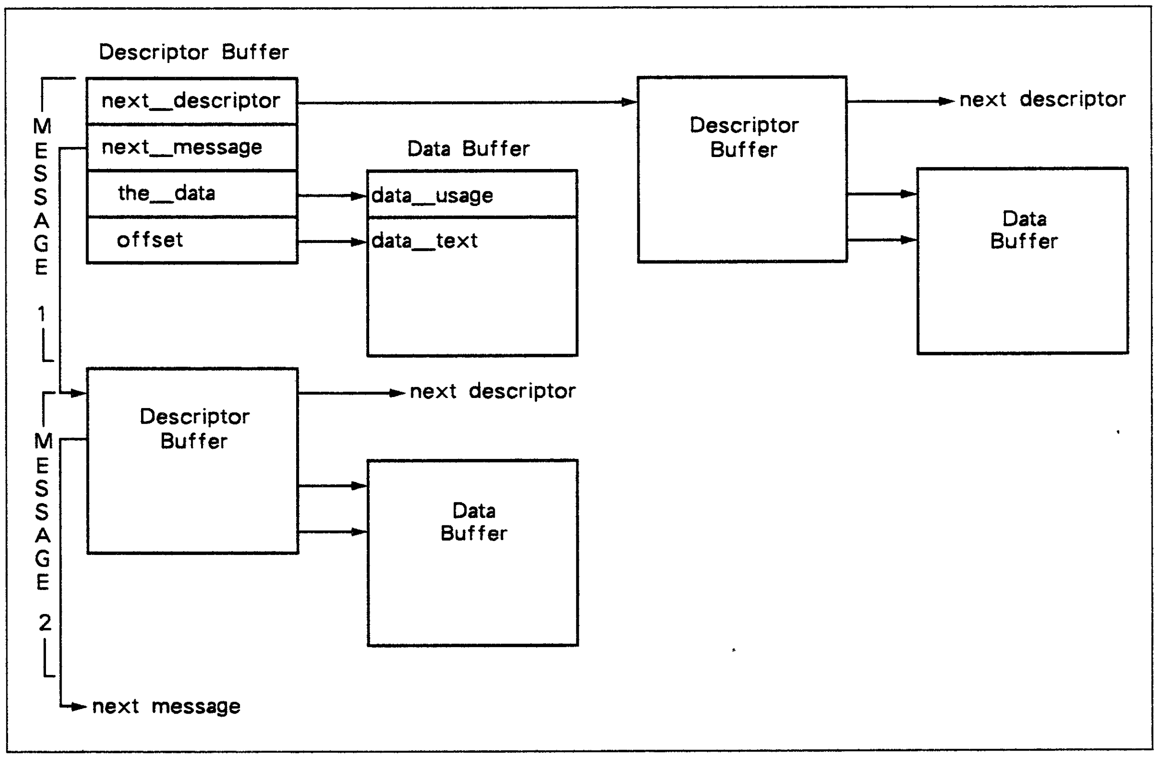


Figure 3-1. Descriptor and Data Buffers

Buffer and Memory States

Four states are defined to indicate the relative availability of buffers and memory in the DI. They are hierarchical in this order: good, fair, poor and congested.

The boundaries between these states are set during configuration and may be changed using network operations commands. Each boundary is expressed as a percentage of the total resource currently available. For example, the boundary between fair and poor might be 20% of memory; if only 15% of memory is currently available, the memory state is said to be poor.

Priorities and Thresholds

The input parameter *threshold* indicates the priority level for data buffer and memory extent requests. Descriptor buffer requests do not require a priority; they are always accepted.

Four data buffer request priorities are defined:

Critical	For requesting buffers for use in sending operator alarms and commands or completing any actions initiated to solve memory congestion problems. Critical requests are accepted in all buffer states.
High	For requesting buffers for use in receiving incoming data, sending a command response, or completing any error processing and recovery. High priority data buffers requests are accepted even if the buffer state is congested.
Medium	For requesting buffers for use in sending Routing protocol data units (PDUs) or error-related log messages. Medium priority data buffer requests are not accepted if the buffer state is congested.
Low	For requesting buffers for use in sending nonerror related log messages, such as statistics log messages. Low priority data buffer requests are not accepted if the buffer state is poor or congested.

Only two memory extent request priorities are defined:

Critical	Critical requests are accepted in all memory states.
Normal	Normal memory extent requests are not accepted if memory state is congested.

Procedures and Functions

Following are brief descriptions of the Executive's memory management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
CLEAR_ALLOCATE	Allocates space from free global memory and clears it
CLEAR_MEMORY	Clears specified memory
CLEAR_WRITE_PROTECT	Clears write protect flag
GET_MEMORY	Gets global memory extents
GET_MPB_EXTENT	Gets MPB RAM memory extents
GET_PMM_EXTENT	Gets private memory extents
MODIFY_WRITE_PROTECT_ BYTE	Modifies write-protected MPB byte
MODIFY_WRITE_PROTECT_ LONG_WORD	Modifies write-protected MPB long word
MODIFY_WRITE_PROTECT_ SHORT_WORD	Modifies write-protected MPB short word
SET_BUFFER_CHAIN_OWNER	Sets owner identification for buffer chain
SET_MEMORY_OWNER	Sets owner identification for memory location
SET_WRITE_PROTECT	Sets the write protect flag

Message Management

In addition to the memory management services just described, the Executive also provides routines that perform data manipulation of messages. Following are brief descriptions of these routines. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
APPEND	Appends a trailer to a message
ASSEMBLE	Assembles message fragments
BROADCAST	Prepares a message for broadcast
BUILD_HEADER_IN_PLACE	Builds a message header
COPY	Copies a message to another buffer chain
FG_TRIM	Trims bytes from the end of message
FIRST_BYTE_ADDRESS	Returns message first byte address
FRAGMENT	Extracts message fragment
GEN_DATA_FIELD	Generates a data field in Management Data Unit (MDU) format
GEN_TEMPLATE_ID	Builds template identifier fields into buffers
GET_DATA_FIELD	Extracts data fields from MDU formatted message
GET_FIRST_BYTE	Returns first byte of message
GET_LAST_BYTE	Returns last byte of message
GET_LONG_BUFFERS	Gets one or more data buffers
GET_MESSAGE_LENGTH	Returns byte length of message
GET_SHORT_BUFFERS	Gets one or more descriptor buffers
GET_SIZE_N_ADDR	Gets size and address of memory extent
M_RELEASE	Decrements message usage count
PCOPY	Copies message to new buffer chain, releasing old buffers
PREFIX	Adds header to front of a message
RELEASE_MESSAGE	Releases data buffer chains
STRIP	Removes header from front of a message

Procedure	Description
STRIP_IN_PLACE	Removes header from front of a message, without changing message address
SUBFIELD	Gets multiple-byte header fields
TRANSLATE_MESSAGE	Translates between character sets
TRIM	Trims bytes from end of descriptor buffer

Queue Management

As described earlier, there are two QCBs maintained in each task's TCB. They are the Express Message QCB and the Normal Message QCB. These exist for the lifetime of the task.

The Express Message QCB is used to inform a task of the successful or unsuccessful completion of a service request, or to notify it of some error condition it must service. The Express Message QCB is always inspected first.

The Normal Message QCB is used for any other ITMs. It is inspected only after the express queue has been serviced.

A QCB contains all the information needed to add new elements to the queue and to access and remove current queue elements. It holds a counter indicating the number of messages currently on the queue and a counter that tallies the total number of elements ever added to the queue. Each QCB contains pointers to both the first and last elements currently on the queue.

Following are brief descriptions of the Executive's queue management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	**Description**
GET_EXPRESS	Gets ITM from express queue
GET_MSG	Gets ITM from the express or normal queue (express queue first)
MESSAGE_DEQUEUE	Extracts a message from the specified task-level queue
MESSAGE_ENQUEUE	Places a message in the specified task-level queue
SEND_EXPRESS	Sends ITM to the express queue of the specified task
SEND_NORMAL	Sends ITM to the normal queue of the specified task

Timer Management

The Executive provides timer management services to its users. Included in these services are three ways for a task to request nonimmediate procedure execution: at a given time, after an interval, or periodically. If specified, parameters may be passed to the task at execution time in non-immediate requests.

Following are brief descriptions of the Executive's timer management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
CALL_AFTER_INTERVAL	Procedure is called after specified time interval
CALL_AT_TIME	Procedure is called at specified time
CALL_PERIODIC	Procedure is called periodically
CANCEL_TIMER	A previously requested timing request is cancelled
CHANGE_TIMER_OWNER	Changes allocating task identifier for timing procedure
DELAY_PROCESSING	Delays processing for a finite period of time
PMP_GET_DATE	Returns current date in specified format
PMP_GET_TIME	Returns current time in specified format
READ_BCD_CLOCK	Reads the real-time clock in BCD format
READ_CLOCK	Reads the binary clock to millisecond accuracy
SET_BCD_CLOCK	Sets the real-time clock
TIME	Converts time intervals (including time-of-day) to milliseconds

Statistics Management 4

Statistics Management 4

Part of CDCNET's base system software provides for the collection and reporting of system statistics. In general, software components concerned with collecting statistics open Service Access Points (SAPs) to the CDCNET Statistics Manager (CSM). These SAPs point to a linked list of one or more statistics data structures (SDSs), which are used for the collection and reporting of statistics. The SDSs are specified by the software component when opening a statistics SAP.

CSM acts as the bridge between commands that request statistics collection and the software that will actually collect the statistics. CSM sets a flag in the SDS that tells the collecting software component whether or not to collect statistics. The software component may ignore the flag and always collect statistics if more resources are required to check this flag than to collect the statistics.

Two statistics buffers are supplied by the software component. While one is used for statistics collection, the other is being reported.

Statistics SAP entries can be located using element type (network solution, communication line, or software component) and element names supplied by the command processors. Each SDS header is defined by a statistics group type. The statistics group types are:

summary	normal process statistics
expanded	statistics beyond normal processing
debug	statistics for debugging

CSM manages statistics reporting by calling software component-supplied procedures that: 1) generate the log message containing collected statistics and 2) call the Log Support Application to issue the log message. CSM calls these procedures whenever statistics are to be reported.

The services performed by CSM are summarized in figure 4-1.

SAP Management

Statistics SAPs can be opened and closed and SAP entries found. In general the SAP management service helps create a dynamic association between command processors and statistics collecting software components.

Statistics Collection

The CSM may enable or disable collection of statistics while specifying a collection interval. CSM users may issue commands to control the collection of statistics without having direct contact with the particular software components that do the collecting, or being aware of the associated data structures.

Statistics Reporting

Statistics are reported periodically according to time intervals specified by CSM commands. They are also reported when the corresponding statistics are started or stopped, or when the statistics SAP is closed. Report procedures supplied by the statistics collecting software components are called to generate and issue the log messages containing the statistics.

Statistics Management Relationships

Figure 4-1 illustrates the central role that CSM plays in the management of CDCNET statistics.

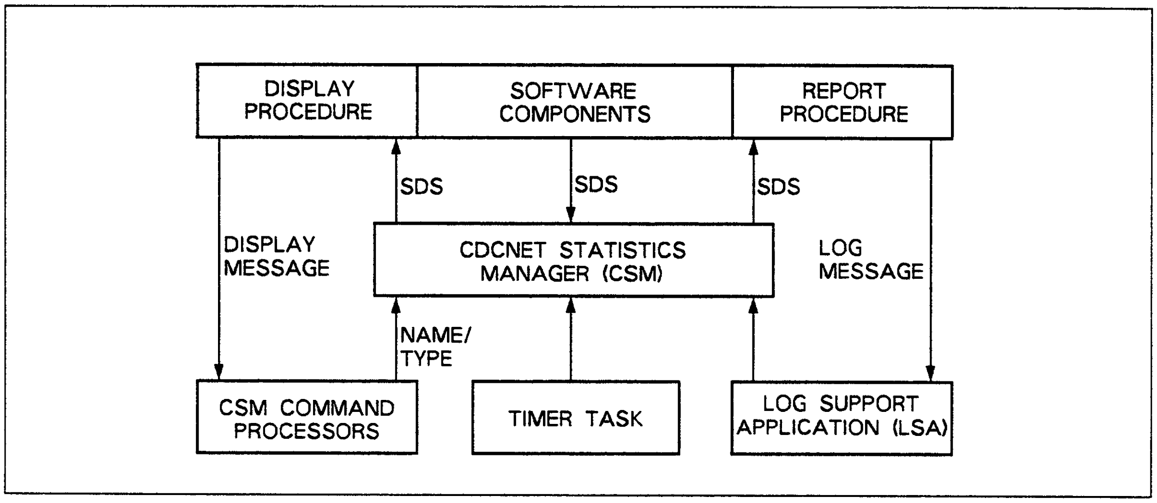


Figure 4-1. CDCNET Statistics Manager

Procedures and Functions

Following are brief descriptions of the statistics management procedures and functions. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
OPEN_STATISTICS_SAP	Opens a new statistics SAP
CLOSE_STATISTICS_SAP	Closes a previously opened statistics SAP
FORCE_STATISTICS_REPORTING	Forces call of the specified statistics function procedure
BUILD_STATISTICS_MSG_HEADER	Initializes statistics message header
INITIALIZE_STATISTICS_RECORD	Initializes MDU-formatted records used in statistics collection

Status Management 5

Status Management 5

This chapter describes CDCNET status management software. It identifies system device status tables, describes the various states and statuses of CDCNET hardware devices and the transitions between them, and discusses the storage and retrieval of state and status information.

Hardware Device Status Tables

Each CDCNET system contains the following device status tables.

- Major Card Status Table (MCST)
- Line Interface Module (LIM) Status Table (LST)
- Port Status Table (PST)
- SMM Bank Status Table (SBST)
- PMM Bank Status Table (PBST)

The MCST is a static table and contains an entry for each of the eight slots in a DI that are reserved for the major boards. Each entry contains information about the type of board installed, its state, status, and other relevant information. MCST entries of board type CIM contain a pointer to the LST. Similarly, entries for board types SMM and PMM contain pointers to the SBSTs and PBSTs, respectively.

The LST is also a static table. It contains an entry for each of the eight slots reserved for the LIM boards. Each entry contains information about the type of LIM (for example, RS449), its state, status and a pointer to the PST.

A separate PST exists for each LIM that is physically present in the DI. Each PST contains up to four entries for the ports on the associated LIM, and each entry contains information about the state and status of the associated port.

Each SBST or PBST contains information about the state and status of the memory banks on the associated SMM or PMM board.

All hardware device status tables are created by the Initial Loader prior to starting the System Ancestor as the initial task in the DI. The board map table created by the on-board diagnostics is used to determine which devices are physically present in the DI and identifies their initial states. For a formal description of the board map table, refer to the CDCNET Network Analysis Manual.

States and Statuses

This section describes the possible values for the state and status of each of the hardware elements in a CDCNET system. It also describes how these values get updated and identifies the valid state transitions.

State of a Hardware Device

At any point in time, the state of a hardware device that is physically present in a CDCNET system can be ON, OFF or DOWN.

The ON state implies that the device is either fully operational or operating in a degraded mode. The ON state further implies that the device is available for use by the system.

The OFF state indicates that the device is not available to the system or diagnostics. A device may be placed in the OFF state to prohibit its further use. The logical equivalent of this state is that the device is not present in the system.

The DOWN state indicates that a device is only available for use by the diagnostic software, and therefore is not available for normal system use.

Initializing the State of Hardware Devices

When a CDCNET system is loaded, the following process is used to initialize the state of hardware devices.

The on-board diagnostics always build a table called the card map table. This table contains, among other things, the result of the execution of the on-board diagnostics on each device as well as the setting (ON or OFF) of the offline switch on each device. Status software uses this information to initialize the states of different devices.

The state of a device is set to OFF if the offline switch is ON. If the offline switch is OFF and the on-board diagnostics have executed successfully on a device, its state is set to ON. If the offline switch is OFF and the on-board diagnostics have not run successfully on a device, then the device state is set to DOWN.

Status of Hardware Devices

A status will be associated with each device at all times. Following is the list of device status values.

- **not_configured**
- **configured**
- **enabled**
- **active**

Below is a table describing the processes and rules used to change the status of the main board devices.

Board Type	Status or Status Transition
MPB	Always **configured**.
SMM	If ON, always **configured**. If OFF, then **not_configured**.
PMM	If ON, always **configured**. If DOWN or OFF, then **not_configured**.
ESCI	Initially, **not_configured**. After a trunk or interface is defined (using the DEFINE command), then **configured**. After an associated CANCEL command, then **not_ configured**. After successful START_TRUNK command, then **active**. After an associated STOP_TRUNK command, then **configured**.
MCI	Initially, **not_configured**. After a trunk or interface is defined (using the DEFINE command), then **configured**. After an associated CANCEL command, then **not_ configured**. After a START_TRUNK command, then **enabled**. If the channel interface is brought up successfully, then **active**; otherwise it remains **enabled**. The status of an MCI card will be changed from **active** or **enabled** to **configured** as a result of successful execution of an associated STOP_TRUNK command.
CIM	Initially, **not_configured**. If any connected LIM is configured, then **configured**. After DEFINE_CIM_INTERFACE, then **configured**.

NOTE

The status of a CIM, ESCI, MCI, LIM or a port is **active** if it is in the DOWN state and is currently being tested by online diagnostics.

Status of LIMs and Ports

After initialization, the status of any LIM or port is **not_configured**. However, after a line or a trunk is defined for a port, the status of the port becomes **configured**. The status of a LIM changes to **configured** if any one of its ports has a configured status. The enabled status does not apply to LIMs.

The status of a port is changed to **not_configured** when the line or trunk defined to use it is cancelled. LIM status is changed from **configured** to **not_configured** after all ports connected to it become **not_configured**.

Port status is changed from **configured** to **enabled** when the START command executes successfully. Status will change from **enabled** to **active** when the port's line or trunk becomes active. Port status is changed from **active** or **enabled** to **configured** when a STOP command is executed successfully.

In the case of an HDLC or X.25 trunk, the term **active** indicates successful completion of the protocol handshake with the Layer 2 peer. In the case of an asynchronous line, the term **active** implies connection with a user of the line.

State Transitions

The following table indicates the possible state transitions and how they occur.

From	To	Process
ON	OFF	This state transition is made using an operator command called CHANGE_ELEMENT_STATE. It is allowed only if the element is not system critical and is not in active use. The MPB, SMM and PMM are system critical devices. An element is considered active if its status is enabled or active.
ON	DOWN	The process and rules that apply to the transition from ON to OFF also apply to the transition from ON to DOWN.
OFF	ON	This state transition is only made using an operator command. It is recommended that the state first be changed to DOWN, followed by an online diagnostics test to make sure that the device can be used by the system.
OFF	DOWN	This state transition is only made using an operator command.
DOWN	OFF	This state transition is only made using an operator command. It is not allowed if the device status is active. In the DOWN state, active status indicates that an online diagnostic test is being run on the device.
DOWN	ON	This state transition is made using an operator command or by online diagnostics after successful testing. The operator command can make this transition while device status is not active.

Acquisition and Release of a Hardware Device

The following routines are provided for request and release of hardware devices. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
GET_CARD_TYPE_AND_ADDRESS	Gets the board type and address for the specified device
RELEASE_HARDWARE_DEVICE	Releases a previously configured hardware device
REQUEST_DIAGNOSTIC_ENTRY	Gets the address of the system status table entry for the specified device
REQUEST_HARDWARE_DEVICE	Configures the specified hardware device

Accessing and Updating the Hardware Device Status Tables

Any software component can copy entries from hardware device status tables using the procedure GET_STATUS_RECORD. The owner of a hardware device can update or re-write an entry in the hardware device status tables using PUT_STATUS_RECORD.

Following is a complete list and brief descriptions of the status management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
CLOSE_STATUS_SAP	Closes previously opened status SAP
GET_NEXT_STATUS_SAP	Retrieves address of software component status table
GET_STATUS_RECORD	Retrieves status record for specified device
GET_STATUS_SAP	Retrieves address of software component status table
OPEN_STATUS_SAP	Registers the address of a software component status table
PUT_STATUS_RECORD	Updates the status record for the named device

Tree (Table) Management 6

Tree (Table) Management 6

Several of the layer and management entities in a DI logically connect a user and either a correspondent or a service using random access tables that are organized and accessed according to a balanced, binary tree solution. This solution provides relatively fast access to any table entry. And because new tables are allocated dynamically, memory is only used for tables allocated.

To create a tree structure, the user must initialize the tree root. The root is a separate, user-created table that records the number of tables in the tree, validation information, and the address of the tree's first node. Tables are accessed by specifying a pointer to the root of the tree being referenced.

A key is used to reference each tree node and its associated table. It is either a 32-bit integer or an adaptable string, depending on the type of tree structure created. Keys are used in tree searches.

Following are brief descriptions of the tree management procedures. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
FIND	Returns the table in a tree associated with the specified integer key
FIND_FIRST	Returns the first table and key that satisfy an input parameter
FIND_FREE_NODE	Finds the first available node
FIND_NEXT	Returns the next table and key that satisfy an input parameter
FIRST_NODE	Allocates space for the first node of a tree
GROW	Adds a new table to a tree
INIT_ROOT	Initializes the root of a tree
PICK	Removes a structure from the specified tree
SFIND	Like FIND, for string keyed trees
SFIND_FIRST	Like FIND_FIRST, for string keyed trees
SFIND_NEXT	Like FIND_NEXT, for string keyed trees
SFIND_WILD_CARDS	Locates wild card matches in a tree, for string keyed trees
SGROW	Like GROW, for string keyed trees
SPICK	Like PICK, for string keyed trees
VISIT_ALL_NODES	Steps through a tree structure, making successive calls to a user supplied routine for each node found

Online Loader 7

Online Loader 7

The Online Loader provides a method for dynamic loading and deloading of software modules within an operational system. The Online Loader provides the following services, described separately below:

- Module Loader
- Module Deloader
- Translation to Modules

Module Loader

The Module Loader service of the Online Loader loads modules by entry point name and module name. The load process consists of reading the object module file, allocating memory for the module sections, moving code and data into the sections, linking external references, and incrementing module use count.

The Module Loader is a separate module in the standard system and may be present or absent independently of other system features.

Module Deloader

The Module Deloader service deloads unused modules. The deloading process consists of deallocating the memory used for the module sections and removing entry point names from the internal loader tables.

The Online Loader's interlock feature prevents the deloading of modules while they are still in use. The module use count indicates the number of tasks or other processes that require a given module. Only when the module use count has gone to zero may the module be removed from the system.

The Module Deloader is a separate module in the standard system and may be present or absent independently of other system features.

Translation to Modules

The Translation to Modules feature provides the entry point name and module name translation to already loaded modules or modules to be loaded. It also provides translations to support the checksum of all loaded modules, validate the program counter, and return transfer addresses.

Procedures and Functions

Following are brief descriptions of the Online Loader interface procedures and functions. Complete documentation of these procedures can be found in chapter 10 of this manual.

Procedure	Description
CHECKSUM_NEXT_MODULE	Runs a checksum of the sections of the next module
DECREMENT_MODULE_USE_COUNT	Decrements the use count of a loaded module
INCREMENT_MODULE_USE_COUNT	Increments the use count of a loaded module
LOAD_ABS_MODULE_AND_DELAY	Loads an absolute module (if not already loaded) while the caller waits
LOAD_ABS_MODULE_AND_PROCEED	Loads an absolute module (if not already loaded); the caller does not wait
LOAD_CMD_PROCESSOR_AND_DELAY	Loads a command processor module (if not already loaded) while the caller waits
LOAD_CMD_PROCESSOR_AND_ PROCEED	Loads a command processor module (if not already loaded); the caller does not wait
LOAD_ENTRY_POINT_AND_DELAY	Loads the module associated with a given entry point (if not already loaded) while the caller waits
LOAD_ENTRY_POINT_AND_PROCEED	Loads the module associated with a given entry point (if not already loaded); the caller does not wait
START_NAMED_TASK_AND_DELAY	Loads module associated with a given entry point (if not already loaded) and starts it as a task while the caller waits
START_NAMED_TASK_AND_PROCEED	Loads module associated with a given entry point (if not already loaded) and starts it as a task; the caller does not wait
VALIDATE_SECTION_ADDRESS	Translates a given address into a module name and section address

Failure Management 8

Failure Management 8

This chapter describes the CDCNET hardware failure management philosophy and its present implementation. The present implementation is not a completed example of the more highly developed philosophy, but is straightforward and in accord with that philosophy.

Hardware failures include the failures of the hardware elements in a DI and the media connected to a DI, as well as the failures caused by illegal use of the DI hardware by software (for example, use of an illegal instruction).

The software that supports failure management in a CDCNET system is referred to collectively as the failure management software. This software has the following objectives.

- It supports the detection and reporting of all hardware failures. The failure report identifies the field replaceable unit (FRU) or units responsible for the failure.
- It provides a method of failure recovery.
- It monitors the behavior of hardware elements and reports behavior that is out of the expected norm.

This chapter continues with a description of failure management concepts, presents a general model of CDCNET failure management, covers the four failure management subsystems in a CDCNET system, and describes the logging of CDCNET failures.

Failure Management Concepts

Hard and Soft Errors

Hardware failures can be classified as hard or soft. A failure is said to be hard if the damage done by it cannot be undone. One example of this is a read parity error. In this case, the information being read has been permanently corrupted.

A failure is said to be soft if the damage done by it can be undone through a recovery process in hardware or software. One example of a soft failure is a single bit error, which is corrected automatically in hardware. Another example of a soft failure is a Cyclic Redundancy Check (CRC) error in an High-Level Data Link Control (HDLC) frame. This error is recovered from when software retransmits the HDLC frame.

Unclassified Events and Failures

CDCNET failure management software also reacts to certain events or failures that are not classified as hard or soft. Unclassified events include the warnings generated by hardware to inform software that the DI battery is getting weak or that the temperature in the DI is getting too high.

Unclassified failures include abnormal operation of the data carrier and data set ready signals on a line connected to a LIM through a modem. Such failures can occur due to noise on the transmission line or during the normal disconnect sequence on some lines. These failures are similar to soft failures because one can correct for them by retrying the operation in which the failure is detected. These failures may also be ignored unless they reach a preset threshhold.

Failure Management Functions

Failure management software has three important functions: detection, reporting, and recovery.

Failure Detection

Most failures are detected by hardware. If hardware detects a failure, it notifies software about it by writing the failure information into a status register and optionally generating an error interrupt (for example, a bus error interrupt). The CDCNET failure management software supports failure detection by providing the interrupt handlers to process the various error interrupts and by periodically monitoring the hardware status registers, a change in whose contents is not always accompanied by an interrupt.

Failure Reporting

Reporting of failures includes extracting the failure information from the hardware status registers and saving it in memory; it also includes notifying the software that executes on the failed element about the failure and logging the failure information in the CDCNET log file with a log message.

If the failure causes a reset, however, then users of the failed element will not be notified of the failure.

Failure Recovery

In the recovery phase, CDCNET software determines whether the failed element can be used again and, if so, through what level of recovery. Necessary hardware and software is re-initialized and users of the failed element are notified of its renewed availability.

Structure of the Failure Management Software

Hardware failures in a CDCNET system are managed on a subsystem basis. Each subsystem processes failures on one or more specific boards (for example, the MCI board) and the media connected to it. CDCNET failure management software defines four subsystems: MPB, CIM, ESCI and MCI. In addition to these subsystems, the failure management software includes some common code which is used by all subsystems. There is a general model for CDCNET failure management on which each subsystem is based. This general model is described below, followed by descriptions of the individual subsystems.

General Model for Failure Management

Figure 8-1 shows the model used as the basis for failure management in each subsystem. The generalized model requires the following functions to be present in each subsystem.

- Fault handlers
- A subsystem failure table
- A subsystem failure management task

In addition, this model requires a common service to maintain statistics about different failures and recoveries.

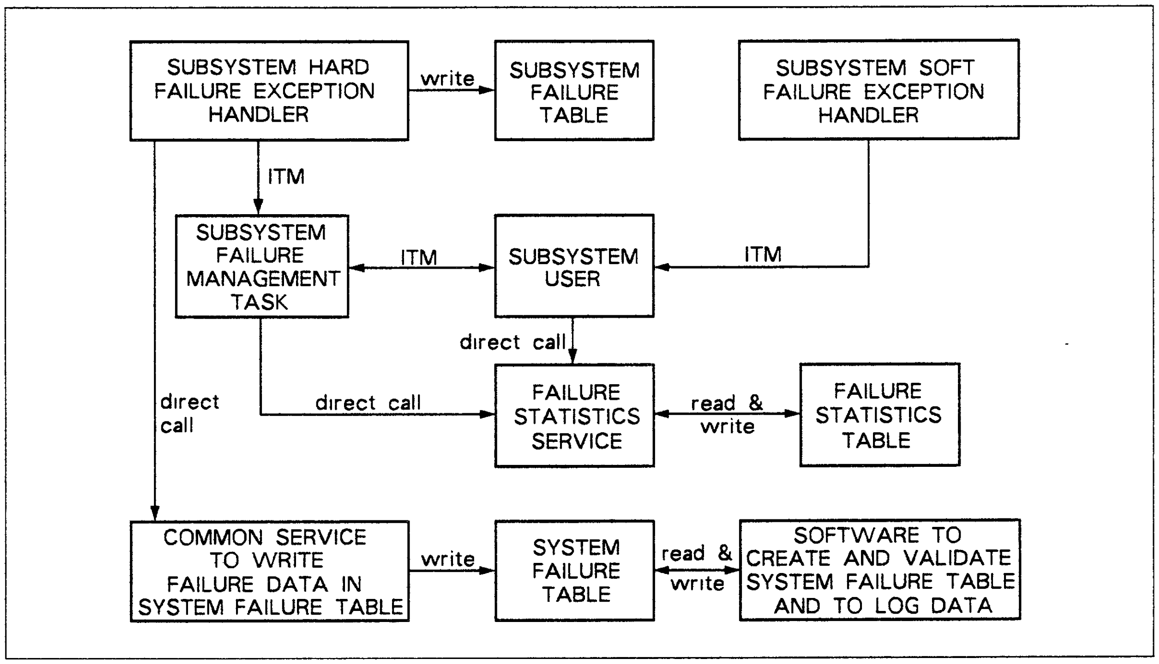

SUBSYSTEM HARD FAILURE EXCEPTION HANDLER
write
SUBSYSTEM FAILURE TABLE
SUBSYSTEM SOFT FAILURE EXCEPTION HANDLER
ITM
SUBSYSTEM FAILURE MANAGEMENT TASK
ITM
SUBSYSTEM USER
ITM
direct call
direct call
direct call
FAILURE STATISTICS SERVICE
read & write
FAILURE STATISTICS TABLE
COMMON SERVICE TO WRITE FAILURE DATA IN SYSTEM FAILURE TABLE
write
SYSTEM FAILURE TABLE
read & write
SOFTWARE TO CREATE AND VALIDATE SYSTEM FAILURE TABLE AND TO LOG DATA

Figure 8-1. General Failure Management Model

MPB Subsystem

The MPB subsystem manages failures on the MPB, SMM and PMM boards. It is also responsible for the management of failures caused by illegal use of hardware by software executing on the MPB board. In addition, it processes environmental warnings such as high temperature and low AC (power) warnings.

MPB software components are expected (but not required) to provide a recovery procedure to be invoked if a failure is detected while the software component is executing. The System Ancestor will invoke a default failure recovery procedure where necessary if one is not provided.

The following software components provide the failure management functions in the MPB subsystem.

MPB bus error handler
SMM error interrupt handler
MPB spurious interrupt handler
MPB level seven interrupt handler
68000 address error handler
68000 exception handler
System Ancestor
System Audit
Executive Error Table

Figure 8-2 shows the different software components in the MPB failure management subsystem and their relationships with each other and with the common failure management software.

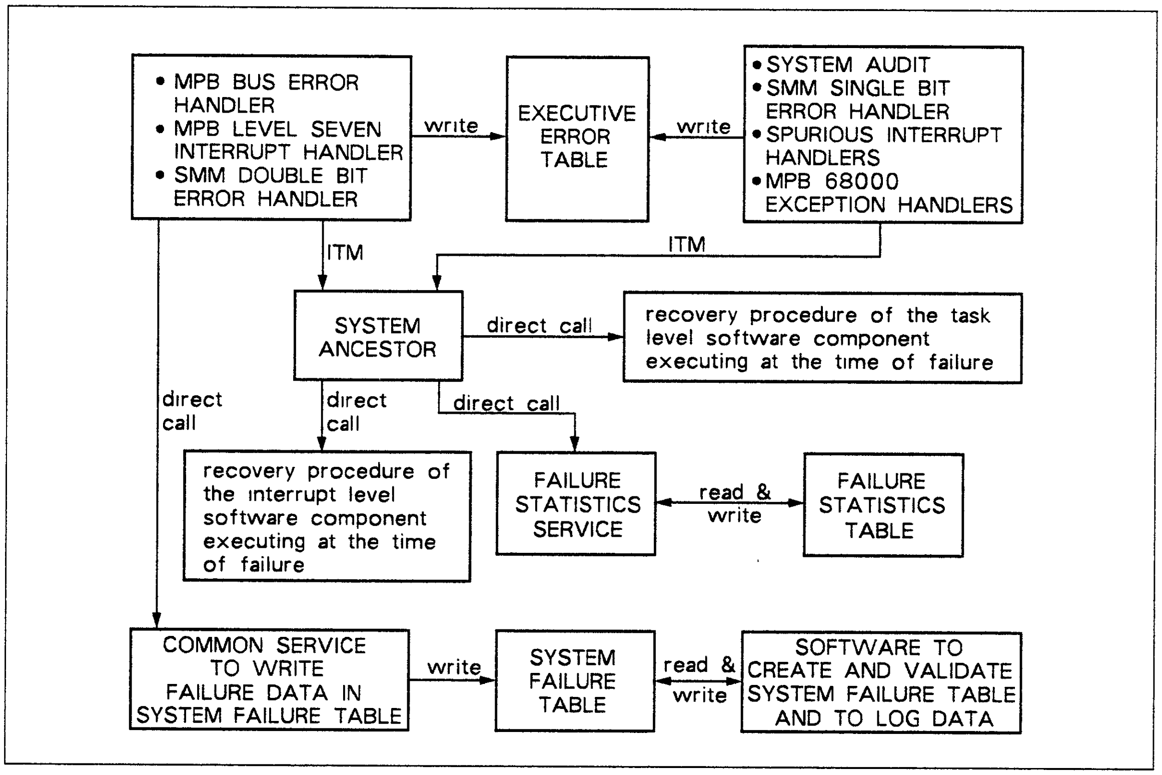

• MPB BUS ERROR HANDLER
• MPB LEVEL SEVEN INTERRUPT HANDLER
• SMM DOUBLE BIT ERROR HANDLER
write
EXECUTIVE ERROR TABLE
write
• SYSTEM AUDIT
• SMM SINGLE BIT ERROR HANDLER
• SPURIOUS INTERRUPT HANDLERS
• MPB 68000 EXCEPTION HANDLERS
ITM
ITM
SYSTEM ANCESTOR
direct call
recovery procedure of the task level software component executing at the time of failure
direct call
direct call
direct call
recovery procedure of the interrupt level software component executing at the time of failure
FAILURE STATISTICS SERVICE
read & write
FAILURE STATISTICS TABLE
COMMON SERVICE TO WRITE FAILURE DATA IN SYSTEM FAILURE TABLE
write
SYSTEM FAILURE TABLE
read & write
SOFTWARE TO CREATE AND VALIDATE SYSTEM FAILURE TABLE AND TO LOG DATA

Figure 8-2. MPB Subsystem Failure Management Model

CIM Subsystem

The CIM subsystem manages failures on the CIM and LIM boards as well as on the LIM ports and the communication lines connected to the LIMs. It is also responsible for the management of failures caused by illegal use of hardware by software executing on the CIM board.

The following software components provide the failure management functions in the CIM subsystem.

CIM bus error handler
CIM spurious interrupt handler
CIM 68000 address error handler
CIM 68000 exception handler
CIM monitor
CIM modem signal monitor
DVM interrupt routines
DVM task
CIM failure table

Figure 8-3 shows the different software components in the CIM failure management subsystem and their relationships with each other and with the common failure management software.

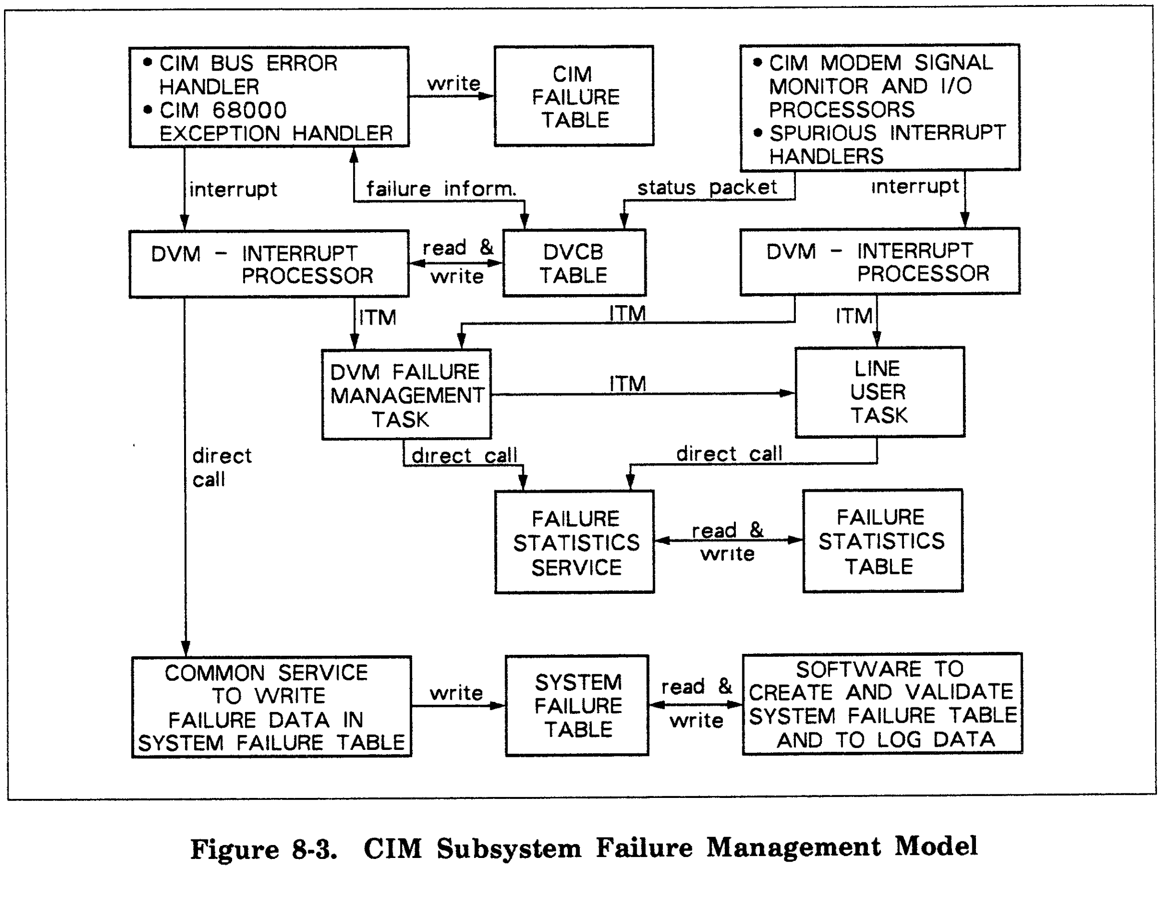

• CIM BUS ERROR HANDLER
• CIM 68000 EXCEPTION HANDLER
write
CIM FAILURE TABLE
• CIM MODEM SIGNAL MONITOR AND I/O PROCESSORS
• SPURIOUS INTERRUPT HANDLERS
interrupt
failure inform.
status packet
interrupt
DVM - INTERRUPT PROCESSOR
read & write
DVCB TABLE
DVM - INTERRUPT PROCESSOR
ITM
ITM
ITM
DVM FAILURE MANAGEMENT TASK
ITM
LINE USER TASK
direct call
direct call
direct call
FAILURE STATISTICS SERVICE
read & write
FAILURE STATISTICS TABLE
COMMON SERVICE TO WRITE FAILURE DATA IN SYSTEM FAILURE TABLE
write
SYSTEM FAILURE TABLE
read & write
SOFTWARE TO CREATE AND VALIDATE SYSTEM FAILURE TABLE AND TO LOG DATA

Figure 8-3. CIM Subsystem Failure Management Model

ESCI Subsystem

The ESCI subsystem manages failures on the ESCI board and any failures of the tranceiver connected to the Ethernet. It also manages failures caused by illegal use of hardware by software executing on the ESCI board. Failure management in the ESCI subsystem is identical to failure management in the CIM subsystem, except in two regards.

The ESCI subsystem includes one additional exception handler called the ESCI M68000 Level Six Interrupt Handler. The reason for this is that the ESCI board has two processors; namely, the M68000 and the Ethernet controller. Only one of these processors can access the internal system bus (ISB) at any point in time. The failures that are normally reported by the bus error are reported by the ESCI bus error interrupt if the ISB was being accessed by the M68000 on the ESCI board. On the other hand, these failures are reported by the Level Six Interrupt Handler on the M68000 processor if they occur while the ethernet controller was accessing the ISB. Functionally, the ESCI bus error handler and the Level Six Interrupt Handler are identical.

The second difference between the CIM and ESCI subsystems is that unlike CIM there is only one user of the ESCI; namely, the ESCI SSR. It is important to note that DVM serves as the failure management task for CIM and ESCI by performing a significant number of failure management functions for these subsystems. DVM does not differentiate between CIM and ESCI in any way.

The following software components provide the failure management functions in the ESCI subsystem.

ESCI bus error handler
ESCI 68000 level six interrupt handler
ESCI spurious interrupt handler
ESCI 68000 address error handler
ESCI 68000 exception handler
ESCI firmware
DVM interrupt processor
DVM failure management task

ESCI failure table

Figure 8-4 shows the different software components in the ESCI subsystem and their relationships with each other and with the common failure management software.

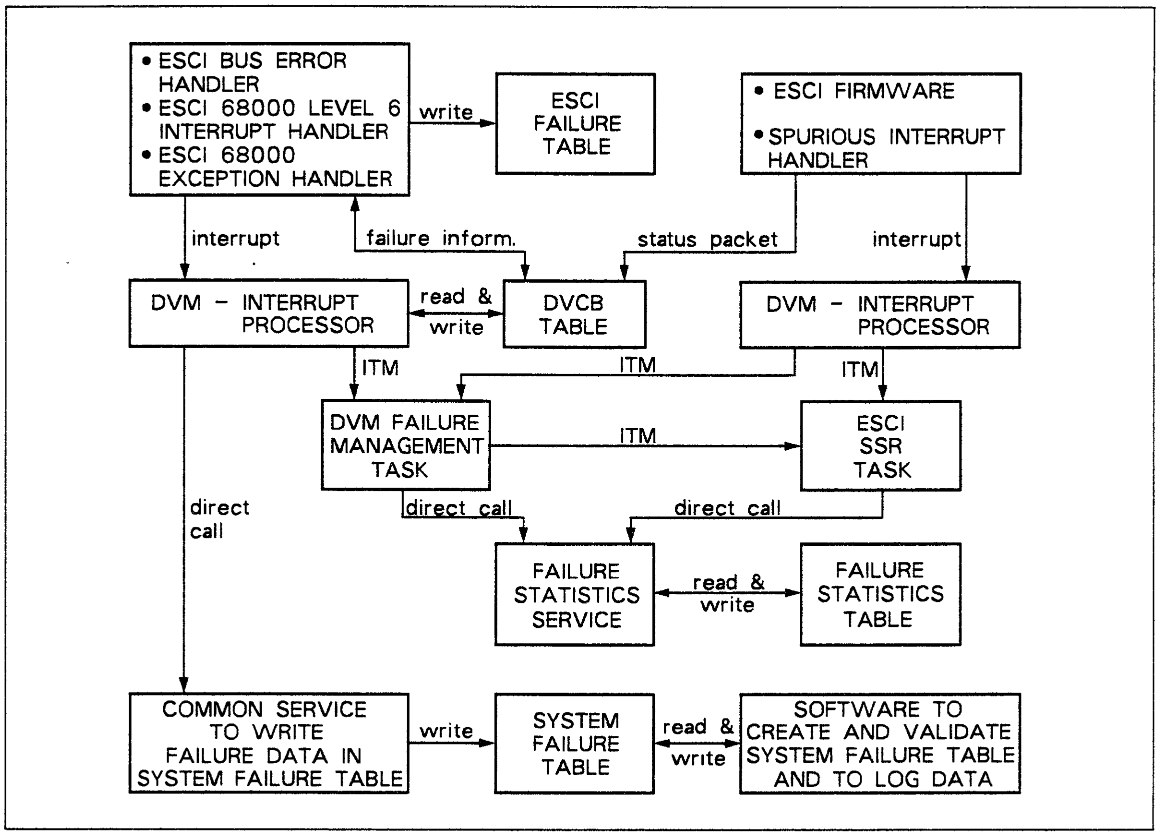

• ESCI BUS ERROR HANDLER
• ESCI 68000 LEVEL 6 INTERRUPT HANDLER
• ESCI 68000 EXCEPTION HANDLER
write
ESCI FAILURE TABLE
• ESCI FIRMWARE
• SPURIOUS INTERRUPT HANDLER
interrupt
failure inform.
status packet
interrupt
DVM - INTERRUPT PROCESSOR
read & write
DVCB TABLE
DVM - INTERRUPT PROCESSOR
ITM
ITM
ITM
DVM FAILURE MANAGEMENT TASK
ITM
ESCI SSR TASK
direct call
direct call
direct call
FAILURE STATISTICS SERVICE
read & write
FAILURE STATISTICS TABLE
COMMON SERVICE TO WRITE FAILURE DATA IN SYSTEM FAILURE TABLE
write
SYSTEM FAILURE TABLE
read & write
SOFTWARE TO CREATE AND VALIDATE SYSTEM FAILURE TABLE AND TO LOG DATA

Figure 8-4. ESCI Subsystem Failure Management Model

MCI Subsystem

The MCI subsystem manages failures on the MCI board and the channel interface connected to it.

The following software components provide the failure management functions in the MCI sybsystem.

MCI driver error interrupt handler
MCI SSR

Figure 8-5 shows the different software components in the MCI subsystem and their relationships with each other and with the common failure management software.

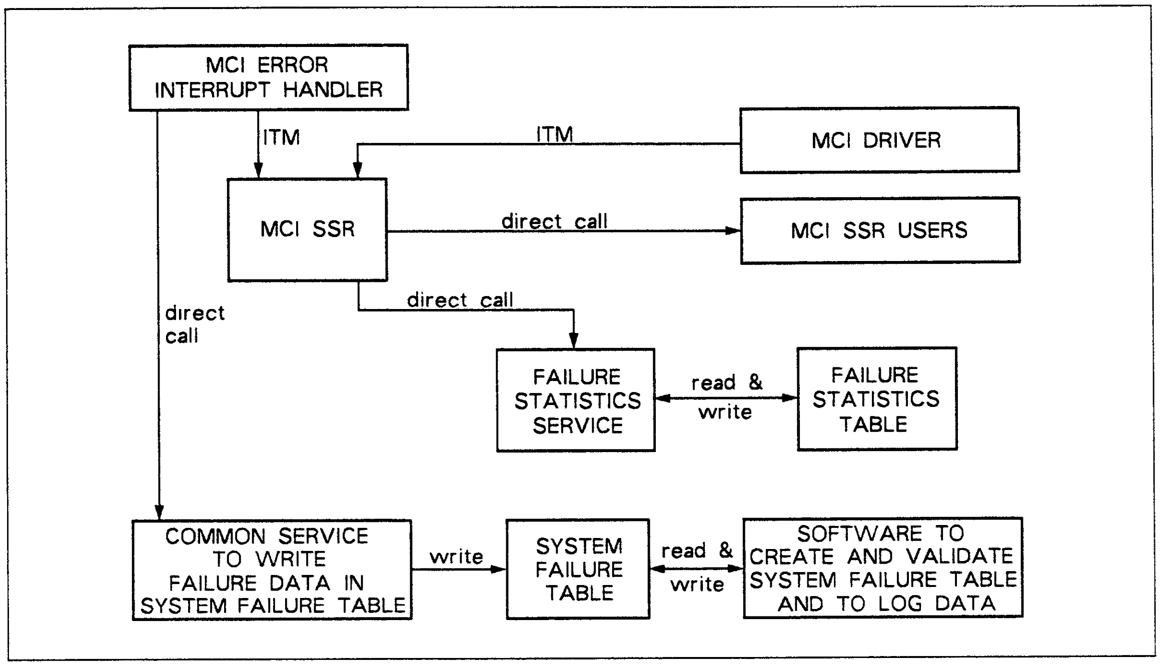

MCI ERROR INTERRUPT HANDLER
ITM
ITM
MCI DRIVER
MCI SSR
direct call
MCI SSR USERS
direct call
direct call
FAILURE STATISTICS SERVICE
read & write
FAILURE STATISTICS TABLE
COMMON SERVICE TO WRITE FAILURE DATA IN SYSTEM FAILURE TABLE
write
SYSTEM FAILURE TABLE
read & write
SOFTWARE TO CREATE AND VALIDATE SYSTEM FAILURE TABLE AND TO LOG DATA

Figure 8-5. MCI Subsystem Failure Management Model

Logging of Failure Information

It is important that all failure information be logged. However, there are two important considerations in defining a process to log failure information.

The first concern is about the ability of CDCNET software to generate and transmit a log message after having encountered a failure. The failure may have corrupted key data structures or program space, thus inhibiting the logging process. The second concern is about generating too many log messages. For example, certain failures are expected to occur and it does not make sense to log each occurrence of such failures.

Unfortunately, hard failures–which generally need to be logged–can potentially inhibit their own logging. On the other hand, soft failures can generally be logged, but don't always need to be. The following two sections address the logging of hard and soft failures and describes how CDCNET software takes care of the two concerns just mentioned.

Logging of Hard Failures

CDCNET software attempts to log each occurrence of a hard failure. If failure recovery does not require resetting the DI, the appropriate subsystem failure management task generates a log message to report the failure. This log message identifies the specific failure and includes all relevant failure information, such as the contents of the corresponding hardware status registers.

If failure recovery requires resetting the DI, no attempt is made to generate a log message prior to reset. In this case the failure information is saved both in the subsystem failure table and the system failure table. Then, if the system is successfully reloaded and restarted, both the system and subsystem initialization software extract the failure information from the failure tables and report it with a log message.

The key to this process is that the subsystem failure tables not get destroyed during the DI reload. These tables, with the exception of the Executive (MPB) Error Table, are not protected in a power-on reset and so are initialized to a clear state.

It is also important that the failure information in the subsystem failure tables not be written over until it has been logged.

WARNING

One implication of this is that if a second failure occurs before information about the first failure has been logged, then the information about the second failure is not saved and may never get logged.

Logging of Soft Failures

The concern with soft failures is to control the number of times they get logged. This is done by defining a threshold for each soft failure and logging the failure only if this threshold is exceeded. CDCNET failure management software defines two logging thresholds, **absolute** and **timed**.

If an absolute threshold is used, all occurrences of a failure are counted and a log message is generated when the total count of failures exceeds a fixed number. This fixed number is known as the absolute threshold. The count of failures is initialized to zero every time a log message is generated to report the failure.

If a timed threshold is used, the number of failures over a fixed interval is counted. If this count exceeds a specified number within the specified interval, the failure is logged. In this case, the number of failures, together with the fixed time interval, define the timed threshold.

The Device Manager (DVM) 9

The Device Manager (DVM) 9

The Device Manager (DVM) is a set of routines responsible for the interface between CDNA's physical and link layers (layers 1 and 2, respectively). By controlling access to the "intelligent" peripheral boards (IPs), the DVM provides its users with a single, common mechanism for reaching the physical I/O layer, and ensures that no two users will be assigned to the same port. DVM's users include stream service routines (SSRs) and TIPs.

This chapter describes each of the general services provided by the DVM, details the order of events in its operation, and describes its major data structures. Each of the DVM procedures mentioned here is fully documented in chapter 10 of this manual.

Device Control Services

These services coordinate the allocation, initialization, and release of control blocks for IPs and their individual ports. They also control the loading of controlware to the boards. Device control procedures are:

Procedure	Description
START_DEVICE_SERVICE	Starts DVM services for the specified device
STOP_DEVICE_SERVICE	Stops execution of a peripheral device service
START_PORT_SERVICE	Establishes service for a specific port
STOP_PORT_SERVICE	Stops service for a given port
RESTART_PORT_SERVICE	Restarts service for a specific port
CHANGE_DVM_TASK_ID	Changes ownership of a specific port to a new task

Status Request Services

These services provide the status of specific hardware to the user. Status request procedures are:

Procedure	Description
GET_CIM_NUMBER	Determines a CIM slot number from the user-supplied LIM and PORT number
DEVICE_STARTED	Determines whether or not the peripheral board has been started
DUMP_IP_MEMORY	Dumps the peripheral board's local RAM
GET_CIM_BOOT_SOURCE	Returns the LIM and port number over which the DI was booted

Data Transfer Services

Data transfer services provide DVM users a means of passing commands to the intelligent peripheral boards. The procedure is called QUEUE_IP_COMMAND.

Diagnostic Services

These services allow MPB-resident diagnostic code to alter the peripheral board environment and to run on-line diagnostic tests. Diagnostic code can also return the peripheral board to its previous environment.

DVM's diagnostic services procedures are:

Procedure	Description
SEND_CR0_TO_IP	Sends command register 0 to an IP
READ_SR0_FROM_IP	Reads status register 0 from an IP
QUEUE_IP_COMMAND	Queues a command to an IP
CHKSUM_IP_COMMAND_PACKET	Performs a checksum operation on a restart packet to be sent to an IP
CHANGE_DVM_INTERRUPT	Changes the interrupt procedure used to handle interrupts from an IP
RESTORE_DVM_INTERRUPT	Restores the original interrupt routine used before the CHANGE_DVM_INTERRUPT procedure executed

NOTE

DVM diagnostic services procedures are not intended for general use.

DVM Order of Events

DVM users can make direct calls to DVM that will execute from within the caller's stack and under control of the caller's task. DVM queues commands to the IP for processing and supplies information to its users by an ITM. These ITMs are asynchronous with respect to calls made by the user to DVM. The process is illustrated in figure 9-1 and described below.

- DVM communicates with a peripheral board using a commonly referenced area of SMM memory, the Device Control Block (DVCB). The DVCB is allocated by DVM and its address is made known to the IP as part of the peripheral board load process. When a user issues a command to an IP through DVM, the command packet is sent to the command queue area of the DVCB. It is up to the peripheral board, then, to check the DVCB to determine if there are any outstanding commands to be processed.
- When the peripheral board completes processing of the command, it will build a response and enqueue that response to the status queue area of the DVCB. Then the peripheral board issues an interrupt to the DVM. DVM will process the status queue and return the appropriate status to the caller using an ITM.
- If the IP receives data from an active port, it will package that data into a status packet and enqueue the packet to the status area of the DVCB. The IP will then issue an interrupt to the DVM. DVM processes this status packet by building an ITM and routing it to the appropriate user.

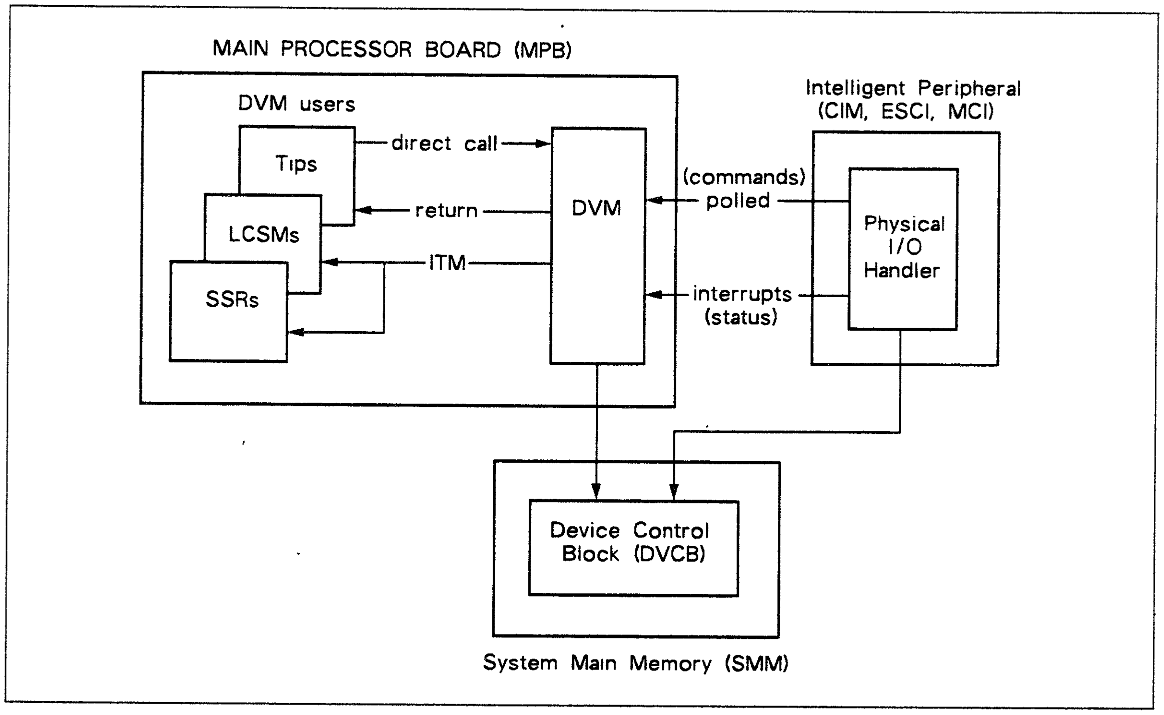


Figure 9-1. DVM Relationships

DVM Major Data Structures

Device Control Block (DVCB)

The DVCB is a data structure that retains information about an interface's IPs. Each peripheral has its own DVCB in SMM. DVM uses DVCBs to control access between tasks residing on the MPB and tasks residing on the peripheral boards.

The structure of the DVCB is illustrated in figure 9-2.

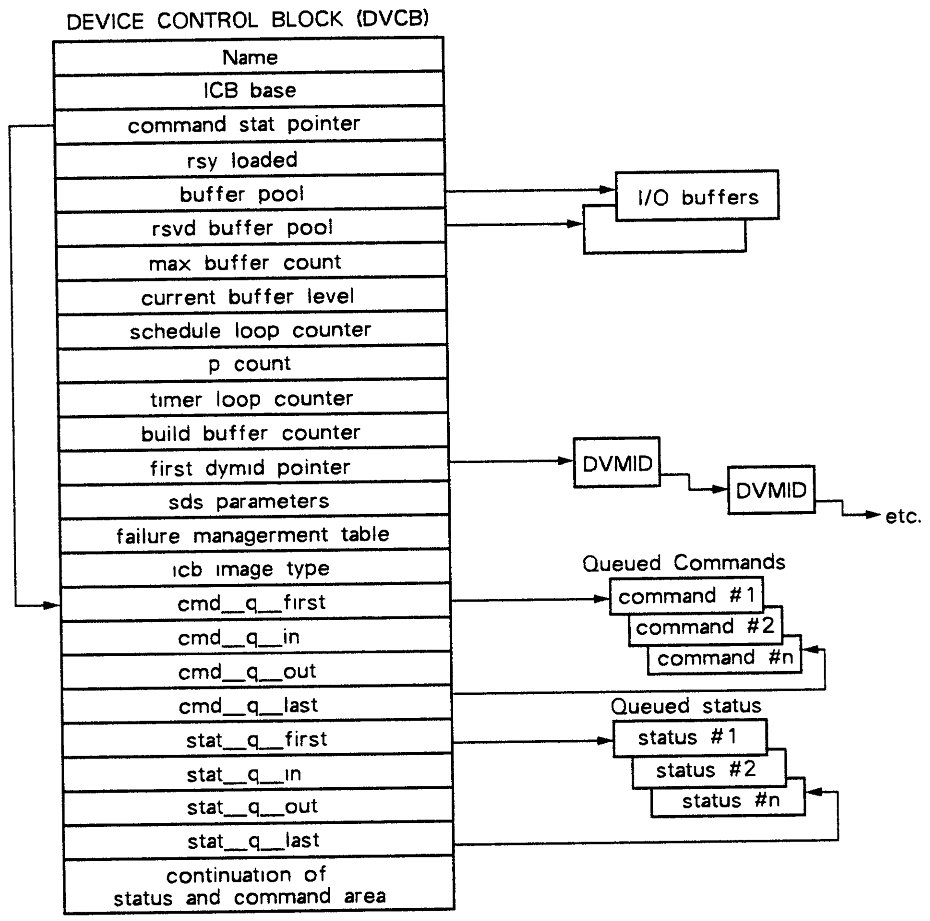


Figure 9-2. Device Control Block

The DVCB contains:

- The addresses of the mapped control and status registers of the associated peripheral.
- A pair of addresses of empty buffer chains to be used by the peripheral as needed to receive data.
- A set of addresses and indexes that control the sequences of status and command queues.
- The address of the first device identifier (referred to as DVMID) that may be assigned to the DVCB.

The DVM maintains an array of DVCB addresses to access the peripherals.

Device Identifier (DVMID)

The DVID structure contains information about a specific port controlled by the IP and also information required by DVM to communicate with the corresponding TIP or SSR.

The structure of a DVID is illustrated in figure 9-3.

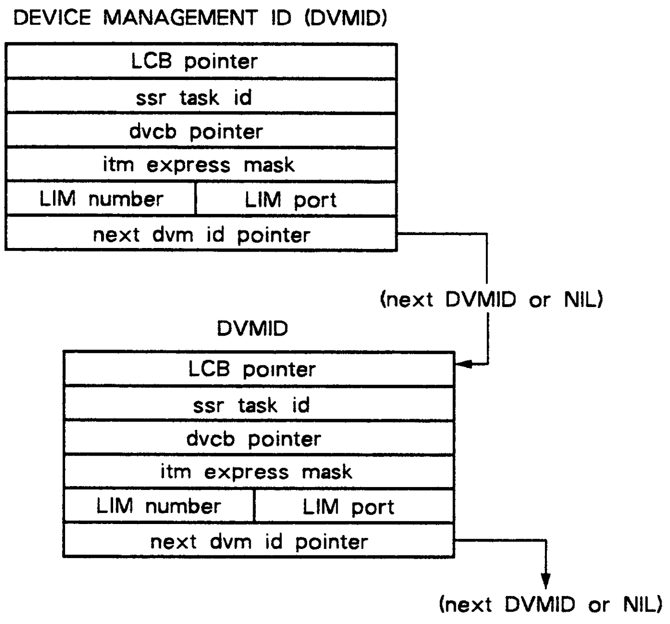


Figure 9-3. Device Identifier

The DVID contains:

- The TASKID of the user to which all status responses are to be routed for this port.
- Address of the owning DVCB.
- A mask specifying which status responses are to be routed using the express message queue and which are to be routed using the normal message queue.
- The physical LIM and PORT numbers assigned to this DVMID (for CIM only).
- The address of the next DVID linked to the DVCB.

DVM Command Packets and Status ITMs

DVM uses two circular queues to support interfaces to the IPs. The command queue is used to pass information from the DVM user to the IP. The status queue is used to pass status packets from the IP back to the DVM user. The IP interrupts the MPB to inform DVM that a status packet should be returned to the DVM user.

Command packets are passed from DVM users to the IPs through the DVM QUEUE_ IP_COMMAND procedure. The IPs use DVM services to return status packets to their users; these are in the form of ITMs. The formats of the command packets and the status ITMs for CIM and ESCI are defined in appendix B.

Procedures and Functions 10

Procedures and Functions 10

This chapter documents the common routines provided by CDCNET's base system software. These procedures and functions may be called on to perform the basic services described in the early chapters of this manual.

Conventions

Descriptions of the procedures and functions adhere to the following conventions:

- The procedure or function name appears at the top of the page in full capital letters.
- A short description of the purpose of the procedure or function follows. This tells you how the procedure is used, not how it is processed.
- The name of the common deck (comdeck) where the XREF procedure is to be found is given next. This must be specified in the procedure's deck reference, as in the following example (using comdeck CMXSISA):

  ```
  (*callc CMXSISA)
  ```

- The format of the call to the procedure is given, including any formal parameters.
- Input parameters are described.
- Output parameters are described.
- Finally, any special remarks concerning the procedure or function are made, including cautions or warnings.

Logical Groups

For easy reference, the procedures and functions have been grouped below by software component.

DI Debugger Interface

DI_DEBUG
DI_DEBUG_INIT

DVM Interface

CHANGE_DVM_INTERRUPT
CHANGE_DVM_TASK_ID
CHKSUM_IP_COMMAND_PACKET
DEVICE_STARTED
DUMP_IP_MEMORY
GET_CIM_BOOT_SOURCE
GET_CIM_NUMBER
QUEUE_IP_COMMAND
READ_SR0_FROM_IP
RESTART_PORT_SERVICE
RESTORE_DVM_INTERRUPT
SEND_CR0_TO_IP
START_DEVICE_SERVICE
START_PORT_SERVICE
STOP_DEVICE_SERVICE
STOP_PORT_SERVICE

General Purpose

ABORT_SYSTEM
ABS, MAX, MIN
CONVERT_INTEGER_TO_POINTER
CONVERT_POINTER_TO_INTEGER
DEAD_STOP
FIELD_SIZE
I_COMPARE
I_COMPARE_COLLATED
I_SCAN
I_TRANSLATE
MDU_TO_ASCII
NAME_MATCH
UNUSED_STACK_

Hardware Device Interfaces

GET_CARD_TYPE_AND_ADDRESS
RELEASE_HARDWARE_DEVICE
REQUEST_DIAGNOSTIC_ENTRY
REQUEST_HARDWARE_DEVICE

Memory Interfaces

CLEAR_ALLOCATE
CLEAR_MEMORY
CLEAR_WRITE_PROTECT
GET_MEMORY
GET_MPB_EXTENT
GET_PMM_EXTENT
MODIFY_WRITE_PROTECT_BYTE
MODIFY_WRITE_PROTECT_LONG_WORD
MODIFY_WRITE_PROTECT_SHORT_WORD
SET_BUFFER_CHAIN_OWNER
SET_MEMORY_OWNER
SET_WRITE_PROTECT

Message Management

APPEND
ASSEMBLE
BROADCAST
BUILD_HEADER_IN_PLACE
COPY
FG_TRIM
FIRST_BYTE_ADDRESS
FRAGMENT
GEN_DATA_FIELD
GEN_TEMPLATE_ID
GET_DATA_FIELD
GET_FIRST_BYTE
GET_LAST_BYTE
GET_LONG_BUFFERS
GET_MESSAGE_LENGTH
GET_SHORT_BUFFERS
GET_SIZE_N_ADDR
MESSAGE_DEQUEUE
MESSAGE_ENQUEUE
M_RELEASE
PCOPY
PREFIX
RELEASE_MESSAGE
STRIP
STRIP_IN_PLACE
SUBFIELD
TRANSLATE_MESSAGE
TRIM

Online Loader Interface

CHECKSUM_NEXT_MODULE
DECREMENT_MODULE_USE_COUNT
INCREMENT_MODULE_USE_COUNT
LOAD_ABS_MODULE_AND_DELAY
LOAD_ABS_MODULE_AND_PROCEED
LOAD_CMD_PROCESSOR_AND_DELAY
LOAD_CMD_PROCESSOR_AND_PROCEED
LOAD_ENTRY_POINT_AND_DELAY
LOAD_ENTRY_POINT_AND_PROCEED
START_NAMED_TASK_AND_DELAY
START_NAMED_TASK_AND_PROCEED
VALIDATE_SECTION_ADDRESS

Queue Management

MESSAGE_DEQUEUE
MESSAGE_ENQUEUE

Statistics Management

OPEN_STATISTICS_SAP
CLOSE_STATISTICS_SAP
FORCE_STATISTICS_REPORTING
BUILD_STATISTICS_MSG_HEADER
INITIALIZE_STATISTICS_RECORD

Status Management

CLOSE_STATUS_SAP
GET_NEXT_STATUS_SAP
GET_STATUS_RECORD
GET_STATUS_SAP
OPEN_STATUS_SAP
PUT_STATUS_RECORD

System Ancestor Interface

DUMP_CLOSE
DUMP_WRITE
RESET_DI
RESET_RECOVERY_PROCEDURE
SET_RECOVERY_PROCEDURE
START_DUMP
START_SYSTEM_TASK

Task Management

ABORT_TASK
DELAY_PROCESSING
GET_EXPRESS
GET_MSG
MAYBE_TASK
NEW_INTERRUPT
NEW_PRIORITY
NOPREMPT
OKPREMPT
RESTORE_TASK
SEND_EXPRESS
SEND_NORMAL
SIGNAL(n)/ACQUIRE(n)
START_TASK
STOP_TASK
SUSPEND
WAIT
WAKE_UP
YIELD

Timer Services

CALL_AFTER_INTERVAL
CALL_AT_TIME
CALL_PERIODIC
CANCEL_TIMER
CHANGE_TIMER_OWNER
PMP_GET_DATE
PMP_GET_TIME
READ_BCD_CLOCK
READ_CLOCK
SET_BCD_CLOCK
TIME

Tree (Table) Management

FIND
FIND_FIRST
FIND_FREE_NODE
FIND_NEXT
FIRST_NODE
GROW
INIT_ROOT
PICK
SFIND
SFIND_FIRST
SFIND_NEXT
SFIND_WILD_CARDS
SGROW
SPICK
VISIT_ALL_NODES

Base system software procedures and functions are described in alphabetical order on the remaining pages of this chapter.

ABORT_SYSTEM

This procedure brings the system to a halt and displays the specified message text at the MPB-connected terminal (if present).

Comdeck CSXABRT

Format **ABORT_SYSTEM (halt_code, message_ptr)**

Input **halt_code: integer**

This parameter indicates what brought the system down. If halt_code is a valid reset (see common deck SIDRC), then a call is made to RESET_DI. Otherwise, a call is made to DEAD_STOP, where a default halt_code is used.

message_ptr: ^string (* <= dbc$single_line)

This parameter is a pointer to an adaptable string containing message text about why the abort was necessary.

Output None.

ABORT_TASK

This procedure signals that a task is undergoing a software failure, and that recovery is required. The indicated task is checked to see if it has a parent task. If it does not, it is stopped with STOP_TASK and the entire system is brought to a halt. If the indicated task does have a parent task, the task is suspended and the parent task is notified with an ITM.

ABORT_TASK is intended to be a response to an illogical software condition, invoking action from the parent to recover or restart the aborted task.

Comdeck	**CMXMTSK**
Format	**ABORT_TASK (abort_code, task_id, status)**
Input	**abort_code: integer** This parameter indicates the reason the caller wants the task aborted. **task_id: task_ptr** This parameter indicates the address of the task to be aborted.
Output	**status: boolean** This parameter indicates whether or not the task was aborted.

ABS, MAX, MIN

These numeric functions are quite predictable. They perform the following operations:

ABS	Returns the absolute value of an integer.
MAX	Returns the greater of two integer values.
MIN	Returns the lesser of two integer values.

Comdeck CMXPMMA

Format

```
value:= ABS (a)
value:= MAX (a, b)
value:= MIN (a, b)
```

Input

a: integer

This parameter may be any integer value.

b: integer

This parameter may be any integer value.

Output

value: integer

This parameter returns the result of the numeric operation being performed.

APPEND

This procedure appends a trailer to a message. The message is checked for use by multiple data streams. If any portion is multiply used, that portion is logically copied.

Comdeck **CMXPAPP**

Format **APPEND (size_of_trailer, addr_of_trailer, message_pointer, threshold, allocation_type, result_status)**

Input

size_of_trailer: non_empty_message_size

This parameter indicates the length of the trailer in bytes. The allowable range is 1 through 65535 bytes.

addr_of_trailer: ^cell

This parameter indicates the address of the trailer.

message_pointer: buf_ptr

This parameter indicates the address of the message to which the trailer will be appended.

threshold: threshold_size

This parameter indicates the threshold for buffer acquisition.

allocation_type: pref_type

This parameter indicates how the call is to be performed. See the description of pref_type (under BUFFER in appendix B) for options.

Output

message_pointer: buf_ptr

This parameter returns the address of the message to which the trailer has been appended. This is different from input only if a new data buffer must be allocated to contain a longer message.

result_status: boolean

This parameter indicates whether or not the trailer was appended to the message.

Remarks In case of a logical copy operation, the message_pointer returned may be different than the message_pointer supplied.

ASSEMBLE

This procedure assembles two message fragments into a single message by attaching the second fragment to the end of the first.

Comdeck **CMXPASS**

Format **ASSEMBLE (fragment_1, fragment_2, threshold)**

Input

fragment_1: buf_ptr

This parameter indicates the address of the first message fragment.

fragment_2: buf_ptr

This parameter indicates the address of the second message fragment.

threshold: threshold_size

This parameter indicates the threshold for buffer acquisition.

Output

fragment_1: buf_ptr

This parameter returns the address of the first fragment, which contains the assembled message on output. If any of fragment_1 is multiply used, the portion so used is logically copied and the output parameter will differ from input.

BROADCAST

This procedure prepares a message for multiple use. This is accomplished by updating the user count of the first descriptor of the message to show the increased number of data streams that must logically release the message before it may be physically released.

Comdeck **CMIPBRO**

Format **BROADCAST (message, number_of_new_data_streams)**

Input **message: buf_ptr**

This parameter indicates the address of the message to be broadcast.

number_of_new_data_streams: 1 .. 32767

This parameter indicates the number of data streams to be added to the user count of the message being prepared for broadcast.

This parameter value is incremented by 1 when the user wants to keep a copy of the message. The call to BROADCAST is often made with the count equal to one since BROADCAST simply adds to the logical number of copies of a message already in place.

Output None.

BUILD_HEADER_IN_PLACE

This procedure gets a buffer or descriptor as needed, creates space in the message to hold the specified header, and returns both the new message address and the address of the header structure. The current first buffer is used if the header will fit and starts on an even byte.

Comdeck **CMXPBLD**

Format **BUILD_HEADER_IN_PLACE (length, addr, message, threshold, allocation_type, success)**

Input

length: non_empty_buffer

This parameter indicates the number of bytes reserved for the header. Allowable range is 1 .. 1024.

addr: ^cell

This parameter indicates the pointer to the address of the header.

message: buf_ptr

This parameter indicates the address of a pointer to the message.

threshold: threshold_size

This parameter indicates the buffer allocation threshold.

allocation_type: pref_type

This parameter indicates how the call is to be performed. See the description of pref_type (under BUFFER in appendix B) for options.

Output

addr: ^cell

This parameter returns the pointer to the address of the header. This will differ from input only if a new buffer has been allocated. A new buffer is allocated if the header does not fit into the original buffer.

message: buf_ptr

This parameter returns the address of a pointer to the message. This will differ from input only if a new buffer has been allocated. A new buffer is allocated if the new message will not fit into the original buffer.

success: boolean

This parameter indicates whether or not the operation was successful.

Remarks The size of a header that is allocated may not exceed the size of the data space of a data buffer; larger headers must be generated using PREFIX.

BUILD_STATISTICS_MSG_HDR

This procedure initializes and puts TIME and REASON fields into a log message.

Comdeck	**SMXSAPM**
Format	**BUILD_STATISTICS_MSG_HDR (sds_header_ptr, time, reason, statistics_log_msg)**
Input	**sds_header_ptr: ^sds_header** This parameter indicates the pointer to a chain of one or more sds_ headers. **time: report_time_type** This parameter indicates the TIME field to be used in the header being built. **reason: statistics_reason_type** This parameter indicates the REASON field to be used in the header being built.
Output	**statistics_log_msg: buf_ptr** This parameter returns the address of the generated statistics log message. If NIL, invalid calling parameters were specified.

CALL_AFTER_INTERVAL, FG_AFTER_INTERVAL

An ITM is enqueued to the timer task requesting the execution of a subroutine after a specified time interval has elapsed. These calls have the following effects:

CALL_AFTER_INTERVAL	Enqueues timer request.
FG_AFTER_INTERVAL	For interrupt routines only; enqueues timer request.

Comdeck CMXMTIM

Format **CALL_AFTER_INTERVAL (interval, parameter, timer_routine, timer_request_identifier)**
FG_AFTER_INTERVAL (interval, parameter, timer_routine, timer_request_identifier)

Input **interval: milliseconds**

This parameter indicates the number of milliseconds in the interval before the subroutine is executed. Milliseconds can be calculated by calling the TIME procedure.

parameter: ^cell

This parameter indicates the address of the user-defined parameters for the procedure to be executed.

timer_routine: ^procedure (parameter: ^cell)

This parameter indicates the subroutine to be executed.

Output **timer_request_identifier: ^timer**

This parameter returns the timer entry address if the procedure executed successfully.

CALL_AT_TIME, FG_AT_TIME

An ITM is enqueued to the timer task requesting the execution of a subroutine at a given time. These calls have the following effects:

CALL_AT_TIME	Enqueues timer request.
FG_AT_TIME	For interrupt routines only; enqueues timer request.

Comdeck **CMXMTIM**

Format **CALL_AT_TIME (time_of_day, parameter, timer_routine, timer_request_identifier)**
FG_AT_TIME (time_of_day, parameter, timer_routine, timer_request_identifier)

Input **time_of_day: milliseconds**

This parameter indicates the execution time in milliseconds, with midnight equal to zero.

parameter: ^cell

This parameter indicates the address of the user-defined parameters for the procedure to be executed.

timer_routine: ^procedure (parameter: ^cell)

This parameter indicates the subroutine to be executed.

Output **timer_request_identifier: ^timer**

This parameter returns the timer entry address if the procedure executed successfully.

Remarks If the requested time has passed (for example, it is now 12:05 a.m. and an execution time of midnight is requested), the request is understood to expire on the next day. See also: CANCEL_TIMER Request.

CALL_PERIODIC, FG_PERIODIC

An ITM is enqueued to the timer task requesting periodic activation of a subroutine until the request is cancelled or until the requesting taskid is no longer valid. Requests from interrupt routines must actually be cancelled.

These calls have the following effects:

CALL_PERIODIC	Enqueues timer request.
FG_PERIODIC	For interrupt routines only; enqueues timer request.

Comdeck CMXMTIM

Format **CALL_PERIODIC (first_expiration, interval, parameter, timer_routine, timer_request_identifier)**
FG_PERIODIC (first_expiration, interval, parameter, timer_routine, timer_request_identifier)

Input

first_expiration: milliseconds

This parameter indicates the starting time in milliseconds for the first call to be made. A value of NIL implies the current time.

interval: milliseconds

This parameter indicates the number of milliseconds in the interval between subroutine executions.

parameter: ^cell

This parameter indicates the address of the user-defined parameters for the procedure to be executed.

timer_routine: ^procedure (parameter: ^cell)

This parameter indicates the subroutine to be executed.

Output

timer_request_identifier: ^timer

This parameter returns the timer entry address if the procedure executed successfully.

Remarks If the requested time has passed (for example, it is now 12:05 a.m. and an execution time of midnight is requested), the request is understood to expire on the next day. See also: CANCEL_TIMER request.

CANCEL_TIMER, FG_CANCEL_TIMER

A previously requested timing function (CALL_AFTER_INTERVAL, CALL_AT_TIME or CALL_PERIODIC) is cancelled. These calls have the following effects:

CANCEL_TIMER	Timer request is cancelled.
FG_CANCEL_TIMER	For interrupt routines only; timer request is cancelled.

Comdeck **CMXMTIM**

Format **CANCEL_TIMER (timer_id, parameter, status)**
FG_CANCEL_TIMER (timer_id, parameter, status)

Input **timer_id: ^timer**

This parameter indicates the timer entry associated with the timing procedure to be cancelled. The timer_id must be the value returned by the timing function call. If an invalid value is specified, the DI will be reset. If NIL, FALSE will be returned in status.

Output **timer_id: NIL**

The timer_id is returned as NIL so that it will not be used again.

parameter: ^cell

This parameter returns the address of the user-defined parameters for the procedure whose timing function is being cancelled.

status: boolean

This parameter indicates whether or not the timer was cancelled successfully.

CHANGE_DVM_INTERRUPT

DVM Diagnostic Services use this procedure to change the interrupt procedure used to handle interrupts from a peripheral.

Comdeck **DMXDIAG**

Format **CHANGE_DVM_INTERRUPT (new_proc, ip_number, owner, status)**

Input

new_proc

This parameter indicates the address of the new interrupt procedure.

ip_number: 0..upper_ip_num

This parameter indicates the slot number of the IP.

Output

owner: task_ptr

This parameter returns the address of the original DVM user.

status: 0..0ffff(16)

This parameter returns the status of CHANGE_DVM_INTERRUPT. The status codes are:

0: command accepted
1: invalid IP slot number
2: IP board is active
4: bus error received on ICB access

CHANGE_DVM_TASK_ID

This procedure is used to change the ownership of a specific port to that of a new task by giving control of the specified DVMID to a new DVM user.

Comdeck **DMXXPS**

Format **CHANGE_DVM_TASK_ID (dvm_id, new_task_id, status)**

Input **dvm_id: ^cell**

This parameter indicates the address of the specified port control block (DVMID).

new_task_id: task_ptr

This parameter indicates the address of the task that is the new owner of the specified port.

Output **status: boolean**

This parameter indicates whether or not ownership has been successfully changed.

CHANGE_TIMER_OWNER

The requested timing function will have its allocating task id changed to the indicated task. If the indicated task equals NIL, the currently running task will be used as the new allocating task for the indicated timer.

Comdeck **CMXMTIM**

Format **CHANGE_TIMER_OWNER (timer_id, task_id, status)**

Input **timer_id: ^timer**

This parameter indicates the timer entry associated with the timing procedure whose ownership is to be transferred.

task_id: task_ptr

This parameter indicates the new allocating task for the indicated timing procedure.

Output **status: boolean**

This parameter indicates whether or not the procedure executed properly. A false status is returned if the timer_id or the task_id is invalid.

CHECKSUM_NEXT_MODULE

Successively validates the section checksums of the next module. The procedure executes as follows:

- The load_identifier parameter is initially passed in as NIL in order to start with the first module.
- The checksums for sections of the current module are successively calculated. These checksums are compared with the sums in the module header. If they are equal, then checksum_valid is set to TRUE; otherwise, it is returned FALSE.
- When the last module has been checksummed, the parameter next_module_found is returned with a value of FALSE. The module use count of the previous module is decremented and the current module is incremented.

Comdeck	**DLXCKNM**
Format	**CHECKSUM_NEXT_MODULE (load_identifier, next_module_found, checksum_valid)**
Input	**load_identifier: dlt$load_id_ptr** This parameter indicates a record that specifies the module characteristics. It is initially set to NIL.
Output	**load_identifier: dlt$load_id_ptr** This parameter is used to pass to the CHECKSUM_NEXT_MODULE call while looping. **next_module_found: boolean** This parameter returns TRUE until the end of the module header linked list is found. The parameter then returns FALSE. **checksum_valid: boolean** This parameter indicates whether the section checksum returned is equal to the section checksum from the time of initial load.

CHKSUM_IP_CMD_PKT

DVM diagnostic services use this procedure to perform a checksum operation on a restart packet to be sent to a peripheral board.

Comdeck	**DMXDIAG**
Format	**CHKSUM_IP_CMD_PKT (packet_ptr)**
Input	**packet_ptr: ^ip_cmd_pkt_type** This parameter indicates the address of the packet that is to have a checksum operation performed on it.
Output	None.

CLEAR_ALLOCATE

This procedure allocates memory from the system heap and clears it to zero. If the memory cannot be allocated at the time of the call, the caller will yield the CPU until the memory request can be satisfied.

Comdeck	**CMXCLAL**
Format	**address:= CLEAR_ALLOCATE (memory_bytes)**
Input	**memory_bytes: 1 .. 32766** This parameter indicates the number of bytes to be allocated.
Output	**address: ^cell** This parameter returns the address of the memory allocated.
Remarks	The allocated memory will always be an even number of bytes and start at an even byte boundary. Use of this procedure is not recommended. Instead, the recommended technique is to allocate the record with the CYBIL allocate statement and then initialize the record by copying a pre-initialized static template.

CLEAR_MEMORY

This procedure clears a specified number of memory bytes.

Comdeck	CMXCLAL
Format	CLEAR_MEMORY (even_start_address, memory_bytes)
Input	**even_start_address: ^cell** This parameter indicates the starting address of memory to be cleared. It is assumed to be an even byte address. **memory_bytes: 0 .. 32766** This parameter indicates the length of memory to be cleared. It is assumed to be an even number of bytes.
Output	None.

CLEAR_WRITE_PROTECT

This procedure clears the write protect flag.

The write protect feature prevents user tasks from writing into MPB RAM. This adds a hardware level of security to prevent loss of the system_id and other configuration information. CLEAR_WRITE_PROTECT is called by a non-preemptible command processor task in order to change the configurable information. SET_WRITE_PROTECT is always called to reenable the hardware protection.

Comdeck	CMICWP
Format	CLEAR_WRITE_PROTECT
Input	None.
Output	None.
Remarks	The proper use of this routine is in conjunction with SET_WRITE_PROTECT. The order of use should be:

1. CLEAR_WRITE_PROTECT,
2. modify the normally write-protected area of memory, and
3. SET_WRITE_PROTECT.

CLOSE_STATISTICS_SAP

This procedure allows a software component to close a previously opened statistics SAP.

Comdeck	**SMXSAPM**
Format	**CLOSE_STATISTICS_SAP (sap_id, element_type, element_name, status)**
Input	**sap_id: 0..0ffff(16)** This parameter identifies the statistics SAP to be closed. **element_type: statistics_type** This parameter indicates the element type for which a statistics SAP is to be closed. **element_name: string (* <= 31)** This parameter indicates the name of the element for which a statistics SAP is to be closed.
Output	**status: close_statistics_status** This parameter returns the call status.

CLOSE_STATUS_SAP

This procedure allows software components to close previously opened status SAPs.

Comdeck	**SDXSSAR**
Format	**CLOSE_STATUS_SAP (sap_number)**
Input	**sap_number: software_sap_range** This parameter uniquely identifies the previously opened SAP that is to be closed. The sap_number must be the sap_number returned on the open_status_sap call.
Output	None.
Remarks	The procedure NOPREMPT is called upon entering CLOSE_STATUS_SAP to suppress task preemption. CLOSE_STATUS_SAP is exited in a non-preemptable state; the caller must make a call to the procedure OKPREMPT if preemptability is desired. Global data modified: software_status_sap_table.

CONVERT_INTEGER_TO_POINTER

Converts an integer to a pointer for users who need to do pointer arithmetic.

Comdeck	CMIPCIP
Format	**address := CONVERT_INTEGER_TO_POINTER (value)**
Input	**value: integer** This parameter indicates the integer to be converted.
Output	**address: ^cell** This parameter returns the integer in pointer format.
Remarks	Use of this procedure should probably be restricted to hardware interface routines.

CONVERT_POINTER_TO_INTEGER

Converts a pointer to an integer for users who need to do pointer arithmetic.

Comdeck	CMIPCPI
Format	**number := CONVERT_POINTER_TO_INTEGER (value)**
Input	**value: ^cell** This parameter indicates the address to be converted.
Output	**number: integer** This parameter returns the integer value of the converted address.
Remarks	Use of this procedure should probably be restricted to hardware interface routines.

COPY

This procedure copies a message to another buffer chain. The message is logically copied to new buffers, and the old set of buffers is released.

Comdeck	CMXPCPY
Format	**COPY (message, threshold)**
Input	**message: buf_ptr** This parameter indicates the message to be copied. **threshold: threshold_size** This parameter indicates the threshold for buffer acquisition.
Output	**message: buf_ptr** This parameter returns the new buffer address of the copied message.
Remarks	The message parameter must be a valid buffer chain address.

DEAD_STOP

This procedure calls RESET_DI with a reset code of software_dead_stop.

Comdeck	CMXPDED
Format	**DEAD_STOP (halt_code)**
Input	**halt_code: 0 .. 0ff(16)** This parameter is required at compile-time, but is currently not used at run-time.
Output	None.

DECREMENT_MODULE_USE_COUNT

The module use count of the indicated entry point is decremented. If the count becomes zero, then the module is made available for deload. If the counter becomes negative, the system is reset and reloaded by DEAD_STOP. If the given entry point name is all blanks, then the module use count of the running task is decremented. This procedure is used when the module use count was previously incremented and procedure STOP_TASK will not be called to decrement the counter.

Comdeck	**DLXDMUC**
Format	**DECREMENT_MODULE_USE_COUNT (entry_point_name, entry_point_found)**
Input	**entry_point_name: pmt$program_name** This parameter indicates the entry point of the module whose use count is to be decremented.
Output	**entry_point_found: boolean** This parameter returns an indication of whether the entry point was located.

DELAY_PROCESSING

This routine may be called to delay processing for a finite period of time, as in a timeout mechanism. A normal return occurs when processing is resumed.

Comdeck CMXPDLY

Format **DELAY_PROCESSING (hours, minutes, seconds, milliseconds)**

Input

hours: 0 .. 24

This parameter indicates the number of hours in the delay.

minutes: 0 .. 59

This parameter indicates the number of minutes in the delay.

seconds: 0 .. 59

This parameter indicates the number of seconds in the delay.

milliseconds: 0 .. 999

This parameter indicates the number of milliseconds in the delay.

Output None.

Remarks The Executive CALL_AFTER_INTERVAL service is used to restart the task. The Executive guarantees that the requestor will wait at least as long as requested, but does not guarantee a maximum period. Thus, DELAY_PROCESSING (0,0,0,200) will delay at least 200 milliseconds, but may delay longer, even up to several seconds in a very busy system.

DEVICE_STARTED

This routine checks whether or not the peripheral board exists and has been started.

Comdeck	**DMXXDS**
Format	**DEVICE_STARTED (ip_number)**
Input	**ip_number: 0 .. upper_ip_num** This parameter indicates the slot number of the peripheral board.
Output	**status: boolean** This parameter returns a boolean value of TRUE if the peripheral board has been started. FALSE is returned if the peripheral board does not exist or has not been started.

DI_DEBUG

This procedure calls the DI Debugger from within a user program.

If the DI Debugger program has already been initialized by a DI_DEBUG request, it will retain its current trap state, enter its main input loop, and wait for Debugger commands to be entered through the DI console.

Comdeck	CMXDBUG
Format	DI_DEBUG
Input	None.
Output	None.

DI_DEBUG_INIT

This procedure initializes the DI Debugger program to console break mode only. While the Debugger is in this mode, all of the available Debugger program error stops are enabled the first time a user presses the BREAK key (or enters a break 2 sequence) from the DI console. If the Debugger program has already been initialized, this request is ignored and the Debugger program is left in its current state.

Comdeck	CMXDBUG
Format	DI_DEBUG_INIT
Input	None.
Output	None.

DUMP_CLOSE

This procedure indicates to the dump task that the transfer of dump information is complete.

Comdeck	CMXSISA
Format	**DUMP_CLOSE (sa_dump_identifier)**
Input	**sa_dump_identifier: ^cell** This parameter indicates the address of the dump control block.
Output	None.
Remarks	If the sa_dump_identifier is not valid, no message will be sent and control returns to the caller.

DUMP_IP_MEMORY

This procedure dumps the peripheral board's local RAM. This is done when either DVM or the peripheral's local RAM returns an error status when executing QUEUE_IP_COMMAND. Failure management software then issues the DUMP_IP_MEMORY procedure to DVM.

Comdeck **DMXDIP**

Format **DUMP_IP_MEMORY (dip_parm_ptr, image_location, status)**

Input **dip_parm_ptr: dip_parm_ptr_type**

This parameter indicates the address of the command parameter packet. This record specifies the number of the peripheral, the address of the dump, and the length of the dump.

Output **image_location: ^cell**

This parameter returns the address of the image location where the dumped RAM is stored. This is located in SMM.

status: 0..0ffff(16)

This parameter returns the status of DUMP_IP_MEMORY. The status codes are:

0: command executed.
1: invalid card slot specified.
2: start device service not issued for specified peripheral board.
3: unable to procure memory for dump.
4: peripheral board not responding to the dump command.

DUMP_WRITE

This procedure appends a header and dump information to the dump buffer chain associated with the dump identifier. If the total number of bytes in the buffer chain is above the maximum allowed in a dump buffer chain, then message(s) will be sent to the dump task identifying buffers to be immediately written to the dump file. This is used to capture any pertinent data during a recovery.

Comdeck **CMXSISA**

Format **DUMP_WRITE (sa_dump_identifier, dump_address, dump_byte_count, threshold)**

Input

sa_dump_identifier: ^cell

This parameter indicates the address of the dump control block.

dump_address: ^cell

This parameter indicates the address of the information to be dumped.

dump_byte_count: sat$max_dump_size

This parameter indicates the number of bytes to dump; rounded up to an even number of bytes.

threshold: threshold_size

This parameter indicates the threshold to be used in obtaining buffers.

Output **None.**

Remarks If the sa_dump_identifier is not valid, then the dump information will be discarded and control returns to the caller (the routine will abort).

FG_TRIM

This procedure trims the specified number of bytes from the end of the data descriptor. If a buffer is completely used up, it is released from memory. If the entire message is less than the requested size, the caller is informed that there are not enough bytes to satisfy the request.

Comdeck	**CMXPFGT**
Format	**FG_TRIM (size, address, message)**
Input	**size: non_empty_message_size** This parameter indicates the number of bytes needed. **address: ^cell** This parameter indicates where to position the bytes that are retrieved. **message: buf_ptr** This parameter indicates the destination address for any diagnostic message generated from the call.
Output	**message: buf_ptr** This parameter returns the address of any diagnostic message generated from the call.

FIELD_SIZE

This function converts a multiple of any of the Management Data Unit (MDU) field types to its length in bytes.

Comdeck	**MEXGDF**
Format	**count := FIELD_SIZE (length, field_type)**
Input	**length: 1 .. mdu_field_size** This parameter indicates how many units of field_type there are in the MDU. **field_type: mdu_field_type** This parameter indicates the MDU field type.
Output	**count: 0 .. mdu_field_size** This parameter returns zero if an unsupported field type was specified; else, number of bytes.

FIND

This procedure finds and returns the table in a tree that is associated with the specified integer key. The table is returned interlocked, (that is, task pre-emption from interrupt levels is disabled).

Comdeck CMXPFIN

Format **address:= FIND (head, key)**

Input **head: ^root**

This parameter indicates the root of the tree structure to be searched.

key: integer

This parameter indicates the key to be searched for.

Output **address: ^cell**

This parameter returns the address of the table located by the FIND operation. If the key is not found, the return is NIL.

FIND_FIRST

This procedure searches the specified integer tree structure in left-node-right order for a node that contains a key greater than the input parameter, key. For each node that satisfies this condition, FIND_FIRST calls the user-supplied procedure (unless parameter qual is NIL). The search continues until the user-supplied procedure returns a value of TRUE (refer to appendix D). See also FIND_NEXT, SFIND_FIRST, and SFIND_NEXT.

This procedure is identical to FIND_NEXT.

Comdeck CMXPFNF

Format **table = FIND_FIRST (head, qual, param, key)**

Input

head: ^root

This parameter indicates the address of the head root of the tree structure to be searched.

key: integer

This parameter indicates the key associated with the node from which the search is to begin.

qual: ^procedure

This parameter indicates the address of the procedure that will be called at each node. Refer to appendix D.

param: ^cell

This parameter indicates the parameter to pass to qual.

Output

key: integer

This parameter returns the key associated with the first table to satisfy the specified parameter, param.

table: ^cell

This parameter returns the address of the first table to satisfy the specified parameter, param.

FIND_FREE_NODE

This procedure finds the first key that is greater than the specified key and that is not associated with a table. The search is performed by comparing the key value passed in the call to the key.numeric value in the current node.

Comdeck **CMXPFFN**

Format **FIND_FREE_NODE (head, key)**

Input **head: ^root**

This parameter indicates the root of the tree to be searched.

key: integer

This parameter indicates the key used for comparison in the search.

Output **key: integer**

This parameter returns the first key value that is greater than the input key and that is not associated with a table.

FIND_NEXT

This procedure searches the specified integer tree structure in left-node-right order for a node that contains a key greater than the input parameter, key. For each node that satisfies this condition, FIND_NEXT calls the user-supplied procedure (unless parameter qual is NIL). The search continues until the user-supplied procedure returns a value of TRUE (refer to appendix D). See also FIND_FIRST, SFIND_FIRST, and SFIND_NEXT.

This procedure is identical to FIND_FIRST.

Comdeck **CMXPFNF**

Format **table = FIND_NEXT (head, qual, param, key)**

Input

head: ^root

This parameter indicates the address of the root of the tree structure to be searched.

key: integer

This parameter indicates the key associated with the node from which the search is to begin.

qual: ^procedure

This parameter indicates the address of the procedure that will be called at each node. Refer to appendix D.

param: ^cell

This parameter indicates the parameter to pass to qual.

Output

key: integer

This parameter returns the key associated with the first table to satisfy the specified parameter, param.

table: ^cell

This parameter returns the address of the first table to satisfy the specified parameter, param.

FIRST_BYTE_ADDRESS

This procedure returns the address of the first byte of a message. It is intended for fast access by protocols that regularly make use of the first byte.

Comdeck	**CMXPFBA**
Format	**byte_address:= FIRST_BYTE_ADDRESS (message)**
Input	**message: buf_ptr** This parameter indicates the message whose first byte address is sought.
Output	**byte_address: ^string (1)** This parameter returns the first byte address of the specified message.

FIRST_NODE

This procedure allocates space for the first node of a tree structure. Associated values are placed in the first node and the head node is linked to the first node.

Comdeck CMXPNEW

Format **FIRST_NODE (head, key, table, size)**

Input

head: ^root

This parameter indicates the root of the tree structure.

key: key_record

This parameter indicates the key to be associated with the first node, for searching operations.

table: ^node_control

This parameter indicates the address of the table to be associated with the first node.

size: integer

This parameter indicates the size in bytes of the specified table.

Output None.

FORCE_STATISTICS_REPORTING

This procedure forces the specified statistics function procedure to be called to issue a log message.

Comdeck **SMXSAPM**

Format **FORCE_STATISTICS_REPORTING (sap_id, select_sds_hdr_ptr, param, status)**

Input

sap_id: 0..0ffff(16)

This parameter identifies the statistics SAP.

select_sds_hdr_ptr: ^sds_header

This parameter indicates the SDS header to be used for logging.

param: ^cell

This parameter indicates the parameter for the statistics function procedure.

Output

status: force_stat_reporting_status

This parameter returns an indication of the call status.

FRAGMENT

This procedure extracts a message fragment from the specified message.

Comdeck **CMXPFRA**

Format **FRAGMENT (size, remainder_ptr, fragment_ptr, threshold)**

Input

size: non_empty_message_size

This parameter indicates the byte length of the message fragment to be extracted.

remainder_ptr: buf_ptr

This parameter indicates the address of the source message buffer.

fragment_ptr: buf_ptr

This parameter indicates the address of the buffer to which the fragment is to be moved.

threshold: threshold_size

This parameter indicates the threshold for buffer acquisition.

Output

remainder_ptr: buf_ptr

This parameter returns the address of the remaining portion of the source message buffer. This parameter will be set to NIL if the length of the size parameter is greater than the length of the source message buffer.

fragment_ptr: buf_ptr

This parameter returns the address of the message fragment buffer.

GEN_DATA_FIELD

This procedure generates a data field in MDU format.

Comdeck	**MEXGDF**
Format	**GEN_DATA_FIELD (msgbuf, field_cell, length, type)**

Input

msgbuf: buf_ptr

This parameter points to the buffer containing the data field(s).

field_cell: ^cell

This parameter indicates the data field.

length: 1..mdu_field_size

This parameter indicates the data field length.

type: mdu_field_type

This parameter indicates a data field type of MDU. Refer to Volume 3 of the Systems Programmer's Reference Manual for a complete description of MDU types.

Output

msgbuf: buf_ptr

This parameter returns a pointer to the buffer containing the MDU data field(s).

GEN_TEMPLATE_ID

This procedure puts template identifier fields into buffers for generation of log messages or command responses. Refer to the Command ME chapter of Volume 2 of the Systems Programmer's Reference Manual for information on log message and command response templates.

Comdeck **CSXGTI**

Format **GEN_TEMPLATE_ID (msgbuf, template_id)**

Input **msgbuf: buf_ptr**

This parameter points to the buffer that contains the log message or command response data fields.

template_id: template_id_type

This parameter indicates the log message or command response template number.

Output **msgbuf: buf_ptr**

This parameter returns the buffer containing data field(s).

GET_CARD_TYPE_AND_ADDRESS

This procedure gets the card type and card address for the device name specified.

Comdeck **SDXGCTA**

Format **GET_CARD_TYPE_AND_ADDRESS (device_name, device_record, device_available)**

Input **device_name: string (*)**

This parameter indicates the hardware device name whose card type and address is desired.

Output **device_record: card_info_record**

This parameter returns the card type and card address for the device name specified.

device_available: boolean

If an invalid device name has been specified or if the associated board type is not physically available in the System Status Table, then this parameter returns FALSE; TRUE otherwise.

GET_CIM_BOOT_SOURCE

This procedure returns the LIM and port number over which a DI was booted. The user must know the slot number of the peripheral board over which the boot was loaded.

Comdeck **DMXXPS**

Format **GET_CIM_BOOT_SOURCE (ip_number, cim_number, port_number, status)**

Input **ip_number: 0..upper_ip_num**

This parameter indicates the slot number of the peripheral board over which the DI was booted.

Output **cim_number: 0..upper_cim_num**

This parameter returns the number of the CIM board for the specified DVMID.

port_number: 0..upper_port_num

This parameter indicates the number of the port over which the DI was booted.

status: 0..0ffff(16)

This parameter returns the status of GET_CIM_BOOT_SOURCE. The status codes are:

0: command executed.
1: invalid board slot supplied.
2: start device service not issued for specified peripheral board.
3: unable to retrieve boot source.
4: peripheral board not responding to request.

GET_CIM_NUMBER

This DVM routine is used to determine the CIM slot number from the user supplied LIM and port numbers.

Comdeck **DMXXPS**

Format **cim_number:= GET_CIM_NUMBER (lim_number, port_number)**

Input **lim_number: 0..upper_lim_num**

This parameter indicates the slot number of the LIM board.

port_number: 0..upper_port_num

This parameter indicates the port number on the LIM board.

Output **cim_number: 0..upper_ip_num**

This parameter returns the CIM slot number for the specified LIM and port numbers. Range is 1..7. A return of 0 signifies an invalid LIM or port number.

GET_DATA_FIELD

This common procedure extracts data fields from MDU formatted messages and returns them in an internal format. Since a data field may consist of several subfields, the data is previewed to determine how much memory is needed. Then the fields are stripped until the field-complete flag is seen.

Comdeck **MEXGDF**

Format **GET_DATA_FIELD (msgbuf, field_cell, len, type)**

Input

msgbuf: buf_ptr

This parameter points to a buffer containing the data unit.

Output

msgbuf: buf_ptr

This parameter returns the updated pointer to buffer containing the data unit.

field_cell: ^cell

This parameter returns the address of the data field. It will be equal to NIL if no more data is available or if any errors are encountered in the data fields.

len: 0 .. mdu_field_size

This parameter returns the data field length.

type: mdu_field_type

This parameter returns the data field type.

Remarks

NOTE

The caller is responsible for freeing the retrieved memory extent.

GET_EXPRESS, MAYBE_EXPRESS

These procedures get ITMs from the express queue. If a message is found on the express queue, it is copied to the addressed space and removed from the ITM queue. The normal queue is not inspected.

These calls have the following effects:

GET_EXPRESS	Control returns after a message has been made available to the caller.
MAYBE_EXPRESS	A message is obtained, or a failure is returned.

Comdeck **CMXMTSK**

Format **GET_EXPRESS (intertask_message, task_sending_message)**
MAYBE_EXPRESS (intertask_message, task_sending_message)

Input None.

Output **intertask_message: ^cell**

This parameter returns the address of the dequeued ITM.

task_sending_message: task_ptr

This parameter returns the task_id of the task that sent the ITM.

GET_FIRST_BYTE

This function returns the first valid byte of text from the specified message in the form of a character. It provides fast access to protocols that regularly make use of the first byte.

Comdeck	**CMXPGFB**
Format	**byte:= GET_FIRST_BYTE (message)**
Input	**message: buf_ptr** This parameter indicates the message whose first byte of text is to be returned.
Output	**byte: char** This parameter returns the first byte of the specified message.

GET_LAST_BYTE

This function returns the last valid byte of text from the specified message in the form of a character.

Comdeck	**CMIGLB**
Format	**byte:= GET_LAST_BYTE (message)**
Input	**message: buf_ptr** This parameter indicates the first descriptor of the message whose last byte of text is to be returned.
Output	**byte: char** This parameter indicates the last byte of the specified message.
Remarks	It is assumed that the last buffer in the chain will not be empty.

GET_LONG_BUFFERS, FG_LONG_BUFFERS, MAYBE_LONG_BUFFERS

These procedures get one or more data buffers, with the following variations:

GET_LONG_BUFFERS	The buffers are obtained.
FG_LONG_BUFFERS	For interrupt routine use only; the buffers are obtained or a failure is returned.
MAYBE_LONG_BUFFERS	The buffers are obtained or a failure is returned.

Comdeck CMXPGBF

Format **GET_LONG_BUFFERS (count, buffer_address, threshold)**
FG_LONG_BUFFERS (count, buffer_address, threshold)
MAYBE_LONG_BUFFERS (count, buffer_address, threshold)

Input **count: buffer_request_limit**

This parameter indicates the number of buffers being requested.

threshold: threshold_size

This parameter indicates the threshold for obtaining buffers.

Output **buffer_address: buf_ptr**

This parameter returns the address of the buffer chain obtained.

GET_MEMORY, FG_MEMORY, MAYBE_MEMORY

These procedures get global memory extents, with the following effects:

GET_MEMORY	The memory extent is obtained.
FG_MEMORY	For interrupt routine use only; the memory extent is obtained or a failure is returned.
MAYBE_MEMORY	The memory extent is obtained or a failure is returned.

Comdeck CMXPGGX

Format GET_MEMORY (extent_returned, extent_size)
FG_MEMORY (extent_returned, extent_size)
MAYBE_MEMORY (extent_returned, extent_size)

Input **extent_size: executive_extent**

This parameter indicates the size of the memory extent to be obtained.

Output **extent_returned: ^cell**

This parameter returns the address of the memory extent obtained.

GET_MESSAGE_LENGTH

This function returns the byte length of the specified message.

Comdeck	CMXPGML
Format	size:= GET_MESSAGE_LENGTH (message)
Input	**message: buf_ptr** This parameter indicates the address of the message to be measured.
Output	**size: message_size** This parameter returns the length in bytes of the message specified.
Remarks	If the address of the message is NIL, then the output parameter returns a value of zero.

GET_MPB_EXTENT, FG_MPB_EXTENT, MAYBE_MPB_EXTENT

These procedures get MPB RAM memory extents with the following effects:

GET_MPB_EXTENT	The memory extent is obtained
FG_MPB_EXTENT	For interrupt routine use only; the memory extent is obtained or a failure is returned.
MAYBE_MPB_EXTENT	The memory extent is obtained or a failure is returned.

Comdeck CMXPGMP

Format **GET_MPB_EXTENT (extent_returned, extent_size)**
FG_MPB_EXTENT (extent_returned, extent_size)
MAYBE_MPB_EXTENT (extent_returned, extent_size)

Input **extent_size: executive_extent**

This parameter indicates the size of the MPB extent to be obtained.

Output **extent_returned: ^cell**

This parameter returns the address of the MPB extent obtained.

GET_MSG, MAYBE_MSG

These procedures get ITMs from the normal or express queues. If a message is found on either the normal or express queue, it is copied to the addressed space, and removed from the ITM queue. The express queue has priority. These calls have the following effects:

GET_MSG	Control returns after a message has been made available to the caller.
MAYBE_MSG	A message is obtained, or a failure is returned.

Comdeck **CMXMTSK**

Format **GET_MSG (intertask_message, sender)**
MAYBE_MSG (intertask_message, sender)

Input None.

Output **intertask_message: ^cell**

This parameter returns the address of the dequeued ITM.

sender: task_ptr

This parameter returns the task_id of the task that sent the ITM.

GET_NEXT_STATUS_SAP

Provides a command processor the ability to retrieve the addresses of its associated software components' status tables when multiple copies are executing at the same time.

Comdeck **SDXSSAR**

Format **GET_NEXT_STATUS_SAP (name, last_sap_table_ptr, next_sap_table_ptr, task_id, successful, response)**

Input **name: string (* <= 31)**

This parameter indicates the name of the software component of interest.

last_sap_table_ptr: ^cell

This parameter indicates the address of the status table of the previously obtained SAP. This parameter should be set to the returned value of the parameter next_sap_table_ptr from the previous call when GET_NEXT_STATUS_SAP is being used recursively.

If no previous SAP was obtained, this parameter should be set to NIL.

Output **next_sap_table_ptr: ^cell**

This parameter returns the address of the software components status table of the next registered SAP in the table.

If the specified value of last_sap_table_ptr is NIL, then next_sap_table_ptr will contain the address of the first associated status SAP for the named software component.

If the next_sap_table_ptr is returned NIL, then the software component has not opened a status SAP or it has no status to report. Or, if the procedure is being used recursively, NIL will be returned when all the associated SAPs have been retrieved.

task_id: task_ptr

This parameter returns the task_id of the software component that opened the software status SAP.

successful: boolean

This parameter returns TRUE if the last_sap_table_ptr was found; FALSE otherwise.

response: buf_ptr

This parameter contains a response to be sent to the origin of the command if its value is not NIL.

Remarks The procedure NOPREMPT is called when entering GET_NEXT_STATUS_SAP to suppress task preemption; GET_NEXT_STATUS_SAP is exited in a non-preemptible state. You must make a call to the procedure OKPREMPT if preemptibility is so desired.

GET_PMM_EXTENT, FG_PMM_EXTENT, MAYBE_PMM_EXTENT

These procedures obtain private memory extents, with the following effects:

GET_PMM_EXTENT	The memory extent is obtained
FG_PMM_EXTENT	For interrupt routine use only; the memory extent is obtained or a failure is returned.
MAYBE_PMM_EXTENT	The memory extent is obtained or a failure is returned.

Comdeck **CMXPGPM**

Format **GET_PMM_EXTENT (extent_returned, extent_size)**
FG_PMM_EXTENT (extent_returned, extent_size)
MAYBE_PMM_EXTENT (extent_returned, extent_size)

Input **extent_size: executive_extent**

This parameter indicates the size of the private memory extent to be obtained.

Output **extent_returned: ^cell**

This parameter returns the address of the private memory extent obtained.

GET_SHORT_BUFFERS, FG_SHORT_BUFFERS, MAYBE_SHORT_BUFFERS

These procedures get one or more descriptor buffers, with the following effects:

GET_SHORT_BUFFERS	The buffers are obtained.
FG_SHORT_BUFFERS	For interrupt routine use only; the buffers are obtained or a failure is returned.
MAYBE_SHORT_BUFFERS	The buffers are obtained or a failure is returned.

Comdeck CMXPGDB

Format **GET_SHORT_BUFFERS (count, buffer_address, threshold)**
FG_SHORT_BUFFERS (count, buffer_address, threshold)
MAYBE_SHORT_BUFFERS (count, buffer_address, threshold)

Input **count: buffer_request_limit**

This parameter indicates the number of buffers to be obtained.

threshold: threshold_size

This parameter indicates the threshold for obtaining buffers.

Output **buffer_address: buf_ptr**

This parameter returns the address of buffer chain obtained.

GET_SIZE_N_ADDR

This procedure gets the size and address of the memory extent for the specified section. The size and address are determined from the indicated start section address.

Comdeck SIXGSIZ

Format **GET_SIZE_N_ADDR (section_address, section_size)**

Input

section_address: ^cell

This parameter indicates the section of interest.

Output

section_address: ^cell

This parameter returns the address of the section whose size is being returned.

section_size: llt$section_length

This parameter returns the size of the specified section.

Remarks

NOTE

Output parameter section_address will be 6 bytes less than input parameter section_address.

GET_STATUS_RECORD

This procedure retrieves a status record for the device name specified.

If an invalid device name is specified or the associated board type is not physically available in the associated System Status Table, then a status indication is returned that indicates that the device named is not available in the DI.

Comdeck **SDXGPSR**

Format **GET_STATUS_RECORD (device_name, device_status_record, device_available)**

Input **device_name: string (maximum_device_name_size)**

This parameter identifies the hardware device name whose status record is desired.

Output **device_status_record: component_status_type**

This parameter returns the status record for the device name specified.

device_available: boolean

If the device named is in the DI, TRUE is returned; FALSE otherwise.

GET_STATUS_SAP

The purpose of this procedure is to provide a command processor the ability to retrieve the address of its associated software component status table.

Comdeck	**SDXSSAR**
Format	**GET_STATUS_SAP (name, sap_table_ptr, task_id, response)**
Input	**name: string (* <= 31)** This parameter indicates the name of the software component of interest.
Output	**sap_table_ptr: ^cell** This parameter returns the address of the software components status table. If the sap_table_ptr is returned NIL, then either the software component has not opened a status SAP or it has no status to report.
	task_id: task_ptr This parameter returns the task_id of the software component who opened the software status SAP.
	response: buf_ptr This parameter returns a response to be sent to the origin of the command if its value is not NIL.
Remarks	The procedure NOPREMPT is called when entering get_status_sap to suppress task preemption. GET_STATUS_SAP is exited in a non-preemptible state and will require the caller to make a call to the procedure OKPREMPT if preemptibility is so desired.

GROW

This procedure adds a new table to a tree structure.

Comdeck **CMXPGRO**

Format **address:= GROW (head, key, table, size)**

Input

head: ^root

This parameter indicates the root of the tree.

key: integer

This parameter indicates the key to be used for searching operations.

table: ^cell

This parameter indicates the table to be added to the tree.

size: integer

This parameter indicates the table size.

Output

address: ^ cell

If an association already exists between the specified key and the table structure, the table is returned and no update is performed. Otherwise, the association is created and NIL is returned.

Remarks

The procedure NOPREMPT is called upon entering GROW to suppress task preemption. GROW is exited in a non-preemptible state and will require the caller to make a call to the procedure OKPREMPT if preemptibility is so desired.

I_COMPARE

This function compares the lengths of two strings.

Comdeck INXCMP

Format **result:= I_COMPARE (string1, string2)**

Input **string1: string (*)**

This parameter indicates the first of two strings specified for comparison. Length may be up to 256 characters.

string2: string (*)

This parameter indicates the second of two strings specified for comparison. Length may be up to 256 characters.

Output **result: -1..1**

If string1 < string2, the return is -1.

If string1 = string2, the return is 0.

If string1 > string2, the return is 1.

I_COMPARE_COLLATED

This function translates characters in two strings according to a translation table, and compares the strings.

Comdeck INXCMPC

Format **result:= I_COMPARE_COLLATED (string1, string2, table)**

Input **string1: string (*)**

This parameter indicates the first of two strings specified for comparison. Length may be up to 256 characters.

string2: string (*)

This parameter indicates the second of two strings specified for comparison. Length may be up to 256 characters.

table: string (256)

This parameter indicates the translation table.

Output **result: -1..1**

If string1 < string2, the return is -1.

If string1 = string2, the return is 0.

If string1 > string2, the return is 1.

I_SCAN

This procedure scans a string from left to right until one of a specified set of characters is found, or until the entire string has been searched.

Comdeck **INXSCAN**

Format **I_SCAN (select, string, index, found_char)**

Input

select: ^packed array [char] OF boolean

This parameter indicates the address of the set of elements that the string is to be searched for.

string: string (*)

This parameter indicates the string to be searched.

Output

index: integer

This parameter returns the location from the left-most character of the string of the character that was found.

found_char: boolean

This parameter returns an indication of whether or not any of the specified characters has been found.

I_TRANSLATE

This procedure translates each character in a source field according to a translation table, and transfers the result to a destination field.

Comdeck	**INXTRAN**
Format	**I_TRANSLATE (table, source, destination)**
Input	**table: string (256)** This parameter specifies the translation table. **source: string (*)** This parameter indicates the string to be translated. Length may be up to 256 characters. **destination: string (*)** This parameter indicates the destination of the string to be translated. Length may be up to 256 characters.
Output	**destination: string (*)** This parameter returns the destination of the translated string. Length may be up to 256 characters.
Remarks	If the length of the source field is less than the length of the destination field, translated blanks fill the destination field. If the source field is larger than the destination field, the right-most characters of the source field are truncated in the destination field.

INCREMENT_MODULE_USE_COUNT

This procedure increments the module use count of a loaded module, thus preventing module deloading. This procedure is only used when START_NAMED_TASK, LOAD_ENTRY_POINT or LOAD_ABSOLUTE_MODULE have not been used to prevent module deloading.

Comdeck **ILXIMUC**

Format **INCREMENT_MODULE_USE_COUNT (entry_point_name, entry_point_found)**

Input **entry_point_name: pmt$program_name**

This parameter indicates the entry point of the module whose use count is to be incremented. If the specified entry point name is all blanks, then the module use count of the currently running task is incremented.

Output **entry_point_found: boolean**

This parameter returns an indication of whether or not the module was found.

INIT_ROOT

This procedure initializes the root of a tree. This includes setting up initial values for interlocks and node addresses, as well as setting up the (up to) four character ASCII name of the table stored in each node.

Comdeck CMIPINT

Format **INIT_ROOT (root, type_node, name)**

Input

root: ^root

This parameter indicates the root of the tree structure that is to be initialized.

type_node: key_type

This parameter indicates the key type to be used in referencing the tree structure's nodes.

name: string (4)

This parameter indicates the ASCII name of the table to be stored in each node.

Output None.

INITIALIZE_STATISTIC_RECORD

This procedure will initialize MDU formatted records used in statistics collection.

Comdeck	**SMXSAPM**
Format	**INITIALIZE_STATISTIC_RECORD (number_of_records, number_of_data_bytes, record_starting_ptr)**
Input	**number_of_records: 1..0ff(16)** This parameter indicates the number of records to initialize. **number_of_data_bytes: number_of_data_byte_types** This parameter indicates the need for a two-, four- or eight-byte statistic record. **record_starting_ptr: ^cell** This parameter is a pointer to the starting record.
Output	**record_starting_ptr: ^cell** This parameter returns a pointer to the next record.

LOAD_ABS_MODULE_AND_DELAY

This procedure loads an absolute module if it is not currently loaded. If the module is already loaded, information is returned pertaining to the module. The caller waits while loading occurs.

Comdeck	**DLXLAMD**
Format	**LOAD_ABS_MODULE_AND_DELAY (module_name, smm_address, load_address, transfer_address, byte_size, absolute_module_found, error_response)**
Input	**module_name: pmt$program_name** This parameter indicates the module to be loaded. Maximum length of string is 31 characters.
Output	**smm_address: ^cell** This parameter returns the starting address of the module in SMM. **load_address: dlt$68000_address** This parameter returns the address where the module has been loaded. **transfer_address: dlt$68000_address** This parameter returns the address at which module execution begins. Range is 0 .. 7fffffff(16). **byte_size: dlt$section_length** This parameter returns the size of the module in bytes. **absolute_module_found: boolean** This parameter indicates whether or not the module was found. **error_response: clt$status** Any error message from the Online Loader is returned by this parameter.
Remarks	The module use count is incremented to prevent module deloading. If the parameter absolute_module_found is returned FALSE, it is the user's responsibility to release the buffer chain returned in error_ **response.condition.**

LOAD_ABS_MODULE_AND_PROCEED

This procedure returns information pertaining to a named module if it is already loaded. If it is not loaded, the procedure sends an ITM to the Online Loader to load the module. The calling procedure continues to operate while the Online Loader is processing the request.

Comdeck **DLXLAMP**

Format **LOAD_ABS_MODULE_AND_PROCEED (module_name, reply_procedure, request_id)**

Input

module_name: pmt$program_name

This parameter indicates the module to be loaded. Maximum length of string is 31 characters.

reply_procedure: ^procedure

This parameter indicates the address of a procedure that returns information about the module to the original calling procedure. Refer to appendix D.

request_id: ^cell

This parameter indicates the address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

Output None.

Remarks The module use count is incremented to prevent module deloading.

This procedure makes use of a reply procedure to capture returned information while processing continues.

If the parameter absolute_module_found is returned FALSE to the reply procedure, it is the user's responsibility to release the buffer chain returned in error_response.condition.

LOAD_CMD_PROCESSOR_AND_DELAY

Given a command processor name, this procedure loads a module if it is not currently loaded. The task attribute block is found and validated (defaults are used on error), and entry point information is returned. The caller waits while loading occurs.

Comdeck **DLXLCPD**

Format **LOAD_CMD_PROCESSOR_AND_DELAY (entry_point_name, entry_point_found, entry_address, task_info, error_response, module_ptr)**

Input

entry_point_name: pmt$program_name

This parameter specifies the name of the entry point of the module to be loaded..

Output

entry_point_found: boolean

This parameter indicates whether the specified entry point was found.

entry_address: ^dlt$entry_description

This parameter returns the address of the loaded module entry point.

task_info: task_attributes

This parameter returns the stack size, priority, task preemptibility, and an indication of whether or not the task is immediate control. Refer to appendix C.

error_response: clt$status

This parameter returns any error message generated by the Online Loader.

module_ptr: dlt$load_id_ptr

This parameter returns the address of the loaded module.

Remarks

The module_use_count is incremented to prevent module deloading.

If the parameter entry_point_found returns FALSE, it is the user's responsibility to release the buffer chain returned in error_response.condition.

LOAD_CMD_PROCESSOR_AND_PROCEED

Given a command processor name, this procedure loads a module if it is not currently loaded. The task attribute block is found and validated (defaults are used on error), and entry point information is returned. The calling procedure continues to operate while the Online Loader is processing the request.

Comdeck DLXLCPP

Format **LOAD_CMD_PROCESSOR_AND_PROCEED (entry_point_name, reply_procedure, request_id)**

Input

entry_point_name: pmt$program_name

This parameter indicates the entry point of the module to be loaded. Maximum length is 31 characters.

reply_procedure: ^procedure

This parameter indicates the address of a procedure that returns information about the module to the original calling procedure. Refer to appendix D.

request_id: ^cell

This parameter indicates the address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

Output None.

Remarks The module_use_count is incremented to prevent module deloading.

This procedure makes use of a reply procedure to capture returned information while processing continues.

If the parameter entry_point_found returns FALSE to the reply procedure, it is the user's responsibility to release the buffer chain returned in error_ resonse.condition.

LOAD_ENTRY_POINT_AND_DELAY

Given an entry point name, this procedure loads the associated module unless it is currently loaded. If the load fails, an error message is returned in the parameter error_response. The task attribute block is found and validated (defaults are used on error). Entry point information is returned. The caller waits while loading occurs.

Comdeck **DLXLEPD**

Format **LOAD_ENTRY_POINT_AND_DELAY (entry_point_name, entry_point_found, entry_address, task_info, error_response, module_ptr)**

Input **entry_point_name: pmt$program_name**

This parameter indicates the entry point of the module to be loaded. Maximum length is 31 characters.

Output **entry_point_found: boolean**

This parameter returns an indication of whether the specified entry point was found.

entry_address: ^dlt$entry_description

This parameter returns the address of the loaded module entry point.

task_info: task_attributes

This parameter returns the stack size, priority, task preemptibility, and an indication of whether or not the task is immediate control. Refer to appendix C.

error_response: clt$status

This parameter returns any error message generated by the Online Loader.

module_ptr: dlt$load_id_ptr

This parameter returns the address of the loaded module.

Remarks The module_use_count is incremented to prevent module deloading.

If the parameter entry_point_found is returned FALSE, it is the USER'S responsibility to release the buffer chain returned in error_response.condition.

LOAD_ENTRY_POINT_AND_PROCEED

Given an entry point name, this procedure loads the associated module unless it is currently loaded. If the load fails, an error message is returned in the parameter error_response. The task attribute block is found and validated (defaults are used on error). Entry point information is returned. The calling procedure continues to operate while the Online Loader is processing the request.

Comdeck **DLXLEPP**

Format **LOAD_ENTRY_POINT_AND_PROCEED (entry_point_name, reply_procedure, request_id)**

Input **entry_point_name: pmt$program_name**

This parameter indicates the entry point of the module to be loaded. Maximum length is 31 characters.

reply_procedure: ^procedure

This parameter indicates the address of a procedure that returns information about the module to the original calling procedure. Refer to appendix D.

request_id: ^cell

This parameter indicates the address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

Output None.

Remarks The module_use_count is incremented to prevent module deloading.

This procedure makes use of a reply procedure to capture returned information while processing continues.

If the parameter entry_point_found is returned FALSE to the reply procedure, it is the user's responsibility to release the buffer chain returned in error_response.condition.

M_RELEASE

This procedure executes the specified number of message release operations.

Comdeck	**CMIPMLR**
Format	**M_RELEASE (message, count)**
Input	**message: buf_ptr** This parameter indicates the address of the descriptor buffer containing the address of the message whose usage count is to be decremented. **count: 1 .. 32767** This parameter indicates the number of usage counts to be subracted from the message.
Output	**message: buf_ptr** This parameter returns the address of the descriptor buffer containing the address of the message whose usage count has been decremented.
Remarks	This procedure has the same effect as multiple release_message requests.

MAYBE_TASK

This procedure will start a task at the specified entry point and establish a parent task to child task relationship. A task that starts another task using this call becomes the parent task; the new task is referred to as the child task. The Executive sends messages to parent tasks regarding errant child tasks. The Executive, in this case, returns control whether or not the task was started.

Comdeck **CMXMTSK**

Format **MAYBE_TASK (module_ptr, task_attributes, start_at, task)**

Input

module_ptr: dlt$load_id_ptr

This parameter indicates a pointer to the task's module header information.

task_attributes: task_attributes

This parameter indicates a record specifying stack size, priority, and whether or not the task is preemtable.

start_at: ^procedure

This parameter indicates the address of the entry point to the task procedure.

Output

task: task_ptr

If the task started, the task ID is returned. NIL is returned if the procedure fails.

Remarks

NOTE

START_NAMED_TASK_AND_DELAY is the recommended procedure to call for starting a task.

MDU_TO_ASCII

Converts MDU syntax to ASCII.

This routine converts a buffer with management data syntax to a buffer containing an ASCII string. No extra data is added. That is, no extraneous -CR- or -LF- 's are added to the converted data. If they are desired they must already be in the buffer to be converted. Note that data is appended to the receiving buffer. If there is none, set the buffer pointer to NIL first.

The various field types are converted as follows:

binary string: ASCII 0's and 1's
binary octet: converted to hexadecimal ASCII digits
character octets: none (already is ASCII)
binary integer: converted to decimal ASCII digits
binary unsigned integer: converted to decimal ASCII digits
bcd: converted to decimal ASCII digits
format: converted to -LF- / -CR- sequence

Comdeck **MEXM2A**

Format **MDU_TO_ASCII (mdubuf, msgbuf)**

Input **mdubuf: buf_ptr**

This parameter indicates the address of the buffer containing the data to be converted to ASCII format.

msgbuf: buf_ptr

This parameter specifies the buffer to receive the ASCII data.

Output **mdubuf: buf_ptr**

This parameter returns the address of the buffer containing the original data.

msgbuf: buf_ptr

ASCII data is appended to this buffer.

MESSAGE_DEQUEUE

This procedure extracts a message from the specified task-level message queue.

Comdeck CMXPQUE

Format **MESSAGE_DEQUEUE (queue, message, time_interval)**

Input

queue: qcb_ptr

This parameter identifies the queue to be dequeued from.

message: buf_ptr

This parameter indicates the address for the dequeued message.

Output

message: buf_ptr

This parameter returns the address of the dequeued message.

time_interval: integer

This parameter returns the amount of time, in milliseconds (accurate to 100 milliseconds) that the message remained in the queue.

Remarks This is a special high speed dequeuing routine specifically for the use of protocol drivers and interlayer interfaces where data traffic is queued.

MESSAGE_ENQUEUE

This procedure places a message in the specified task-level message queue.

Comdeck CMXPQUE

Format **MESSAGE_ENQUEUE (queue, message)**

Input **queue: qcb_ptr**

This parameter indicates the address of the Queue Control Block of the queue to be enqueued.

message: buf_ptr

This parameter indicates the message to be enqueued.

Output None.

Remarks This is a special high-speed enqueuing routine specifically for the use of protocol drivers and inter-layer interfaces where data traffic is queued.

MODIFY_WRITE_PROTECT_BYTE

This procedure modifies a byte in MPB write-protected RAM. After the field is updated, MPB RAM write-protect is again set.

Comdeck	CMXMWPB
Format	**MODIFY_WRITE_PROTECT_BYTE (^byte, new_value)**
Input	**^byte: ^cell** This parameter indicates the pointer to the write-protected byte to be changed. **new_value: 0 .. 0ff(16)** This parameter indicates the new value for the specified byte.
Output	None.
Remarks	This routine will only work on bytes.

MODIFY_WRITE_PROTECT_LONG_WORD

This procedure modifies a long word in MPB write-protected RAM. After the field is updated, MPB RAM write-protect is again set.

Comdeck	CMXMWPL
Format	**MODIFY_WRITE_PROTECT_LONG_WORD (^long_word, new_value)**
Input	**^long_word: ^cell** This parameter indicates the pointer to the write-protected long word to be changed. **new_value: integer** This parameter indicates the new value for the specified long word.
Output	None.
Remarks	This routine will only work on long words.

MODIFY_WRITE_PROTECT_SHORT_WORD

This procedure modifies a short word in MPB write-protected RAM. After the field is updated, MPB RAM write-protect is again set.

Comdeck	CMXMWPS
Format	**MODIFY_WRITE_PROTECT_SHORT_WORD (^short_word, new_value)**
Input	**^short_word: ^cell** This parameter indicates the pointer to the write-protected short word to be changed. **new_value: 0 .. 0ffff(16)** This parameter indicates the new value for the specified short word.
Output	None.
Remarks	This routine will only work on short words.

NAME_MATCH

This function compares a name string with a model. The name string may contain wild card attributes. If the two strings conform (match), the function returns a TRUE value; otherwise, it returns FALSE.

The following characters have special meaning. These characters may be used in the name string as wild card entries.

[...]	Any single character among those in brackets.
a - z	Within a bracketed group, a range of characters is represented with a dash (-). For example, "a - z", where "a" and "z" are any two characters for which the expression a <= z or a >= z is accepted.
*	Any character string including the NULL string.
?	Any single character.
'	If the model contains any special characters, those special characters (*, [, ?) must be surrounded with single quotes. If the model contains a single quote, 2 single quotes must be in the name. Example: the name string A'*'B matches the model string A*B, and the name string A''B matches the model string A'B.

Comdeck **CSXPNAM**

Format **result := NAME_MATCH (name, model)**

Input **name: string(*)**

This parameter indicates the name to be compared.

model: string(*)

This parameter indicates the model to compare against.

Output **result: boolean**

This parameter returns an indication of whether or not the named string matched the model.

Remarks If a '?' special character is followed by an '*' special character (i.e: ?*) the '*' special character is considered the NULL string.

Special characters are not recgonized within a bracketed group.

NEW_INTERRUPT

This procedure will program the specified vector with the address of a new interrupt service routine. When this service is used, the caller becomes the parent task to the interrupt routine, which is referred to as the child task. The Executive will send the parent messages with workcodes in the range 0..15 regarding errant children. Refer to the appendix F for explanation of these values.

Comdeck **CMXMTSK**

Format **NEW_INTERRUPT (vector, server, task_id)**

Input **vector: 2 .. 255**

This parameter specifies the vector number that will point to the new interrupt service routine. Appendix E lists the interrupt vector numbers.

server: ^procedure

This parameter indicates the address of the procedure that will process the interrupt vector.

Output **task_id: task_ptr**

This parameter returns the address of the task servicing the interrupt service routine. If the interrupt_routine parameter contains an invalid address, either an address error interrupt or a bus error is issued.

NEW_PRIORITY

This procedure changes a task's priority to the requested level.

Comdeck **CMXMTSK**

Format **NEW_PRIORITY (requested_priority, task, status)**

Input **requested_priority: priorities**

This parameter indicates the task's new priority.

task: task_ptr

This parameter indicates the task whose priority is to be changed.

Output **status: boolean**

This parameter returns an indication of whether or not the procedure was successful.

NOPREMPT

This procedure suppresses the task preemption capability.

Comdeck	CMXPPRM
Format	NOPREMPT
Input	None.
Output	None.

OKPREMPT

This procedure restores the task preemption capability.

Comdeck	**CMXPPRM**
Format	**OKPREMPT**
Input	None.
Output	None.

OPEN_STATISTICS_SAP

This procedure allows a software component that is collecting statistics to make itself known to the CDCNET Statistics Manager (CSM). The combination of element_type and element_name defines a SAP entry. Each sds_header specifies a different group.

Comdeck **SMXSAPM**

Format **OPEN_STATISTICS_SAP (element_type, element_name, sds_header_ptr, report_interval, sap_id, status)**

Input

element_type: statistics_type

This parameter indicates the element type for which statistics are to be gathered.

element_name: string (* <= 31)

This parameter indicates the name of the element for which statistics are to be gathered.

sds_header_ptr: ^sds_header

This parameter indicates the pointer to a chain of one or more sds_headers.

report_interval: 1..60 * 60 * 24

This parameter indicates the interval for statistical reporting period. The maximum interval allowed is 24 hours.

Output

sap_id: 0..0ffff(16)

This parameter identifies the opened SAP.

status: open_statistics_status

This parameter returns the call status.

OPEN_STATUS_SAP

This procedure allows a software component to register the address of its status table by opening a software status SAP. A software component may call the OPEN_STATUS_SAP routine after it is initialized and is capable of reporting status.

Comdeck **SDXSSAR**

Format **OPEN_STATUS_SAP (name, task_id, sap_table_ptr, sap_number, status)**

Input **name: string (* <= 31)**

This parameter indicates the module name of the software component for which a SAP is to be opened.

task_id: task_ptr

This parameter identifies the task_id of the software component who will open the software status SAP.

sap_table_ptr: ^cell

This parameter identifies the address of the software components status table.

Output **sap_number: software_sap_range**

This parameter uniquely identifies the status SAP opened. The sap_number must be used when later closing a status SAP.

status: access_status_type

This parameter indicates if the SAP requested was opened. If the SAP was not opened, try again, but be warned that memory is low.

Remarks The procedure NOPREMPT is called when entering OPEN_STATUS_SAP to suppress task preemption. OPEN_STATUS_SAP is exited in a non-preemptable state and will require the caller to make a call to the procedure OKPREMPT if preemptability is so desired.

Global data modified:

software_status_sap_table

PCOPY

This procedure physically copies a message to a new buffer chain, and releases the old set of buffers. Data is compact in the new buffers; the first (n-1) buffers are full, and the last one has all of its empty space in the trailing portion of the buffer.

Comdeck **CMXPCPY**

Format **PCOPY (message, threshold, success)**

Input

message: buf_ptr

This parameter indicates the address of the message to be copied.

threshold: threshold_size

This parameter indicates the priority for obtaining buffers.

Output

message: buf_ptr

This parameter returns the address of the copied message.

success: boolean

This parameter is TRUE if a new buffer chain was successfully obtained and FALSE if a new buffer chain is not currently available.

Remarks This time-consuming operation requires at least 3-5 microseconds per byte copied. It is recommended that the caller either run at a relatively low task priority, or yield control sometime after the routine returns to avoid time slice overrun, and to permit other processes to be active.

PICK

This procedure removes a structure from the specified tree, according to the key provided, and returns the associated data entry, or NIL (if no matching entry is found).

Comdeck CMXPPIC

Format **address := PICK (head, key)**

Input **head: ^root**

This parameter indicates the root of the tree to be picked from.

key: integer

This parameter indicates the key of the node to be picked from the tree.

Output **address: ^cell**

This parameter returns the address of the structure associated with the key.

PMP_GET_DATE

This procedure returns the current date in the specified format.

Comdeck **PMXGDAT**

Format **PMP_GET_DATE (format, date, date_str_len)**

Input **format: ost$date_formats**

This parameter specifies the format in which the date will be returned. Valid specifications are:

Specify	Format	Example
osc$month_date	month DD, YYYY	June 21, 1986
osc$mdy_date	MM/DD/YY	06/21/86
osc$iso_date	YYYY-MM-DD	1986-21-06
osc$ordinal_date	YYYYDDD	1986172
osc$dmy_date	DD/MM/YY	21/06/86
osc$default_date	Installation specified.	

Output **date: ost$date**

This parameter returns the current date.

date_str_len: 1 .. 18

This parameter returns the length of the date parameter.

PMP_GET_TIME

The procedure returns the current time of day in the specified format.

Comdeck	**PMXGTIM**
Format	**PMP_GET_TIME (format, time, time_str_len)**

Input

format: ost$time_formats

This parameter specifies the format in which the time will be returned. Valid specifications are:

Specify	Format	Example
osc$ampm_time	HH:MM AM or PM	1:15 PM
osc$hms_time	HH:MM:SS	13:15:21
osc$millisecond_ time	HH:MM:SS:MMM	13:15:21:453
osc$default_time	Installation specified.	

Output

time_str: ost$time

This parameter returns the current time.

time_str_len: 1 .. 12

This parameter returns the length of the time_str parameter.

PREFIX

This procedure adds a header to the front of a message. The header is back filled to facilitate insertion of the next header (to be prefixed) in the same buffer.

Comdeck CMXPPRE

Format **PREFIX (size, address, message, threshold, allocation_type, success)**

Input

size: non_empty_message_size

This parameter indicates the size of the header to be prefixed.

address: ^cell

This parameter indicates the address of the header to be prefixed.

message: buf_ptr

This parameter indicates the message to which the header will be prefixed.

threshold: threshold_size

This parameter specifies the buffer threshold to use.

allocation_type: pref_type

This parameter indicates how the buffer allocations are to be performed internally by PREFIX. See the description of pref_type in appendix C.

Output

message: buf_ptr

This parameter returns the message to which the header has been prefixed.

success: boolean

This parameter indicates whether the call was successful.

Remarks If a conditional request is made (i.e. preference type = conditional) and that buffer request is not satisfied, then a failure status is returned.

PUT_STATUS_RECORD

This procedure updates the status record for the device name specified.

Comdeck	**SDXGPSR**
Format	**PUT_STATUS_RECORD (device_status_record, owner_key, status)**
Input	**device_status_record: component_status_type** This parameter indicates the status record to be updated. Just the status field will be updated via this routine. **owner_key: ^cell** This parameter indicates the address of the configuration table that was provided on the REQUEST_HARDWARE_DEVICE call.
Output	**status: boolean** If the status record was updated, TRUE is returned; FALSE otherwise.
Remarks	Global data modified: major_card_status_table lim_status_table port_status_table(s) smm_bank_status_table(s) pmm_bank_status_table

QUEUE_IP_COMMAND

This DVM procedure is used by a stream service routine or TIP to queue commands to the DVCB.

Comdeck **DMXQS**

Format **QUEUE_IP_COMMAND (dvm_id, ip_cmd_ptr, status)**

Input **dvm_id: ^cell**

This parameter indicates the address of the DVMID.

ip_cmd_ptr: ^cim_cmd_pkt_rec_type

This parameter indicates the address of the command packet being queued.

Output **status: 0..0ffff(16)**

This parameter returns the status of the QUEUE_IP_COMMAND call. The status codes are:

0: command accepted.
1: invalid DVMID supplied.
2: no command queue entries available (after retrying for 0.5 seconds).

READ_BCD_CLOCK

This procedure reads the realtime clock.

Comdeck	CMXMTIM
Format	READ_BCD_CLOCK (**the_time**)
Input	None.
Output	**the_time: ^bcd_time** This parameter returns the time in BCD format, as read from the realtime clock.

READ_CLOCK

This procedure reads the binary clock to millisecond accuracy.

Comdeck CMXMTIM

Format **READ_CLOCK (the_time)**

Input None.

Output **the_time: integer**

This parameter returns the time, as read from the binary clock.

READ_SR0_FROM_IP

DVM diagnostic services use this procedure to read a status register from an IP.

Comdeck DMXDIAG

Format **READ_SR0_FROM_IP (sr, ip_number, reg_number, status)**

Input **ip_number: 0..7**

This parameter indicates the slot number of the IP.

reg_number: 0..3

This parameter indicates the status register number.

Output **sr: diag_ip_sr_type**

This parameter returns a packed record containing the register value.

status: 0..0ffff(16)

This parameter returns the status of READ_SR0_FROM_IP. The status codes are:

0: command accepted.
1: invalid IP slot number.
2: IP board is active.
3: invalid command register number.
4: bus error received during ICB access.

RELEASE_HARDWARE_DEVICE

This procedure allows the owner of a previously configured hardware device to release, or free, the device. It requires its caller to provide the owner's key as an input parameter.

The table address in the associated System Status Table (SST) is set to NIL and the hardware device status is set to not_configured if the following conditions are met:

- A valid device name is specified
- The associated board type is physically available in the slot specified by the device name
- The state of the device is not OFF
- Its status is **configured** and
- The owner key matches the configuration address specified on the request.

Comdeck **SDXAHWD**

Format **RELEASE_HARDWARE_DEVICE (device_name, owner_key, status)**

Input **device_name: string (maximum_device_name_size)**

This parameter indicates the name of the hardware device to be released.

owner_key: ^cell

This parameter indicates the address of the owner's configuration table specified on the REQUEST_HARDWARE_DEVICE call.

Output **status: access_device_status_type**

This parameter indicates whether the device was successfully released.

Remarks Global data modified:

major_card_status_table
port_status_table(s)

RELEASE_MESSAGE, FG_RELEASE_MESSAGE

These procedures release chains of data buffers, with the following effects:

RELEASE_MESSAGE	The buffer(s) are released.
FG_RELEASE_MESSAGE	For interrupt routine use only; the buffer(s) are released.

Comdeck **CMXPRLB**

Format **RELEASE_MESSAGE (message)**
FG_RELEASE_MESSAGE (message)

Input **message: buf_ptr**

This parameter indicates the address of the message to be released.

Output **message: buf_ptr**

This parameter returns NIL.

REQUEST_DIAGNOSTIC_ENTRY

This procedure obtains the address of the SST entry for the specified device. If a valid device name is specified and the associated board type is physically available, then the address of the associated SST is returned.

Comdeck DGXAHWD

Format **REQUEST_DIAGNOSTIC_ENTRY (device_name, kind_of_status_table, system_status_table_ptr, status)**

Input **device_name: string (maximum_device_name_size)**

This parameter indicates the name of the hardware device whose SST entry is being requested.

Output **kind_of_status_table: system_status_table_type**

This parameter returns the type of SST the system_status_table_ptr points to.

system_status_table_ptr: ^cell

This parameter returns the address of the SST associated with the specified device.

status: boolean

This parameter indicates if the address of the named device's status table was returned (TRUE); FALSE if the named device is not available in the DI.

REQUEST_HARDWARE_DEVICE

This procedure configures the specified hardware device. The parameters for the request routine identify the device being requested and provide an owner's key to be associated with the device while it is owned by the caller.

If the following conditions are met, then the caller's configuration table address is placed in the associated SST and the device status is set to configured:

A valid device name is specified

The associated board type is physically available in the slot specified by the device name

The state of the device is ON

Device status is initially **not_configured**

Comdeck **SDXAHWD**

Format **REQUEST_HARDWARE_DEVICE (device_name, port_owner, cnfg_table_ptr, status)**

Input **device_name: string (maximum_device_name_size)**

This parameter indicates the name of the hardware device to be configured.

port_owner: port_owner_type

This parameter indicates the port owner type. This parameter is only required when a port is requested; if a device other than a port is requested, this parameter can be ignored.

cnfg_table_ptr: ^cell

This parameter indicates the address of the caller's configuration table.

Output **status: access_device_status_type**

This parameter indicates if the device was successfully configured. If the device was not configured, this parameter identifies the reason why.

Remarks Global data modified:

major_card_status_table
port_status_table(s)

RESET_DI

This procedure resets the DI.

Comdeck	**CMXRDI**
Format	**RESET_DI (software_reset_code)**
Input	**software_reset_code: integer** This parameter indicates the software error that forced the reset. Refer to appendix G.
Output	None.

RESET_RECOVERY_PROCEDURE

This procedure removes the specified recovery procedure from the recovery stack. It is used at the exit of any procedure that calls SET_RECOVERY_PROCEDURE.

Comdeck CMISISA

Format **RESET_RECOVERY_PROCEDURE (procedure_address)**

Input **procedure_address: ^procedure**

This parameter indicates the pointer to the local recovery procedure.

Output None.

RESTART_PORT_SERVICE

DVM users use this procedure to restart service for a particular port after reload of a peripheral board.

Comdeck	**DMXXPS**
Format	**RESTART_PORT_SERVICE (sps_parm_ptr, dvm_id, status)**

Input

sps_parm_ptr: ^sps_parm_type

This parameter indicates the address of the START_PORT_SERVICE parameter block.

dvm_id: ^cell

This parameter indicates the address of the specified DVMID.

Output

status: 0..0ffff(16)

This parameter returns the status of RESTART_PORT_SERVICE. The status codes are:

0: command accepted.
1: invalid peripheral board slot specified.
2: device service not started for specified peripheral board.
3: service already started for requested port.
4: invalid DVMID supplied.
5: board reload is in progress.

RESTORE_DVM_INTERRUPT

DVM diagnostic services use this procedure to change the new interrupt routine to the original interrupt routine specified before CHANGE_DVM_INTERRUPT executed.

Comdeck **DMXDIAG**

Format **RESTORE_DVM_INTERRUPT (ip_number, status)**

Input **ip_number: 0..upper_ip_num**

This parameter indicates the slot number of the IP.

Output **status: 0..0ffff(16)**

This parameter returns the status of RESTORE_DVM_INTERRUPT. The status codes are:

0: command accepted.
1: invalid peripheral slot number.

RESTORE_TASK

This procedure restores a task from a suspended state that it entered either with SUSPEND or ABORT_TASK.

Comdeck	**CMXMTSK**
Format	**RESTORE_TASK (task_id, status)**
Input	**task_id: task_ptr** This parameter indicates the address of the task to be restored.
Output	**status: boolean** This parameter indicates whether the task was restored.

SEND_CR0_TO_IP

DVM Diagnostic Services use this procedure to send a command register to a peripheral. It will not execute when START_DEVICE_SERVICE is active for the peripheral board.

Comdeck **DMXDIAG**

Format **SEND_CR0_TO_IP (cr, ip_number, reg_number, status)**

Input **cr: diag_ip_cr_type**

This parameter specifies a packed record containing the command register.

ip_number: 0..7

This parameter indicates the slot number of the peripheral board.

reg_number: 0..3 This parameter indicates the command register number.

Output **status: 0..0ffff(16)**

This parameter returns the status of SEND_CR0_TO_IP. The status codes are:

0: command accepted.
1: invalid peripheral board slot number.
2: peripheral board is active.
3: invalid command register number.
4: bus error received during ICB access.

SEND_EXPRESS, FG_TO_EXPRESS

These procedures send ITMs to express queues in first-in, first-out (FIFO) order. If the task is waiting for a message on this queue, the message is copied directly to the waiting task's data space. These calls have the following effects:

SEND_EXPRESS	Message is sent to target task.
FG_TO_EXPRESS	For interrupt routine use only; message is sent to target task.

Comdeck **CMXMTSK**

Format **SEND_EXPRESS (size, message, target, status)**
FG_TO_EXPRESS (size, message, target, status)

Input

size: 1 .. 32767

This parameter indicates the size of the message to be sent.

message: ^cell

This parameter indicates the address of the ITM to be sent.

target: task_ptr

This parameter indicates the address of the task to which the ITM is to be sent.

Output

status: boolean

This parameter returns TRUE if the ITM was sent; FALSE if the target task address was invalid.

SEND_NORMAL, FG_TO_NORMAL

These procedures send ITMs to normal queues. If the task is waiting for a message on this queue, the message is copied directly to the waiting task's data space. These calls have the following effects:

SEND_NORMAL	Message is sent to target task.
FG_TO_NORMAL	For interrupt routine use only; message is sent to target task.

Comdeck CMXMTSK

Format **SEND_NORMAL (size, message, target, status)**
FG_TO_NORMAL (size, message, target, status)

Input **size: 1 .. 32767**

This parameter indicates the size of the message to be sent.

message: ^cell

This parameter indicates the address of the ITM to be sent.

target: task_ptr

This parameter indicates the address of the task to which the ITM is to be sent.

Output **status: boolean**

This parameter returns TRUE if the ITM was sent; FALSE if the target task address was invalid.

SET_BCD_CLOCK

This procedure sets the realtime clock to the specified time.

Comdeck	CMXMTIM
Format	**SET_BCD_CLOCK (the_time)**
Input	**the_time: ^bcd_time** This parameter indicates the time of day in BCD format.
Output	None.

SET_BUFFER_CHAIN_OWNER

This procedure places an owner identification into the executive allocation field of each descriptor and data buffer in the buffer chain. This is provided so memory and buffer audit may be accomplished meaningfully.

Comdeck	CSXSBCO
Format	**SET_BUFFER_CHAIN_OWNER (buffer_address, owner_id)**
Input	**buffer_address: buf_ptr** This parameter indicates the buffer chain whose ownership is to be set. **owner_id: memory_owner_type** This parameter indicates the buffer chain owner.
Output	None.

SET_MEMORY_OWNER

This procedure sets an owner identification into the allocator id field of the header of the specified memory location.

Comdeck	**CSXSMO**
Format	**SET_MEMORY_OWNER (memory_address, owner_id)**
Input	**memory_address: ^cell** This parameter indicates the memory location whose ownership is to be set. **owner_id: memory_owner_type** This parameter indicates the memory location owner.
Output	None.

SET_RECOVERY_PROCEDURE

This procedure pushes the specified recovery block onto the recovery stack of the calling task.

Comdeck CMISISA

Format **SET_RECOVERY_PROCEDURE (recovery_block, procedure_address)**

Input **recovery_block: sat$recovery_block**

This parameter indicates the empty recovery block to be pushed onto the task's recovery stack.

procedure_address: ^procedure

This parameter indicates the caller's recovery routine.

Output **recovery_block: sat$recovery_block**

This parameter returns the initialized recovery block that has been pushed onto the task's recovery stack.

SET_WRITE_PROTECT

This procedure sets the write protect flag. The proper use of this routine is in conjunction with CLEAR_WRITE_PROTECT. Refer to CLEAR_WRITE_PROTECT for more information.

Comdeck	CMISWP
Format	SET_WRITE_PROTECT
Input	None.
Output	None.

SFIND

This procedure finds and returns the table in a tree structure that is associated with the specified ASCII key.

Comdeck	**CMXPFIN**
Format	**address := SFIND (head, key)**
Input	**head: ^root** This parameter indicates the root address of the tree table access structure to be searched. **key: ^string (*)** This parameter indicates the key to be searched for.
Output	**address: ^ cell** This parameter returns the address of the table associated with the provided key. If no such table is found, NIL is returned.
Remarks	See GROW.

SFIND_FIRST

This procedure visits each node of the specified string keyed tree structure in left-node-right order and calls the user-supplied procedure. The search continues until the user-supplied procedure returns a value of TRUE in the field, bool. See also SFIND_NEXT, FIND_FIRST, and FIND_NEXT.

Comdeck CMXPFNF

Format **table = SFIND_FIRST (head, key, qual, param)**

Input

head: ^root

This parameter indicates the address of the head root of the tree structure to be searched.

qual: ^procedure

This parameter indicates the address of the test function to be called. If NIL, then SFIND_FIRST will return the smallest key in the tree structure. Refer to appendix D.

param: ^cell

This parameter indicates the parameter to pass to qual.

Output

key: ^string (*)

This parameter returns the smallest key, associated with a table entry, that satisfies the procedure qual.

table: ^cell

This parameter returns the entry in the tree associated with the first key that has satisfied the procedure qual.

SFIND_NEXT

This procedure searches the specified string tree structure in left-node-right order for a node that contains a key greater than the input parameter, key. For each node that satisfies this condition, SFIND_NEXT calls the user-supplied procedure. The search continues until the user-supplied procedure returns a value of TRUE in the field, bool. See also SFIND_FIRST, FIND_FIRST, and FIND_NEXT.

Comdeck **CMXPFNX**

Format **table = SFIND_NEXT (head, key, qual, param)**

Input

head: ^root

This parameter indicates the root of the tree structure to be searched.

key: ^string (*)

This parameter indicates the key to be searched for.

qual: ^procedure

This parameter indicates the address of the test function to be called. If NIL, then SFIND_FIRST will return the key that is the next largest compared with the input parameter key. Refer to appendix D.

param: ^cell

This parameter indicates the parameter to pass to qual.

Output

key: ^string (*)

This parameter returns the first key that satisfies the procedure qual.

table: ^cell

This parameter returns the entry in the tree associated with the first key that has satisfied the procedure.

SFIND_WILD_CARDS

This procedure locates wild card matches in string-keyed tree structures and terminates when all nodes have been processed or when the user supplied routine returns a value of TRUE in the quit_processing parameter.

Comdeck CSXPFWC

Format **SFIND_WILD_CARDS (head, key, process_match, params)**

Input

head: ^root

This parameter indicates the pointer to root of the tree.

key: ^string (* <= max_name_size)

This parameter indicates the pointer to wild card key.

process_match: ^procedure

This parameter indicates the supplied routine. Refer to appendix D.

params: ^cell

This parameter indicates the pointer to parameter list.

Output None.

SGROW

This procedure adds a new table to the specified string-keyed tree structure. If an association already exists between the provided key and a table structure, the associated table is returned and no update is performed. Otherwise, the association is created, and NIL is returned.

Comdeck CMXPGRO

Format **address := SGROW (head, key, table, size)**

Input

head: ^root

This parameter indicates the root of the tree to be added on to.

key: ^string (*)

This parameter indicates the key to be associated with the new table for searching operations.

table: ^cell

This parameter indicates the table to be added to the tree.

size: integer

This parameter indicates the size of the table to be added, in bytes.

Output

address: ^cell

This parameter returns the location of the table in the tree table access structure.

Remarks See GROW.

SIGNAL(n)/ACQUIRE(n)

These inline procedures test for, and set, semaphores, permitting multiple processor acquisition of data structures in a controlled manner. Multiple semaphores can be acquired at once; however, if the Executive fails to acquire one or more of the semaphores, then none of the semaphores are acquired.

Resources must be acquired in this manner, but may be released simply by storing a zero in the address specified on the ACQUIRE/SIGNAL call. The executive clears the entire byte when it releases the resources.

These calls have the following effects:

SIGNAL(n)	The resources are acquired, or a failure is returned.
ACQUIRE(n)	Control returns when the resource list is entirely acquired.

Comdeck **CMXMTSK**

Format **SIGNAL(n) (address(n), status)**
ACQUIRE(n) (address(n), status)
where n is an integer in the range 1..4.

Input **address(n): ^cell**

This parameter indicates the address to be tested for semaphore, or set.

Output **status: boolean**

This parameter returns an indication of whether the call completed successfully.

Remarks If an ACQUIRE(n) request is used by a non-preemptible task and the resource is not available, the entire system is halted.

SPICK

This procedure removes a structure from the specified string-keyed tree and returns the associated data entry, or NIL.

Comdeck	**CMXPPIC**
Format	**address:= SPICK (head, key)**
Input	**head: ^root** This parameter indicates the root of the source tree. **key: ^string (*)** This parameter indicates the key associated with the structure to be picked from the tree.
Output	**address: ^cell** This parameter returns the table associated with the specified key in the specified tree structure.

START_DEVICE_SERVICE

This procedure initializes a peripheral board. The CIM Monitor Command Processor allocates and initializes a DVCB specific to the peripheral, and boots the peripheral into operation.

Comdeck **DMXXDS**

Format **START_DEVICE_SERVICE (sds_parm_ptr, status)**

Input **sds_parm_ptr: ^sds_parm_type**

This parameter indicates the address of the START_DEVICE_SERVICE parameter block. This record specifies the peripheral board slot number, number of parts, number of buffers, status and command queue sizes, and address and length of the module to be loaded.

Output **status: 0..0ffff(16)**

This parameter returns the status of START_DEVICE_SERVICE call. The status codes are:

0: command accepted.
1: invalid peripheral board slot was specified.
2: service has already been started for the specified peripheral.
3: no SMM memory is available for the DVCB.
4: peripheral is not available, or is busy.
5: unable to notify peripheral of DVCB location.
6: unable to move code block for peripheral.
7: unable to cause peripheral to execute.

START_NAMED_TASK_AND_DELAY

Given an entry point name, this procedure starts the appropriate task, loading the module(s) first, if necessary. The caller waits while the request is processed.

Comdeck	**DLXSNTK**
Format	**START_NAMED_TASK_AND_DELAY (entry_point_name, task_started, task_id, error_response)**
Input	**entry_point_name: pmt$program_name** This parameter indicates the entry point name of the task to be started.
Output	**task_started: boolean** This parameter returns an indication of whether the task was started. **task_id: task_ptr** This parameter returns the task id of the started task. **error_response: clt$status** This parameter returns any error messages generated by the Online Loader.
Remarks	The module use count is incremented to prevent module deloading. If the parameter task_started is returned FALSE, it is the user's responsibility to release the buffer chain returned in error_response.condition.

START_NAMED_TASK_AND_PROCEED

Given an entry point name, this procedure starts the appropriate task, loading the module(s) first, if necessary. The calling task is allowed to continue work during loading.

Comdeck **DLXSNTK**

Format **START_NAMED_TASK_AND_PROCEED (entry_point_name, reply_procedure, request_id)**

Input

entry_point_name: pmt$program_name

This parameter indicates the entry point name of the task to be started.

reply_procedure: ^procedure

This parameter indicates the address of a procedure that returns information about the module to the original calling procedure. Refer to appendix D.

request_id: ^cell

This parameter indicates the address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

Output None.

Remarks The module use count is incremented to prevent module deloading.

This procedure makes use of a reply procedure to capture returned information while processing continues.

If the parameter task_started is returned FALSE, it is the user's responsibility to release the buffer chain returned in error_ response.condition.

START_PORT_SERVICE

This DVM procedure establishes service to a particular port. It allocates and initializes the port control block (DVMID) specific to the LIM port.

Comdeck	**DMXXPS**
Format	**START_PORT_SERVICE (sps_parm_ptr, dvm_id, status)**
Input	**sps_parm_ptr: ^sps_parm_type** This parameter indicates the address of the START_PORT_SERVICE parameter block. This record specifies the LIM port number, slot number of the board, and the address of the DVM user.
Output	**dvm_id: ^cell** This parameter returns the address of the DVMID specific to the LIM port. **status: 0..0ffff(16)** This parameter returns the status of START_PORT_SERVICE. The status codes are: 0: command accepted. 1: invalid peripheral board slot number specified. 2: device service not started for specified peripheral. 3: service already started for specified port. 4: no SMM memory available for DVMID block. 5: invalid DVMID supplied (DVM software problem).

START_SYSTEM_TASK

This procedure starts a task with the system ancestor as its parent. A reply procedure is used to communicate with the caller.

Comdeck CMXSISA

Format **START_SYSTEM_TASK (transfer_address, priority, stack_size, reply_procedure, request_id)**

Input

transfer_address: ^procedure

This parameter indicates the task entry point of the task to be started.

priority: priorities

This parameter indicates task priority of the task to be started.

stack_size: stack_size

This parameter indicates the stack size of the system task to be started.

reply_procedure: ^procedure

This parameter provides procedure linkage. Refer to appendix D.

request_id: ^cell

This parameter indicates the user request identifier to link request and response.

Output None.

Remarks The supplied reply procedure should have minimal functionality since it executes under the system ancestor task.

START_NAMED_TASK_AND_DELAY is the recommended procedure to call for starting a task. It will look up the entry point and call START_SYSTEM_TASK, which in turn calls MAYBE_TASK.

START_TASK

This procedure starts a task at a procedure entry point. A task that starts another task using this call is referred to as a parent task; the new task is referred to as the child task. The Executive will send the parent messages with workcodes in the range 0..15 regarding errant child tasks. Refer to appendix F for explanations of these values.

Comdeck **CMXMTSK**

Format **START_TASK (module_ptr, task_attr, lex_level_zero_xdcl, task)**

Input

module_ptr: dlt$load_id_ptr

This parameter indicates the pointer to be placed into the TCB for the task.

task_attr: task_attributes

This parameter indicates the pointer to the task attributes of the task to be started.

lex_level_zero_xdcl: ^procedure

This parameter indicates the address of the entry point to the task procedure.

Output

task: task_ptr

This parameter returns the task id of the started task.

Remarks

NOTE

START_NAMED_TASK_AND_DELAY is the recommended procedure to call for starting a task. It will look up the entry point and call START_SYSTEM_TASK, which in turn calls MAYBE_TASK.

STOP_DEVICE_SERVICE

This DVM procedure stops execution of a particular intelligent peripheral. The CIM Monitor Command Processor disables the line to the peripheral and then deallocates (releases) the DVCB specific to the peripheral.

Comdeck	**DMXXDS**
Format	**STOP_DEVICE_SERVICE (qds_parm_ptr, status)**
Input	**qds_parm_ptr: ^qds_parm_type** This parameter indicates the address of the QUIT_DEVICE_SERVICE parameter block. This record specifies the slot number assigned to the particular peripheral board.
Output	**status: 0 .. 0ffff(16)** This parameter returns the status of STOP_DEVICE_SERVICE. The status codes are: 0: command accepted. 1: invalid peripheral board slot number has been specified. 2: service has not been started for the specified peripheral board.

STOP_PORT_SERVICE

This procedure stops service for a particular port used by stream service routines or TIPs. It deallocates the port control block (DVMID) specific to the LIM port.

Comdeck	**DMXXPS**
Format	**STOP_PORT_SERVICE (qps_parm_ptr, status)**
Input	**qps_parm_ptr: qps_parm_type** This parameter indicates the address of the DVMID specific to the LIM port.
Output	**status: 0..0ffff(16)** This parameter returns the status of STOP_PORT_SERVICE. The status codes are: 0: command accepted. 1: invalid DVMID supplied.

STOP_TASK

This procedure stops a task and permanently removes it from the system.

Comdeck	**CMXMTSK**
Format	**STOP_TASK (task, status)**
Input	**task: task_ptr** This parameter indicates the task to be stopped.
Output	**status: boolean** This parameter returns an indication of whether the call was successful.
Remarks	The module use count for the specified task is decremented.

STRIP

This procedure removes a header from the front of the specified message.

STRIP differs from STRIP_IN_PLACE in that the passed user space is always used and no attempt is made to not move the stripped header space (in other words, data is always moved). STRIP_IN_PLACE calls STRIP if data movement is required.

Comdeck **CMXPSTR**

Format **STRIP (hdr_size, address, msg, threshold)**

Input

hdr_size: non_empty_message_size

This parameter indicates the size of the header to be stripped.

address: ^cell

This parameter indicates the location for movement of stripped space.

msg: buf_ptr

This parameter indicates the message whose header is to be stripped.

threshold: threshold_size

This parameter specifies the threshold for buffer acquisition.

Output

msg: buf_ptr

This parameter returns the message whose header has been stripped.

STRIP_IN_PLACE

This procedure returns the header address (without moving it, if possible) and logically removes the header from the message. If the header is contained in one buffer, is not multiply used, and begins on an even byte boundary, the header address is returned and the offset is changed to remove the header. Otherwise, STRIP is called to move the header to the user's area.

Comdeck CMXPSIP

Format **STRIP_IN_PLACE (hdr_size, address, table, message, threshold)**

Input

hdr_size: non_empty_message_size

This parameter indicates the size of the header to be stripped.

address: ^cell

This parameter indicates the location for movement of stripped space if STRIP must be called.

message: buf_ptr

This parameter indicates the message whose header is to be stripped.

threshold: threshold_size

This parameter specifies the threshold for buffer acquisition.

Output

table: ^cell

This parameter returns a pointer to the stripped data, on an even byte boundary.

message: buf_ptr

This parameter points to the message whose header has been stripped.

SUBFIELD

This procedure obtains multiple-byte header field(s) and copies them to the specified text area.

Comdeck **CMXPSUB**

Format **SUBFIELD (displacement, length, text, message)**

Input

displacement: message_size

This parameter indicates an offset within the message from which to begin copying data.

length: non_empty_message_size

This parameter indicates the number of bytes in the subfield.

text: ^cell

This parameter indicates the address that the subfield is copied to.

message: buf_ptr

This parameter indicates the address of the message to copy from.

Output

length: non_empty_message_size

This parameter returns the number of bytes in subfield.

Remarks The message parameter may not be equal to NIL. It must be a valid descriptor buffer address.

SUSPEND

This procedure suspends a task without notifying the parent task. SUSPEND is similar to ABORT_TASK, and is restored with the same call, but is intended for use by another task that wishes to take matters into its own hands.

Comdeck	**CMXMTSK**
Format	**SUSPEND (task, status)**
Input	**task: task_ptr** This parameter indicates the task to be suspended.
Output	**status: boolean** This parameter indicates whether the call was successful.

TIME

This procedure converts time_of_day or intervals to milliseconds. For example, the time 1:53:22 PM is input as (13,53,22); an interval of 10 seconds is input as (0,0,10).

Comdeck CMXMTIM

Format **mil_time := TIME (hour, minute, second)**

Input **hour: 0 .. 24**

This parameter indicates the time-of-day or interval hours.

minute: 0 .. 59

This parameter indicates the time-of-day or interval minutes.

second: 0 .. 59

This parameter indicates the time-of-day or interval seconds.

Output **mil_time: milliseconds**

This parameter returns the time-of-day or interval expressed in milliseconds since midnight.

Remarks Midnight is input either as (0,0,0) or (24,0,0).

TRANSLATE_MESSAGE

This procedure is used for character set translation, such as EBCDIC to ASCII, or ASCII to Baudot. The translation table provides a mapping of the 'from' character set to the 'to' character set.

Comdeck **CMXPTRA**

Format **TRANSLATE_MESSAGE (message, table, threshold)**

Input **message: buf_ptr**

This parameter indicates the message to be translated.

table: string (256)

This parameter indicates the translation table.

threshold: threshold_size

This parameter indicates the buffer allocation threshold.

Output **message: buf_ptr**

This parameter returns the translated message.

Remarks The addresses for message and table must be valid. The table parameter will normally specify a read-only static data structure.

This is a highly time consuming operation, requiring a minimum of 5 microseconds per character translated. It is recommended that the caller yield control sometime after returning to avoid time slice overrun.

TRIM

This procedure trims the specified number of bytes from the end of the message. Any buffers emptied as a result are returned to the free buffer pool. If the entire message is less than the requested size, the caller is informed that there are not enough bytes to satisfy the request and DEAD_STOP is called.

Comdeck CMXPTRI

Format **TRIM (size, address, message, threshold)**

Input

size: non_empty_message_size

This parameter indicates the number of bytes needed. If size is NULL, nothing is done and the return is immediate.

address: ^cell

This parameter specifies where to position the bytes that are retrieved.

message: buf_ptr

This parameter indicates the message to be trimmed.

threshold: threshold_size

This parameter specifies the threshold for buffer acquisition.

Output

message: buf_ptr

This parameter returns the first data descriptor.

UNUSED_STACK_

This function returns the number of reserved user stack bytes that have not been used since the specified task was first started.

Comdeck **CMXMTSK**

Format **size:= UNUSED_STACK_ (task_id)**

Input **task_id: task_ptr**

This parameter indicates the task whose user stack area is to be checked.

Output **size: integer**

The return indicates the number of reserved user stack bytes that have not been used since the specified task was first started.

Zero is returned if the amount of unused space has not changed since the last check was made. This response can be used to detect a new minimum stack area. The minimum unused stack space is maintained in the task control block (TCB) at TCBSPACE.

A negative response is returned for any of the following conditions:

- The indicated stack has overflowed its reserved stack area.
- The task_id is not a valid task pointer.
- The task_id is NIL and there is no running task.

A negative response indicates an illogical software condition that requires action from the caller to recover or abort the task. The TASK_OVERFLOWED field in the system configuration table will contain a pointer to the task that returned the negative response.

VALIDATE_SECTION_ADDRESS

This procedure translates a given address into a module name and a section address with offset. If the specified range is not found, or the address is invalid for the MPB, the output parameter valid_section is set to FALSE.

Comdeck **DLXVSA**

Format **VALIDATE_SECTION_ADDRESS (address, valid_section, module_name, section_address, offset)**

Input

address: ^cell

This parameter indicates the address to be translated.

Output

valid_section: boolean

This parameter returns an indication of whether the address to be translated is valid for the MPB.

module_name: pmt$program_name

This parameter returns the associated module name of the user-specified address.

section_address: ^dlt$section

This parameter returns the associated section address of the user-specified address.

offset: llt$section_offset

This parameter returns the associated offset-within-section of the user-specified address.

VISIT_ALL_NODES

This procedure steps through a tree one node at a time, allowing the caller to process information at each node using a user supplied routine.

The user supplied routine has three parameters: a pointer to the first associated table, a pointer to cell (user parameters to be passed through to the process routine), and a boolean value. Stepping through the tree will continue until all elements in the tree have been exhausted or the boolean value returned in the user supplied routine is FALSE.

Comdeck CMXPVAL

Format VISIT_ALL_NODES (ptr, process, param)

Input

ptr: ^node

This parameter indicates the pointer to current node.

process: ^procedure

This parameter indicates the user provided procedure. Refer to appendix D.

param: ^cell

This parameter indicates the pointer to user parameters.

Output None.

Remarks

WARNING

User should not modify a tree structure while VISIT_ALL_NODES is executing.

WAIT

This procedure puts the executing task to sleep until a WAKEUP is received for the task. This is similar to SEND_MESSAGE, where the message content is void. It could be used where a task wishes to wait for an interrupt routine or other task accomplishes something before looking at its ITM queues again.

Comdeck	CMXMTSK
Format	WAIT
Input	None.
Output	None.

WAKE_UP, FG_WAKE_UP

This procedure wakes up a waiting task. If the task has executed a WAIT call, it is scheduled. If not, a flag is set indicating that the next WAIT call is to be treated as a YIELD.

These calls have the following effects:

WAKE_UP	The task is awakened.
FG_WAKE_UP	For interrupt routine use only; the task is awakened.

Comdeck CMXMTSK

Format **WAKE_UP (task, status)**
FG_WAKE_UP (task, status)

Input **task: task_ptr**

This parameter indicates the task to be awakened.

Output **status: boolean**

This parameter returns an indication of whether the call was successful.

YIELD

This procedure yields control of the CPU by allowing tasks on the same or higher priority to execute first; the task that yields is placed at the end of the queue for tasks executing at its priority. This is useful for giving up control of the CPU to other tasks.

Comdeck	CMXMTSK
Format	YIELD
Input	None.
Output	None.

Appendixes

Glossary A

B

Base System Software

CDCNET software that works to initialize the device interface and maintain it during operation. Base System Software also provides a set of common routines for use by other CDCNET software.

Boot File

A file that contains the basic set of software that is loaded into a device when the device interface requests to be loaded. A boot file brings a device interface up to a basic operational state. Further definition of the device interface's functions is provided by its system configuration file.

Bootstrap

A technique or device designed to bring itself into a desired state by means of its own action. For example, a machine routine whose first few instructions are sufficient to bring the rest of itself into the computer from an input device.

Buffer

One of two structures for the storage of data in device interface memory. Refer to Data Buffer and Descriptor Buffer.

C

CDCNET

Refer to Control Data Distributed Communications Network.

CDCNET Statistics Manager (CSM)

The CDCNET Statistics Manager provides a bridge between commands that request statistics collection and the software that actually collects the statistics.

CDNA

Refer to Control Data Network Architecture.

CIM

Refer to Communications Interface Module.

Communications Interface Module (CIM)

The logic board within a CDCNET device interface that controls transmissions between the Line Interface Module (LIM) bus and the internal system bus (ISB).

Configuration Procurer

Software that obtains and submits for execution the device interface configuration file commands to the Command ME. This is done at system startup.

Control Data Distributed Communications Network (CDCNET)

1. The collection of compatible hardware and software products offered by Control Data Corporation to interconnect computer resources into distributed communications networks.

2. A network that is interconnected by Control Data Network Architecture (CDNA)-compatible hardware and software products.

Control Data Network Architecture (CDNA)

The network architecture designed by Control Data Corporation. CDNA follows the lower layer recommendations of the International Standards Organization's (ISO) Open System Interconnection (OSI) reference model.

CRC

Refer to Cyclic Redundancy Check.

CSM

Refer to CDCNET Statistics Manager.

CYBIL

Primary implementation language for NOS/VE computer systems and CDCNET software.

Cyclic Redundancy Check (CRC)

A check code transmitted with blocks of data. This code is used by several protocols.

D

Data Buffer

A structure for storing user data in device interface memory; contrast with descriptor buffer. A pointer is associated with the first character of data in the buffer. Data buffer length is configurable.

Data Communications Equipment (DCE)

The hardware that links data terminating equipment (DTE) to communications media. Data communications equipment is normally a modem or modem equivalent (data set).

Data Terminating Equipment (DTE)

Data communications equipment that allows human interaction with the databases and operations of a network.

DCE

Refer to Data Communications Equipment.

Descriptor Buffer

A data structure used for chaining data buffers.

Device Control Block (DVCB)

The Device Control Block is a data structure that retains information about a device interface's intelligent peripheral (IP) boards. Each IP has its own DVCB.

Device Identifier (DVMID)

A data structure that contains information specific to a port being controlled by an intelligent peripheral (IP) board.

Device Interface (DI)

The communications processor that Control Data offers as its CDCNET hardware product. Also called a CDCNET device interface.

Device Manager (DVM)

A set of routines responsible for the interface between CDNA's physical and link layers (layers 1 and 2, respectively).

DI

Refer to Device Interface.

DTE

Refer to Data Terminating Equipment.

DVCB

Refer to Device Control Block.

DVM

Refer to Device Manager.

DVMID

Refer to Device Identifier.

E

Ethernet

A baseband local area network protocol developed by the XEROX Corporation. CDCNET supports an Ethernet-compatible network.

Executive

A realtime monitor for the device interface which acts as the center of its software operating system. The Executive implements a set of procedures that enable users to efficiently share the system's available processing power and memory resources. It also provides some performance information about its own operation and that of its users.

F

Field Replaceable Unit (FRU)

Any hardware component that is designed for easy replacement on the customer site, such as a device interface logic board.

FIFO

Refer to First-In/First-Out.

First-In/First-Out (FIFO)

This term applies to data processing services in which requests are serviced in the same order they are received.

FRU

Refer to Field Replaceable Unit.

G

Gateway

A software interface between systems with different architectures and protocols.

H

HDLC

Refer to High-Level Data Link Control.

High-Level Data Link Control (HDLC)

The International Standards Organization's (ISO) bit-oriented protocol for the data link layer of the Open Systems Interconnection (OSI) reference model.

Host

Refer to Host Computer.

Host Computer

A mainframe computer system, connected to a communications network, that provides primary services such as data base access, user application execution, or program compilation. For CDCNET, a host computer provides network support functions, including maintenance of device interface load files.

I

Internal System Bus (ISB)

The circuitry within a CDCNET device interface that relays signals between the logic boards of the device interface.

Interrupt Service Routine

Code within the Executive which services hardware interrupt requests.

Intertask Message (ITM)

A means of scheduling work for device interface software routines that relies on a queueing service. There are two types of ITM: normal and express.

ISB

Refer to Internal System Bus.

ITM

Refer to Intertask Message.

L

LIM

Refer to Line Interface Module.

LIM Status Table (LST)

A data structure that maintains information about any LIM boards installed in the eight LIM slots of a DI.

Line Interface Module (LIM)

A smaller logic board within a CDCNET device interface that enables the device interface to be attached to terminal, workstation, and unit record equipment lines.

LST

Refer to LIM Status Table.

M

Main Processor Board (MPB)

The logic board within a CDCNET device interface that provides the primary processing power for the device interface.

Major Card Status Table (MCST)

A data structure that maintains information regarding any logic boards that have been installed in the eight major board slots of the DI.

Management Data Unit (MDU)

A generic data structure used within protocol data units for the expression of variable data types.

MCST

Refer to Major Card Status Table.

MDU

Refer to Management Data Unit.

MPB

Refer to Main Processor Board.

O

Online Loader

A CDCNET service that loads software into device interfaces when the software is needed while the network is operational, as opposed to initial loader, which loads software into device interfaces only when they are started up (initialized).

P

Packet

A group of binary digits, including data and control elements, switched and transmitted as a data unit by communications networks. The packet's data, control signals, and error-control information are arranged in a specific format. Different types of networks use different sizes of packets.

Parent Task

A parent task is a task that calls another task and supplies a recovery procedure for the same. If the called task fails, the parent task is notified.

PBST

Refer to PMM Bank Status Table.

PMM

Refer to Private Memory Module.

PMM Bank Status Table (PBST)

A data structure that maintains information regarding the memory banks of the Private Memory Module.

Port Status Table (PST)

A data structure that maintains information regarding the ports serviced by a Line Interface Module (LIM). There is one PST for each LIM that is physically present in a DI.

Private Memory Module (PMM)

An optional device interface board with 128K bytes of static RAM dedicated to the Main Processor Board (MPB) for code execution.

Programming System Report (PSR)

An official report to CDC of a problem with CDC software. A PSR can be sent to CDC either in hard-copy form or by using the online SOLVER program.

Protocol

A set of conventions that must be followed to achieve complete communications between the computer-related resources in a network. A protocol can reflect the following:

1. A set of predefined coding sequences, such as the control byte envelopes added to (or removed from) data exchanged with a terminal.

2. A set of data addressing and division methods, such as the block mechanism used between a network application program and Network Access Method.

3. A set of procedures that control communications, such as the supervisory message sequences used between a network application program and Network Access. Method.

Protocol Data Unit (PDU)

A data unit that is used to communicate information between peer entities.

PSR

See Programming System Report.

PST

Refer to Port Status Table.

S

SAP

Refer to Service Access Point.

SBST

Refer to SMM Bank Status Table.

SDS

Refer to Statistics Data Structure.

Service Access Point (SAP)

An exchange point between the services of two adjacent Control Data Network Architecture (CDNA) layers.

SMM

Refer to System Main Memory.

SMM Bank Status Table

A data structure that maintains information regarding the memory banks of the System Main Memory (SMM) board.

SOLVER

An online utility maintained by CDC that contains a database of reported software problems and solutions. SOLVER can be used for writing a PSR to report a problem with software.

SSR

Refer to Stream Service Routine.

Stack

An area in memory used as temporary storage for chaining calls during task or interrupt service routine execution. Task calls are chained on a user stack. Interrupt service routine calls are chained on a supervisor stack.

Statistics Data Structure (SDS)

A data structure that is used for the collection and reporting of DI software component statistics.

Stream Service Routine (SSR)

Software that implements Control Data's Network Architecture layer 2 (the data link layer). A stream service routine enables communication over a specific type of network solution.

Supervisor State

The higher of two privilege levels of CPU operation in a CDCNET DI. Supervisor state is used to process Interrupt Service Routines. Contrast with User State.

System Main Memory (SMM)

A device interface board with 1024K byte increments of dynamic RAM accessible by all interfaces and the resident Main Processor Board (MPB).

T

Task

Any code set within the Executive that is not an Interrupt Service Routine. Each task has a unique stack, intertask message queue, and priority.

Task Control Block (TCB)

The task control block is a data structure that maintains the context of a task while it is executing.

TCB

Refer to Task Control Block.

Terminal Interface Program (TIP)

CDCNET software that resides in terminal device interfaces and enables terminals/workstations that employ specific terminal protocols (such as asyn HASP, IBM 2780/3780, and IBM 3270) to communicate in CDCNET networks.

TIP

Refer to Terminal Interface Program.

U

User State

The lower of two privilege levels of CPU operation in a CDCNET DI. User state is used to process tasks. Contrast with Supervisor State.

X

X.25

The Consultative Committee of International Telephone and Telegraph (CCITT) standard for the interface between data terminal equipment and data communications equipment in an X.25 packet switching network.

Data Structures B

Buffers

Following are the definitions of descriptor and data buffers (also known as small and large buffers, respectively). Descriptor buffers are linked by the next_descriptor field and chains of buffers are linked by the next_message field.

Comdeck **CMDTBUF**

Format **VAR name: BUF_PTR**

CYBIL definition:

```
CONST
    max_buffer_size = 2304,
    max_chars_in_buffer = max_buffer_size - 2,
    critical_priority = 0,
    default_sbufflen = 32,
    default_lbufflen = 144,
    high_priority = 1,
    max_sbufflen = 64,
    max_lbufflen = max_buffer_size,
    medium_priority = 2,
    min_sbufflen = 32,
    min_lbufflen = 64,
    low_priority = 3,
    memory_overhead = 6,
    lbuff_overhead = memory_overhead+2,
    self = 1; { added to destination_count for broadcast, etc.

  TYPE
    non_empty_message_size = 1 .. 65535,
    message_size = 0 .. 65535,
    chars_in_buffer = 0 .. max_chars_in_buffer,
    non_empty_buffer = 1 .. max_chars_in_buffer,
    pref_type = (absolute@, conditional@, yield@); { See NOTE, below

  TYPE
    data_descriptor = record             { Descriptor Buffer Definition
      next_descriptor: ^data_descriptor, { next buffer in msg
      next_message: ^data_descriptor, { next msg in queue
      the_data: ^data_space_record,   { the actual stored data
      decstamp: integer,              { millisecond time stamp
      offset: non_empty_buffer,       { distance from the_data to 1st byte
      count_buffer: chars_in_buffer,  { # bytes data in buffer
      count_message: message_size,    { # bytes data in message (1st buffer only)
      usage_descriptor: 0 .. 32767,   { usage count of descriptor
      user_data: data_descriptor_user_data_type, { user defined data
    recend,

    data_space_record = record       { Data Buffer Definition
      data_usage: 0 .. 32767, {usage count for data space
      data_text: ARRAY [1..max_chars_in_buffer+1] OF CELL,
    recend,

    buffer_request_limit = 1 .. 999,

    executive_extent = 1 .. 32767, { size of executive extent
    buf_ptr = ^data_descriptor;
```

```
{
{   The following are definitions of user defined data kept in the
{   data_descriptor record. These fields are normally unused since
{   the common subroutines request sbufflen (32) bytes for the
{   data_descriptor record.
{

  TYPE
    data_descriptor_user_data_type = record
      case integer of

      = 1 = { XEROX TRANSPORT
        sequence: 0 .. 0ffff(16),

      = 2 = { TDSM (text_processor)
        text_process_1: ^cell,
        text_process_2: ^cell,

      = 3 = { TDSM (output_queue)
        marked_output: boolean,

      casend,
    recend;
```

Remarks Be aware of what the common subroutines do with buffers; for example, buffers may be released when STRIP is called.

NOTE

There are three options defined by the type pref_type. These options are used in calls to APPEND, PREFIX, and BUILD_HEADER_IN_PLACE. Internally the options have the following meanings when the routine obtains data buffers.

absolute@	Uses the sure interface; always returns a successful status.
conditional@	Uses the maybe interface; returns the request status from the Executive to the user.
yield@	Uses the maybe interface; if successful, returns that status to the user. If not, yields and repeats the process.

DVM Command Packets and Status ITMs

Following are the CYBIL definitions for device manager (DVM) command packets and status intertask messages (ITMs). Refer to chapter 9 for a description of DVM and the services it provides.

```
{================================================================}
{                   CIM COMMAND PACKET RECORD                    }
{================================================================}

TYPE
  cim_cmd_pkt_rec_type = record
    code:             cc_cmd_range,
    dvm_id:           ^cell,
    case integer of

    = 100 = { force packet size }
      body:             array [4 .. 8] of 0 .. 0ffff(16),

    = cc_hb =
      hb_status: cim_flag_set_type,

    = cc_configure_line,
      cc_reconfigure_line =
      config_info_ptr:  ^cim_config_rec_type,

    = cc_start_input =
      input_flags:      cim_flag_set_type,
      input_setup_state: integer,

    = cc_start_output =
      output_flags:     cim_flag_set_type,
      output_setup_state: integer,
      output_buf_ptr:   ^data_descriptor,

    = cc_terminate_io =
      terminate_flags:  cim_flag_set_type,

    = cc_define_user_area =
      user_data_ptr:    ^cim_user_data_rec_type,

    = cc_execute_state =
      state_flags:      cim_flag_set_type,
      execute_setup_state: integer,

    = cc_line_setup =
      lim_num:          0 .. upper_lim_num,
      lim_port:         0 .. upper_port_num,

    casend,
  recend;
```

```
{=================================================================}
{                CIM STATUS RESPONSE & ITM RECORD                 }
{=================================================================}

TYPE
  cim_status_rec_type = record
    dvm_id:             ^cell,
    case integer of

    = 0 = { force packet size }
      stat_pkt_size:    array [4 .. 8] of 0 .. 0ffff(16),

    = sc_card_ok =
      hb_status: cim_flag_set_type,

    = sc_line_configured,
      sc_line_reconfigured =
      config_status:    boolean,
      config_info_ptr: ^cim_config_rec_type,

    = sc_line_deleted =
      xflags: cim_flag_set_type,

    = sc_line_enabled =
      enable_flags:     cim_flag_set_type,

    = sc_line_disabled =
      yflags: cim_flag_set_type,

    = sc_input_received =
      input_flags: cim_flag_set_type,
      input_buf_ptr:    ^data_descriptor,

    = sc_output_sent =
      output_flags:     cim_flag_set_type,
      output_buf_ptr:   ^data_descriptor,

    = sc_io_terminated =
      terminate_flags:  cim_flag_set_type,

    = sc_user_area_defined =
      user_area_flags:  cim_flag_set_type,
      user_data_ptr: ^cim_user_data_rec_type,

    = sc_state_executed =
      zflags: cim_flag_set_type,

    = sc_line_disconnect =
      disconnect_flags: cim_flag_set_type,

    casend,
  recend;


TYPE
  cim_stat_itm_rec_type = record
    status:             sc_status_range,
    info:               cim_status_rec_type,
  RECEND;
```

```
{================================================================}
{     DATA TYPE TO DEFINE CMD PKTS TO ESCI VIA DVM               }
{================================================================}

TYPE
  uint16 = 0 .. 0ffff(16),

  fesci_cmd_pkt_type =  record case boolean of
  =true=
    fesci_cmd_code_type: uint16,
    dvm_id: ^cell,
    case 0 .. 255 of
    = cc_strtup =
      config_tbl_ptr_type: ^ll_esci_lib_type,
    = cc_xmit =
      xmt_buffer: buf_ptr,
    casend,
  =false=
    cim_cmd: cim_cmd_pkt_rec_type,
  casend recend;


{================================================================}
{     DATA TYPES TO DEFINE ESCI INTERTASK MESSAGE                }
{================================================================}

TYPE
  esci_state_type = (idle, operational, suspended);

TYPE
  esci_status_type = (stat_null, stat_up, stat_dwn,
                      stat_sus, stat_failed);

TYPE

  esci_inttsk_msg_type=  record

    message_type: uint16,

    case 0 .. 65535 of
    = 0 =
      byte:                    PACKED ARRAY [0..126] of char,

    = esci_startup_cmd, esci_shutdown_cmd, esci_suspend_cmd,
      esci_resume_cmd, esci_statistics_cmd, esci_wakeup_cmd=
      esci_lib_ptr:^esci_lib_type,

    = esci_nures_res, esci_rcv_res, esci_xmit_res, esci_stistc_res,
      esci_switches_res, esci_tdr_res, esci_diag_res, esci_nop_res,
      esci_dump_res, esci_xsub_res =
      dvm_id: ^cell,

      case 0  . 65535 of
      = esci_nures_res =
        esci_status: esci_status_type,
        reason: reason_code_set,

      = esci_rcv_res =
        rcv_status_info: packed record
          CASE boolean OF
          = TRUE =
            rcv_status_bits: rcv_status_bits_set,
          = FALSE =
            chk_586: packed record
              frm_stat: 0..0f(16),
              frm_err: 0..3f(16),
            recend,
          CASEND,
        recend,
        rcvd_buffer: ALIGNED buf_ptr,
      = esci_xmit_res =
          xmt_status_info:  packed record
          bc_mc_frame: boolean,
          rsv_stat_1: boolean,
          frame_transmitted: boolean,
          abort_requested: boolean,

          rsv_stat_2: boolean,
          no_carrier_sensed· boolean,
          lost_cts: boolean,
          memory_underrun: boolean,

          deferred: boolean,
          heart_beat: boolean,
          too_many_collisions: boolean,
          rsv_stat_3: boolean,

            number_of_collisions: 0 .. 15,
          recend,
          xmted_buffer: ALIGNED buf_ptr,

      = esci_ststc_res =
          not_used_stats: uint16,
          crc_errors: uint16,
          aln_errors: uint16,
          rsc_errors: uint16,
          ovr_errors: uint16,

        = esci_switches_res =
          sense_switches: set of (esci_switch_1, esci_switch_2,
          esci_switch_3, esci_switch_4, esci_switch_5, esci_switch_6,
          esci_switch_7,  esci_switch_8, esci_switch_rsv_1,
          esci_switch_rsv_2, esci_switch_rsv_3, esci_switch_rsv_4,
          esci_switch_rsv_5, esci_switch_rsv_6, esci_switch_rsv_7,
          esci_switch_rsv_8),

        = esci_tdr_res =
          tdr_result:  record
            link_ok: boolean,
            link_not_ok. boolean,
            link_open: boolean,
            link_short: boolean,
            tdr_resv_1: boolean,
            tdr_time: 0 .. 7ff(16),
          recend,

        = esci_diag_res =
          diagnose_result.  record
            diagnose_resv_1: 2 .. 0ffff(16),
            diagnose_result: boolean,
          recend,

        = esci_dump_res =
          dump_status: uint16,
          dump_buffer: ^cell,

        = esci_xsub_res =
          subroutine_info: ^cell,
      casend,
    casend,
  recend,
```

Executive Error Table

The Executive Error Table is initialized by the system Executive and is located in MPB RAM.

Comdeck CMCERTB

CYBIL definition.

```
CONST
  number_of_error_buffers = 3; { Number of error buffers, minus one

TYPE
  executive_error_table = record
    stop_supervisor_stack_pointer: ^cell, { supervisor stack pointer if exec
                                          { stop
    last_error_address: ^error_buffer,
    lock_last_error: 0 .. 0ffff(16), { last_error_address being updated
    address_error_being_processed: 0 .. 0ffff(16),
    number_of_spurious_interrupts: 0 .. 0ffff(16),
    smm_error_count: array[ 0 .. 7 ] of 0 .  16),
    total_error_count: 0 .. 0ffff(16),
    system_ancestor_tcb: task_ptr,
    debug_address_called_on_error: ^cell,
    error_buffers: array[ 0 .. number_of_error_buffers ] of error_buffer,
  recend;

TYPE
  error_buffer = record
    executive_error_code: ex_error_codes,
    lock_error_buffer: 0 .. 0ffff(16), { non-zero k error buffer
    binclock_at_time_of_error: integer,
    d0_thru_d7: array[ 0 .. 7 ] of integer,
    a0_thru_a6: array[ 0 .. 6 ] of integer,
    status_register: 0 .. 0ffff(16),
    supervisor_stack_pointer: ^cell,
    user_stack_pointer: ^cell,
    program_counter: ^cell,
    tcb_for_running_task: task_ptr,
    module_name: pmt$program_name,
    module_offset: 0 .. 0ffff(16),
    error_during_firewall: 0 .. 0ffff(16), { if non-zerro then error during
                                           { firewall
    firewall_procedure_address: ^cell,
    mpb_status_register: mpb_status_word,
    case ex_error_codes of
    = bus_error_i, address_error_i =
      first_failure_capture_address: ^cell,
      bus_exception_status: 0 .. 0ffff(16),
      access_address: ^cell,
      instruction_register: 0 .. 0ffff(16),
    = smm_single_bit_error_i, smm_double_bit_error_i =
      smm_card_slot: 0 .. 7,
      smm_error_log: 0 .. 0ffff(16),
    casend
  recend;

TYPE
  ex_error_codes = ( unused_0,
                     unused_1,
                     bus_error_i,
                     address_error_i,
                     illegal_instruction_i,
                     zero_divide_i,
                     chk_instruction_i,
                     trapv_instruction_i,
                     privilege_violation_i,
                     trace_interrupt_i,
                     line_1010_interrupt_i,
                     line_1111_interrupt_i,
                     smm_single_bit_error_i,
                     smm_double_bit_error_i,
                     task_runs_too_long_i );

VAR
 exec_error_table· [XREF] executive_error_table;
```

```
{
{  MPB status register from address 6100(16) in mpb ram
{

 CONST
  mpb_status_byte = 6100(16);

 TYPE
  mpb_status_word = packed record
   access_code: type_of_memory,
   access_type: type_of_io,
   dtack_time_out: boolean,
   bus_lock_time_out: boolean,
   parity_error: boolean,
   write_protect: boolean,
   dead_man_time_out: boolean,
   bit_not_used: boolean,
   manual_reset: boolean,
   external_clear: boolean,
   a_c_low: boolean,
   temperature_shutdown: boolean,
   temperature_warning: boolean,
   battery_low: boolean,
  recend;

TYPE type_of_memory = (pmm_bus, itb_bus, mpb_random, no_bus);

TYPE type_of_io = (no_io, read_io, write_io, intack_io);
```

MPB RAM Definition

MPB RAM resides at a fixed address within DI memory.

Comdeck **SIDRAM**

CYBIL definition:

```
VAR
  mpb_ram_ptr. [STATIC, READ] ^mpb_ram := NIL, { ^MPB_RAM from byte address 0

TYPE
  mpb_ram = packed record { description of mpb ram starting from address 0
    vector: array [1 .. 256] of ^cell, { vector space
    system_id: system_id_type, { unique identifier for this hardware box
    system_id_checksum: 0 .. 0ffff(16), { system_id checksum
    table_format_version: 0 .  0ffff(16), { version of this RAM table format
    status: 0 .. 0ff(16), { MPB status register low 4 bits (if NMI occurs)
    mpb_ram_zeroed: 0 .. 0ff(16), { MPB RAM zeroed flag
    smm_size: integer, { # contiguous usable SMM bytes from 100,000(16)
    boot_map_entry_address: ^cell, { ^map entry used as bootstrap card
    auto_dump_table_address: ^cell, { ^ Auto Dump Table
    reset_status: 0 .. 0ff(16), { reset status saved from most current reset
    reset_code: 0 .. 0ff(16), { reset code ( from both software and hardware
    software_error_code: 0 .. 0ff(16), { software error code
    hardware_reset_code: 0 .. 0ff(16), { possible hardware cause for reset
    version: 0 .. 0ffff(16), { version within last accepted help offer
    network_id: integer, { network id within last accepted help offer
    help_system_id: system_id_type, { system id within last accepted help
    auto_dump_subroutine_address: ^cell, { ^ Auto Dump Table generator
    auto_dump_subroutine_length. 0 .. 0ffff(16), { length in 16-bit words
    auto_dump_subroutine_checksum: 0 .. 0ffff(16), { 16-bit ones complement
    map_table· ALIGNED array [1 .. 72] of integer, { card map table
    reserved_4_bytes. integer, { reserved for future use
    mpb_error_routine_pointer: ^cell, { starting address of MPB error routine
    mpb_error_routine_length: 0 .. 0ffff(16), { length in 16-bit words
    pmm_error_routine_pointer: ^cell, { starting address of error routine
    pmm_error_routine_length: 0 .. 0ffff(16), { length in 16-bit words
    smm_error_routine_pointer: ^cell, { starting address of error routine
    smm_error_routine_length: 0 .. 0ffff(16), { length in 16-b   words
    expected_smm_interrupt_flag: ^cell, { expected SMM interrupt flag pointer
    ept_address: ALIGNED ^cell, { starting address of the entry point table
    loaded_module_list: ^llt$module_header, { pointer to 1st entry
    unsatisfied_externals: ^cell, { ^ unsatisfied externals table
    desbuflen. integer, { length of descriptor buffers
    datbuflen: ALIGNED integer, { length of data buffers
    reserved_memory: ALIGNED 0 .. 32767, { reserved memory for critical use
    initial_loader_checksum· ALIGNED 0 .. 0ffff(16), {
    sys_cnfg_ptr: ^cell, { address of executive configuration table
    system_ancestor_task_id: ALIGNED task_ptr, { ^system ancestor tcb
    current_3b_ephemeral_sapid: ALIGNED sap_id_type, { next 3b SAP to assign
  recend;

  CONST
    software_error_address = 41a(16); { ^mpb_ram_ptr^.software_error_code
```

Queue Control Block

The queue control block (QCB) structure maintains an intertask message queue for use by the task that owns it. It is defined by type qcb@.

Comdeck **CMDTTSK**

Format **VAR name: qcb_ptr**

CYBIL definition:

```
TYPE
  qcb@ = record
    length: 0 .. 32767, { current length of queue
    count: 0 .. 32767, { number of enqueues that have happened to this QCB
    qnext: buf_ptr,
    qlast: buf_ptr,
    qcharacters: integer, { number of characters in queue
  recend;

TYPE
    qcb_ptr = ^qcb@;
```

System Configuration Table

The System Configuration Table is a data structure that retains the status of essential CDCNET system variables. It is in the form of a record with fields indicating such things as the highest address in MPB RAM and the states of memory and buffers.

Comdeck CMCCNFG

Format VAR sys_cnfg: [xref] exec_config

CYBIL definition:

```
    VAR
      address. ^exec_config,
      table. exec_config;

   TYPE
     exec_config = record
       maxprior: 0 .. 32767, { highest valid priority -- lowest is zero
       databac: 0 .. 32767, { data buffer available count
       descbac: 0 .. 32767, { descriptor buffer available count
       lbufflen· integer, { data space length in bytes
       sbufflen: integer, { descriptor buffer length in bytes
       stdstack: integer, { standard stack allocation
       running: task_ptr, { task_ptr of running task
       curprior: priorities, { currently running priority
       schprior: priorities, { highest scheduled priority
       pmtok: boolean, { task preemption permission flag
       vecslice. integer, { interrupt vector for time slice interrupt
       vecintvl: integer, { interrupt vector for interval timer interrupt
       vecclock: integer, { interrupt vector for millisecond interrupt
       mpbramtop: integer, {numerically largest address in mpb ram
       privatetop: integer, { numerically largest address in private memory
       globfree: integer, { number of bytes of free global memory
       locfree: integer, { number of bytes of free private memory
       mpbfree: integer, { number of bytes of free mpb ram memory
       globfrag. 0 .  32767, { number of extents of free global memory
       locfrag: 0 .. 32767, { number of extents of free private memory
       mpbfrag: 0 :. 32767, { number of extents of free mpb ram memory
       deloadtyp: deload_flag, { type of memory to release flag for deload task
       deloadtcb: task_ptr, { task_ptr of deload task
       deloadmpb· 0 .. 0ffff(16), { deloadable bytes of mpb ram
       deloadpmm: integer, { deloadable bytes of private memory
       deloadsmm: integer, { deloadable bytes of global memory
       mpbthresh. 0 .. 0ffff(16), { dload threshold for mpb am
       pmmthresh: integer, { deload threshold for private memory
       smmthresh: integer, { deload threshold for global memory
       pmtreq: boolean, { task will yield on next trap 1 or trap 4 if set
       retryflag: 0 .. 32767, { retry in progress flag
       clocktyp: 0 .. 1, { 0 = millisecond clock; 1 = real time c  ck
       timertcb: task_ptr, { task_ptr of time task
       diagflag: diagset, { current debug support tools set
       binclock: integer, { .1 second accuracy binary time of day
       decclock: bcd_time, { .1 second accuracy bcd date/time
       assumed_year: 0 .. 32767, {assumed year used by executive
       firewall: integer, { address of interrupt firewall chain
       prilist: array [priorities] of qcb@, { ready lists for tasks scheduled at priorities
       globmem: qcb@, { global memory extent list
       privmem: qcb@, { private memory extent list
       mpbmem: qcb@, { mpb ram memory extent list
       badextnt. qcb@, { bad extent list
       iptlist: qcb@, { defined interrupts list
       lbuffq: qcb@, { data buffer queue
       sbuffq: qcb@, { descriptor buffer queue
       data_buffer_count: 0 .. 32767, { number of data buffers
       descriptor_buffer_count: 0 .. 32767, { number of descriptor buffers
       expire_stp: 0 .. 32767, { expire state transition processor timer
       stack_overflow_space: integer, { size of stack overflow area allocated
       task_overflowed: task_ptr, { task_ptr of task which has overflowed its stack
       pc_chkinst_address· integer,  { PC where chk instruction executed
       usp_chkinst_address: integer, { USP when chk instruction executed
       mpb_light_state: light_status, { status of mpb lights
    idle_loop_count: integer, { executions of idle loop since last clear
    reservetop: integer, { numerically largest address in reserve memory
    rsvfree: integer, { number of bytes of reserve ram memory
    rsvfrag: 0 .. 32767, { number of extents of reserve global memory
    rsvmem: qcb@, { reserve ram memory extent list
    memory_state: memory_ state_type, {depends on amount of free memory
    buffer_state: buffer_ state_type, {depends on amount of free memory
    stp_timer: ^timer,  { timer id of state transition processor
    cio_b: cio_port_b,  { CIO port B bit settings
    cio_c: ALIGNED cio_port_c,  { CIO port C bit settings
    supervisor_state_ok: 0 .. 0ffff(16),  { 1 = OK, 0 = user task  recend;

TYPE
  priorities = 0 .. max_priority,
  stack_size = min_stack_size .. max_stack_size;

CONST
  max_priority = 7,
  min_stack_size = 0,
  max_stack_size = 2000(16);

TYPE
  deload_flag = ( llc$mpb, llc$pmm, llc$smm );
```

Task Control Block

The task control block (TCB) structure describes task constants and types for use by the Executive. It is defined by type taskid@.

Comdeck **CMDTTSK**

Format **VAR name: task_ptr**

CYBIL definition:

```
TYPE
  taskid@ = packed record { packed to force correct data mappings
    next_task: task_ptr, { chain to next task_ptr
    id: integer, { = ''TCB'
    stsiz: integer, { size of current stack segment
    chldq: task_ptr, { task_ptr of my next sibling
    adult: task_ptr, { task_ptr of my parent
    child: task_ptr, { task_ptr of my child
    stack: integer, { address of my current stack segment
    state_fill: 0 .. 31,
    state: 0 .. 7, { my.current state
    transition_fill: 0 .. 15,
    trans: 0 .. 15, { transition that entered this state
    tran: array [0 .. 15] of 0 .. 65535, { counts of transitions to date
    slices: 0 .. 65535, { count of time slice overruns to date
    flag_fill_1: 0 .. 31,
    preempted: boolean, { task has been preempted; registers all saved (else only
                        { A6 and D7)
    hold: boolean, { used by timer task to deflect requests into Normal queue
    wku: boolean, { wakeup pending if set
    flag_fill_2: 0 .. 255,
    express: qcb@, { inter-task message queue
    normal: qcb@, { inter-task message queue
    preempt_permit : 0 .. 32768, { zero = task not preemptible; else preemptible
    cpriority: 0 .. 32768, { my nominal priority
    priority: 0 .. 32768, { my actual priority
    d_registers: array [0 .. 7] of integer, { only register D7 normally valid
    a_registers: array [0 .. 6] of ^cell,   { only register A6 normally valid
    usp: ^cell, { user stack pointer
    sr: 0 .. 0ffff(16), { status register
    pc: ^cell, { program counter
    tcbfrb: ^sat$recovery_block, { pointer to task failure recovery block
    tcb_epa:  ^cell, { task entry point address
    tcb_space: integer, { amount of unused space in reserved stack area
    tcbmhp: dlt$module_header_ptr, { pointer to module header
    age: 0 .. 0ffff(16), { age within dispatch queue
  recend;

TYPE
  task_ptr = ^taskid@;
```

Constants and Common Types C

Constants

Following is an alphabetical listing of the CYBIL constants used in the procedures and functions defined in chapter 10.

dbc$single_line = 79 { characters

dlc$default_priority = 0

dlc$max_section_checksum = 0ffff(16)

dlc$max_section_offset = 7fffffff(16)

dlc$max_section_ordinal = 0ffff(16)

dlc$maximum_68000_address = 7fffffff(16)

max_buf_level = 400

max_cmd_q_size = 200

max_number_ports = 32

max_priority = 7

max_response_message_id = 65535

max_stack_size = 8192

max_stat_q_size = 200

maximum_device_name_size = 11

mdu_field_size = 32000

min_response_message_id = 33000

min_stack_size = 2048

osc$max_name_size = 31

rcv_threshold = 0 { buffer and descriptor thresholds

svm_thresh = 0 { SVM buffer threshold

upper_ip_num = 7

upper_lim_num = 7

upper_port_num = 3

Common Types

Following is an alphabetical listing of the CYBIL types used in common by the procedures and functions defined in chapter 10.

```
access_device_status_type =
 (device_state_not_on,
  device_state_off,
  device_status_not_cnfg,
  device_configured,
  device_not_configurable,
  device_unavailable,
  device_released,
  invalid_cnfg_table_address,
  device_status_already_cnfg,
  invalid_key)

access_status_type =
 (sap_opened,
  sap_not_opened)

bcd = 0 .. 9

bcd_time = packed record
  lyear:    bcd,
  ryear:    bcd,
  lmonth:   bcd,
  rmonth:   bcd,
  lday:     bcd,
  rday:     bcd,
  lhour:    bcd,
  rhour:    bcd,
  lminute:  bcd,
  rminute:  bcd,
  lsecond:  bcd,
  rsecond:  bcd,
  deci:     bcd,
  centi:    bcd,
  milli:    bcd,
recend

buffer_request_limit = 1 .. 999

buffer_state_type =
 (buffer_good,
  buffer_fair,
  buffer_poor,
  buffer_congested)
```

card_info_record = record
```
  card_type: hardware_resource_type,
  primary_address: integer,
  secondary_address: integer,
recend
```

chars_in_buffer = 0 .. max_chars_in_buffer

cim_config_rec_type = packed record
```
  line_mode:           0..255,
  char_length:         0..255,
  stop_bits:           0..255,
  crc_type:            0..255,
  crc_preset:          0..255,
  modem_mode:          0..255,
  modem_type:          0..255,
  not_used:            0..0ffff(16),  { place holder
  input_baud_rate:     0..0ffff(16),  { bps rate time const
  output_baud_rate:    0..0ffff(16),
  state_prg_indx:      0..255,
  clock_mode:          0..255,
recend
```

cim_flag_set_type = set of 0..15

cim_user_data_mask_type = set of 0..cim_user_data_max

cim_user_data_max = 79

cim_user_data_rec_type = record
```
  valid_data:  cim_user_data_mask_type,
  data:        cim_user_data_type,
recend
```

cim_user_data_type = packed array [0..cim_user_data_max] of 0..0ff(16)

close_statistics_status =
```
 (statistics_sap_closed,
  statistics_sap_entry_not_found,
  mismatch_statistics_sap)
```

clt$status = record
```
  normal: boolean,
  response_id: min_response_message_id .. max_response_message_id,
  condition: buf_ptr,                     { management data unit syntax
recend
```

```
component_status_type = record
  name: string (maximum_device_name_size), { Hardware physical device name
  state: device_state_type,                { device state
  status: device_status_type,              { device status
recend

device_state_type =
 (device_on,
  device_off,
  device_down)

device_status_type =
 (device_not_cnfg,
  device_cnfg,
  device_enabled,
  device_active)

dg_data_buffer_priority = medium_priority  { CIM online diagnostic

diagset = set of diagset@

diagset@ =
 (unused_15, { unused flag position
  unused_14, { unused flag position
  unused_13, { unused flag position
  unused_12, { unused flag position
  unused_11, { unused flag position
  unused_10, { unused flag position
  unused_9,  { unused flag position
  unused_8,  { unused flag position
  diagwrap,  { wrap-around journal buffer
  diagpcsr,  { log PC and SR on precise CMCSTAT entry
  diagipts,  { log spurious interrupts
  diagclc,   { perform excess buffer collection
  diagdups,  { test for duplicate buffer release
  diagbufs,  { log memory allocation/release in journal
  diagevnt,  { log task event messages in journal
  diagtask)  { log task state changes in journal

dip_parm_type = record          { Dump Intelligent Peripheral
  ip_number:   0..upper_ip_num,           { board slot number of device
  dump_start:  ^cell,                   { local RAM address
  dump_length: 0..0ffff(16)           { size of RAM to dump
recend

display_procedure = ^PROCEDURE
 (sds_hdr_ptr:   ^sds_header;
  time:          report_time_type;
  VAR disp_msg:  buf_ptr)
```

```
dlc$default_immediate_control = FALSE

dlc$default_preemptibility = FALSE

dlc$max_section_length = dlc$max_section_offset

dlt$ampm_time = string (8)

dlt$checksum = 0 .. ilc$max_section_checksum

dlt$date = packed record                  { Date request return value
  fill: 0 ..1f(16),
  case date_format: dlt$date_formats of
  = dlc$month_date =
    month: dlt$month_date,                      { month DD, YYYY
  = dlc$mdy_date =
    mdy: dlt$mdy_date,                          { MM/DD/YY
  = dlc$iso_date =
    iso: dlt$iso_date,                        { YYYY-MM-DD
  = dlc$ordinal_date =
    ordinal: dlt$ordinal_date,               { YYYYDDD
  = dlc$dmy_date =
    dmy: dlt$dmy_date                           { DD/MM/YY
  casend,
recend

dlt$date_formats =
 (dlc$default_date,
  dlc$month_date,
  dlc$mdy_date,
  dlc$iso_date,
  dlc$ordinal_date,
  dlc$dmy_date)

dlt$dmy_date = string (8)

dlt$entry_description = record
  node: node_control,
  name: pmt$program_name,
  address: dlt$68000_address,
  module_header_address: ^dlt$module_header,
  link_address: ^dlt$entry_description,
  declaration_matching_required: boolean,
  declaration_matching_value: string (8),
  language: dlt$module_generator,
recend

dlt$hms_time = string (8)

dlt$iso_date = string (10)
```

dlt$load_id_ptr = ilt$module_header_ptr

dlt$maximum_modules = 0 .. dlc$max_section_ordinal

dlt$mdy_date = string (8)

dlt$millisecond_time = string (12)

dlt$module_attributes = set of (dlc$nonbindable, dlc$nonexecutable)

dlt$module_generator =
```
 (dlc$algol,
  dlc$apl,
  dlc$basic,
  dlc$cobol,
  dlc$assembler,
  dlc$fortran,
  dlc$object_library_generator,
  dlc$pascal,
  dlc$cybil,
  dlc$pl_i,
  dlc$unknown_generator,
  dlc$the_c_language,
  dlc$ada)
```

dlt$module_header = record
```
  link_address: dlt$module_header_ptr,
  mod_head: dlt$module_identification,
  allocated_sections: array [0..*] of dlt$section_identification,
recend
```

**dlt$module_header_ptr** = ^dlt$module_header

dlt$module_identification = record
```
  name:  pmt$program_name,
  kind:  dlt$module_kind,
  time_created:  dlt$time,
  date_created:  dlt$date,
  attributes:  dlt$module_attributes,
  breakpoint_set:  boolean,
  retain:  boolean,
  member_of_internal_set:  boolean,
  use_count:  dlt$maximum_modules,
  reference_list:  ^dlt$module_reference,
  module_status:  dlt$module_status,
  entry_address:  ^dlt$entry_description,
  greatest_section_ordinal:  dlt$section_ordinal,
  transfer_symbol_address:  ^dlt$entry_description,
recend
```

dlt$module_kind =
 (dlc$mi_virtual_state,
  dlc$vector_virtual_state,
  dlc$iou,
  dlc$motorola_68000,
  dlc$p_code,
  dlc$motorola_68000_absolute)

dlt$module_reference = record
  link_address: ^dlt$module_reference,
  reference_link: ^dlt$module_header,
recend

dlt$module_status =
 (dlc$active,
  dlc$deloaded,
  dlc$load_in_progress)

dlt$month_date = string (18)

dlt$ordinal_date = string (7)

dlt$section = array [1..*] of 0..255

dlt$section_access_attribute =
 (dlc$read,
  dlc$write,
  dlc$execute,
  dlc$binding,
  dlc$read_other,
  dlc$write_other,
  dlc$execute_other,
  dlc$binding_other)

dlt$section_access_attributes = set of dlt$section_access_attribute

dlt$section_address_range = - (dlc$max_section_offset + 1) ..
  dlc$max_section_offset

dlt$section_identification = record
  checksum: dlt$checksum,
  length: dlt$section_length,
  attributes: dlt$section_access_attributes,
  case 1..2 OF
  = 1 =
    address: ^dlt$68000_attributes,
    module_kind: dlt$module_kind,
  = 2 =
    section_address: ^dlt$section,
    kind: dlt$section_kind.
  casend,
recend

```
dlt$section_kind =
 (dlc$code_section,
  dlc$binding_section,
  dlc$working_storage_section,
  dlc$common_block,
  dlc$extensible_working_storage,
  dlc$extensible_common_block,
  dlc$line_table_section)

dlt$section_length = 0 .. dlc$max_section_length

dlt$section_offset = 0 .. dlc$max_section_offset

dlt$section_ordinal = 0 .. dlc$max_section_ordinal

dlt$time = packed record                  { Time request return value.
  fill: 0 .. 3f(16),
  case time_format: dlt$time_formats of
  = dlc$ampm_time =
    ampm: dlt$ampm_time,                            { HH:MM AM or PM
  = dlc$hms_time =
    hms: dlt$hms_time,                             { HH:MM:SS
  = dlc$millisecond_time =
    millisecond: dlt$millisecond_time,          { HH:MM:SS.MMM
  casend,
recend

dlt$time_formats =
 (dlc$default_time,
  dlc$ampm_time,
  dlc$hms_time,
  dlc$millisecond_time)

dlt$68000_absolute = packed record
  load_address: dlt$68000_address,
  transfer_address: dlt$68000_address,
  text: ALIGNED SEQ ( * ), { REP n OF byte
recend

dlt$68000_address = 0 .. dlc$maximum_68000_address

dm_input_buffers_priority = high_priority     { DVM priority

dm_configuration_priority = medium_priority  { DVM priority

executive_extent = 1 .. 32750

force_stat_reporting_status =
 (statistics_report_issued,
  statistics_sap_not_found,
  sds_header_address_invalid)
```

```
hardware_resource_type =
  (mpb,
   cim,
   esci,
   reserved_3,
   reserved_4,
   reserved_5,
   reserved_6,
   pim,
   pmm,
   smm,
   reserved_10,
   reserved_11,
   disc,
   mci,
   dci,
   slot_empty,
   lim,
   port,
   bank)

ip_cmd_pkt_type = packed record
  case integer of
  = 0 =
    command,
    status:        0..255,
    destination,
    source:        ^cell,
    length,
    filler,
    checksum:      0..65535,
  = 1 =
    cmd_status:  0..65535,
    destination_adr,
    source_adr:  integer,
  = 2 =
    cmd_wrd:       array [0..6] of 65535,
  casend,
recend

itm_exp_mask_type = set of sc_status_range

key_type = (numeric_key@, pointer_key@, string_key@)

key_record = record
  case key_kind: key_type of
  = numeric_key@ =
    numeric: integer,
  = pointer_key@ =
    pointer: ^cell,
  = string_key@ =
    string_type: ^string ( * ),
  casend
recend
```

```
light_status = packed record
  operational: 0 .. 3,        { always zero for operational state
  mpb_busy: boolean,           { On for busy mpb
  test_state: test_status,
  failed_card: 0 .. 7,        { failed card slot number
  unused_byte: 0 .. 0FF(16), { force word align
recend

mdu_field_type =
 (bin_str,
  bin_octet,
  char_octet,
  bin_int,
  bin_sint,
  bcd_char,
  format)

memory_owner_type = 0 .. 3fff(16)

memory_state_type =
 (memory_good,
  memory_fair,
  memory_poor,
  memory_congested)

message_size = 0 .. 65535

milliseconds = integer

mpb_status_word = packed record
  access_code: type_of_memory,
  access_type: type_of_io,
  dtack_time_out: boolean,
  bus_lock_time_out: boolean,
  parity_error: boolean,
  write_protect: boolean,
  dead_man_time_out: boolean,
  bit_not_used: boolean,
  manual_reset: boolean,
  external_clear: boolean,
  a_c_low: boolean,
  temperature_shutdown: boolean,
  temperature_warning: boolean,
  battery_low: boolean,
recend

node = record
  balance: condition_range, { balance factor for sub-tree
  association: ^node_control, { points to user data
  key: key_record,
  llink: ^node, { sub-tree links
  rlink: ^node,
recend
```

```
node_control = record
  length: executive_extent, { length of the associated table
  dump_id: string (4), { validity check, should contain user value
recend

non_empty_buffer = 1 .. max_chars_in_buffer

non_empty_message_size = 1 .. 65535

number_of_data_byte_types =
 (two_byte_statistic,
  four_byte_statistic,
  eight_byte_statistic)

open_statistics_status =
 (statistics_sap_opened,
  statistics_sap_entry_exists,
  sds_header_not_included,
  open_statis_sap_insuf_resrc)

ost$ampm_time = string (8)

ost$date = record
  case date_format: ost$date_formats of
  = osc$month_date =
    month: ost$month_date,          { month DD, YYYY
  = osc$mdy_date =
    mdy: ost$mdy_date,              { MM/DD/YY
  = osc$iso_date =
    iso: ost$iso_date,              { YYYY-MM-DD
  = osc$ordinal_date =
    ordinal: ost$ordinal_date,      { YYYYDDD
  = osc$dmy_date =
    dmy: ost$dmy_date               { DD/MM/YY
  casend,
recend

ost$date_formats =
 (osc$default_date,
  osc$month_date,
  osc$mdy_date,
  osc$iso_date,
  osc$ordinal_date,
  osc$dmy_date)

ost$dmy_date = string (8)

ost$hms_time = string (8)

ost$iso_date = string (10)
```

```
ost$mdy_date = string (8)

ost$millisecond_time = string (12)

ost$month_date = string (18)

ost$name = string (osc$max_name_size)

ost$ordinal_date = string (7)

ost$time = record
  case time_format: ost$time_formats of
  = osc$ampm_time =
    ampm: ost$ampm_time,                { HH:MM AM or PM
  = osc$hms_time =
    hms: ost$hms_time,                  { HH:MM:SS
  = osc$millisecond_time =
    millisecond: ost$millisecond_time,  { HH:MM:SS.MMM
  casend,
recend

ost$time_formats =
 (osc$default_time,
  osc$ampm_time,
  osc$hms_time,
  osc$millisecond_time)

pmt$program_name = ost$name

port_owner_type =
 (device_available,
  hdlc_owner,
  x25_owner,
  lcm_owner)

pref_type =
 (absolute@,
  conditional@,
  yield@)

priorities = 0 .. max_priority

qds_parm_type = record                  { stop (quit) port service
  ip_number:  0..upper_ip_num,                    { card slot number of device
recend

qps_parm_type = record             { stop (quit) port service
  dvmid_ptr:   ^cell,               { pointer to dvmid block
recend
```

```
report_time_type = record
  start:  ost$hms_time,          { HH:MM:SS }
  ending:  ost$hms_time,         { HH:MM:SS }
recend

root = record
  num_tables: integer,           { number of tables in tree
  num_nodes: integer,            { total number of nodes in the tree
  dump_id: string (4),           { validity check, should contain user value
  type_node: key_type,           { how is tree accessed
  link: ^node,
recend

sap_id_type = 0 .. 0ffff(16)

sat$max_dump_size = 0 .. 4096

sat$recovery_block = record
  procedure_address: ^procedure,  { pointer to code and static link address
  sa_dump_identifier: ^cell,      { sat$dump_identifier, ptr to dump control block
  previous_link: ^sat$recovery_block, { previous recovery block on stack
recend

sds_header = record
  sds_buf1_ptr: ^cell, { User defined collection buffers
  sds_buf2_ptr: ^cell,
  group: statistics_group_type,
  log_msg_number: 0 .. 0ffff(16),
  log_template_id: template_id_type,
  function_proc: statistics_function_procedure,
  display_proc: display_procedure,
  next_header: ^sds_header,
  collecting: boolean, { Collecting statistics & next reporting
  collecting_buf1: boolean, { TRUE=buffer1 FALSE= buffer2
recend

sds_parm_type = record           { Start Device Service
  ip_number:        0..upper_ip_num,         { card slot number of device
  num_ports:        0..max_number_ports,     { number of I/O ports on dvc
  buf_lev:          1..max_buf_level,        { number of buffers in pool
  stat_q_size:      16..max_stat_q_size,     { size of status queue
  cmd_q_size:       16..max_cmd_q_size,      { size of command queue
  load_module_ptr:  ^cell,                   { ptr to first byte of load module
  load_module_len:  integer,                 { length of load module
recend

sps_parm_type = record           { Start Port Service
  ssr_tid:          taskid,                  { task id of ssr
  ip_number:        0..upper_ip_num,         { board slot number of device
  lim_num:          0..upper_lim_num,        { LIM number if board is a CIM
  lim_port:         0..upper_port_num,       { LIM port number
  itm_exp_mask:     itm_exp_mask_type,       { mask for express intertask msg
recend
```

software_sap_range = 1 .. 0ffff(16)

stack_size = min_stack_size .. max_stack_size

statistics_function_codes =
 (issue_report_and_clear_buffers,
  clear_buffers,
  start_collecting,
  stop_collecting,
  select_buffer1,
  select_buffer2)

statistics_function_procedure =
```
  ^PROCEDURE ( {
      sds_hdr_ptr: ^sds_header;
      function_code: statistics_function_codes;
      reason: statistics_reason_type;
      time: report_time_type;
      param: ^cell;
  VAR status: statistics_function_status)
```

statistics_function_status =
 (stat_funct_success,
  stat_funct_insuf_resources,
  stat_funct_sw_err)

statistics_group_type =
 (st_summary,
  st_expanded,
  st_debug)

statistics_priority = medium_priority { Statistics priority

statistics_reason_type =
 (periodic_report,
  start_reporting,
  stop_reporting,
  close_sap,
  forced_report)

statistics_type =
 (comm_line,
  nw_solution,
  sw_component)

system_id_type = record
```
  upper: 0 .. 0ffff(16),
  lower: integer,
recend
```

```
system_status_table_type =
 (major_card_table_type,
  lim_table_type,
  port_table_type,
  smm_bank_table_type,
  pmm_bank_table_type)

task_attributes = record
  stack_allocation: stack_size,
  task_priority: priorities,
  preemptable: boolean,
  immediate_control: boolean,
recend

template_id_type = cme$min_template_id .. cme$max_template_id

test_status =
 (no_fault,
  fault,
  test_in_progress,
  unused)

threshold_size = 0 .. 7

timer = record
  next_one: ^timer,          { next timer in queue
  length: 0 .. 32767,        { length of what follows
  code: 0 .. 15,             { identifying code
  tod: milliseconds,         { time of day to pop
  period: milliseconds,      { period,if periodic timer
  param: ^cell,              { parameter for subroutine
  proc: ^procedure,          { address of subroutine
  mark: integer,             { = '!TIM'
recend

type_of_io =
 (no_io,
  read_io,
  write_io,
  intack_io)

type_of_memory =
 (pmm_bus,
  itb_bus,
  mpb_random,
  no_bus)
```

Procedure Types D

Some of the procedures in chapter 10 have parameters that point to procedures. These nested procedures are defined in this appendix, using the same format as the procedures presented in chapter 10. Procedures from chapter 10 that contain nested procedures are:

```
FIND_FIRST                             SFIND_NEXT
FIND_NEXT                              SFIND_WILD_CARDS
LOAD_ABS_MODULE_AND_PROCEED            START_NAMED_TASK_AND_PROCEED
LOAD_CMD_PROCESSOR_AND_PROCEED         START_SYSTEM_TASK
LOAD_ENTRY_POINT_AND_PROCEED           VISIT_ALL_NODES
SFIND_FIRST
```

Below are nested procedure definitions:

From **FIND_FIRST, FIND_NEXT, SFIND_FIRST** and **SFIND_NEXT:**

Input

ptr: ^cell

Address of the table associated with the node being tested.

param: ^cell

Address of the parameter being passed to this procedure.

Output

bool: boolean

Indicates whether or not the table being tested satisfies the **param** parameter.

From **LOAD_ABS_MODULE_AND_PROCEED:**

Input

request_id: ^cell

Address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

absolute_module_found: boolean

Indicates whether or not the called module is found.

smm_address: ^cell

Starting address of the module in SMM.

load_address: dlt$68000_address

Address where the module will be loaded.

transfer_address: dlt$68000_address

Address at which module execution begins.

byte_size: dlt$section_length

Size of the module in bytes.

error_response: clt$status

Any error messages from the Online Loader.

Output

None.

From **LOAD_CMD_PROCESSOR_AND_PROCEED** and **LOAD_ENTRY_POINT_AND_PROCEED:**

Input

request_id: ^cell

Address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

entry_point_found: boolean

Indicates whether or not the module entry point was found.

entry_address: ^dlt$entry_description

Starting address of the module entry point.

task_info: task_attributes

A record specifying stack size, priority, and whether or not the task is preemptible.

error_response: clt$status

Any error messages generated by the Online Loader.

module_ptr: dlt$load_id_ptr

Address of a record containing information about the module.

Output

None.

From **SFIND_WILD_CARDS:**

Input

table: ^cell

Pointer to user table.

params: ^cell

Pointer to user parameters.

Output

quit_processing: boolean

Indicates whether wild card search is to be terminated.

From START_NAMED_TASK_AND_PROCEED:

Input

request_id: ^cell

Address of an identifier. If the calling procedure is making more than one request for the module, this identifies the request.

task_started: boolean

Indicates whether the task was started.

task_id: task_ptr

The task identifier of the started task.

error_response: clt$status

Any error messages generated by the Online Loader.

Output

None.

From START_SYSTEM_TASK

Input

request_id: ^cell

User request identifier to link request and response.

task_id: _ptr

Task identifier of task started.

Output

None.

From VISIT_ALL_NODES:

Input

p: ^cell

Pointer to user table.

key: integer

Associated node key.

param: ^cell

Pointer to user parameters.

Output

more: boolean

If TRUE, continue search; if FALSE, terminate search.

Vector Table Usage E

The vector table is found in the first 400(16) bytes of DI memory. It is defined in common deck CMDVECT. The vector table is used as follows during CDCNET operation (refer to Motorola's M68000 User's Manual for more information).

Vector	Use
0	Reset: Initial System Stack Pointer. Label RESETSP
1	Reset: Initial PC
2	Bus Error
3	Address Error
4	Illegal Instruction
5	Zero Division
6	Check Instruction
7	Trap V Instruction
8	Privilege Violation
9	Trace
10	Line 1010 Emulator. Unimplemented op code
11	Line 1111 Emulator. Unimplemented op code
12-23	Reserved for future enhancements by Motorola
24	Spurious. For when the interrupt cycle has been started but cleared before completion
25	Level 1 Interrupt Autovector. Reserved for possible use on the 68000 Extension Bus.
26	Level 2 Interrupt Autovector. Real Time Clock Interrupt
27	Level 3 Interrupt Autovector. Software Timers and Clocks and Attention Switch
28	Level 4 Interrupt Autovector. ISB Interrupts (scanned) 8 cards (Control Bus Vector)
29	Level 5 Interrupt Autovector. Extension Bus
30	Level 6 Interrupt Autovector. SSC (Serial Port)
31	Level 7 Interrupt Autovector. Errors. Level 7 interrupts are non-maskable. "ACLOW" will indicate potential power failure, cause status to be saved, and then stop. "ERRORS" will include over-temperature condition.
32	TRAP 0 : fast_bg (also called maybe_bg) (background)
33	TRAP 1 : sure_bg

34	TRAP 2 : fast_fg (foreground)
35	TRAP 3 : fire in. Saves registers. Controlled recovery point. If another vector is invoked then TRAP 3 sets up firewall.
36	TRAP 4 : fire out. Resets firewall. If no task to preempt then it restores registers and returns from exception.
37	TRAP 5 : set_interval
38	TRAP 6 : set_slice
39	TRAP 7 : reserved for executive
40	TRAP 8 : used by MCI
41	TRAP 9 : reserved for I/O subsystem (for cards)
42	TRAP A : reserved for I/O subsystem "
43	TRAP B : reserved for I/O subsystem "
44	TRAP C : reserved for I/O subsystem "
45	TRAP D : used by DI Resident Debugger
46	TRAP E : reserved for I/O subsystem "
47	TRAP F : used by DVM
48-63	Reserved for future enhancements by Motorola
64	time slice
66	time interval
65, 67, 69, 71, 73, 75, 77, 79	SCCVECT (used by DI Debugger) (SCC)
68, 70, 72, 74, 76, 78	CIO User Interrupt Vectors
80-127	Expansion
128-255	Available for major boards. Eight vectors alloted for each of 16 possible board slots.

Intertask Message Workcode Definitions F

Following are the workcodes for intertask messages, their respective values and explanations.

Comdeck **CMDITM**

CYBIL definitions:

```
    {================================================
    { Executive Intertask Message Workcode Definitions
    {================================================

,exec_iptfail1               = 0000(16) { Bus/address error in interrupt
,exec_iptfail2               = 0001(16) { Other error in interrupt
,exec_tskfail1               = 0002(16) { Bus/address error in task
,exec_tskfail2               = 0003(16) { Other error in task
,exec_extent_gone            = 0004(16) { Memory extent vanished
,exec_stoptask               = 0005(16) { Stop Task
,exec_aborttask              = 0006(16) { Abort Task
,exec_new_vector_owner       = 0007(16) { New vector owner
,exec_jour_msg               = 0008(16) { Journal message
,exec_dest_failed            = 0009(16) { Destination failed
,exec_excess_slice           = 000a(16) { Excess Slice
,exec_error                  = 000b(16) { MPB failure error for system_ancestor
,exec_end_of_day             = 000c(16) { End of day message to timer task
,exec_new_time               = 000d(16) { New time of day request for timer task
,exec_periodic_timer         = 000e(16) { Periodic timer request for timer task
,exec_after_interval         = 000f(16) { After interval timer request for timer
                                        { task
,exec_at_time                = 0010(16) { Call at time request for timer task
,exec_periodic_after         = 0011(16) { Periodic request after interval for
                                        { timer task

    {================================================
    { Command ME Intertask Message Workcode Definitions
    {================================================

,c_me_msgcode                = 0014(16) { Command Processor I/F task
,c_me_respcode               = 0015(16) { Response to clp_process_command
,c_me_xport_msg              = 0016(16) { Command from transport I/F
,c_me_3b_msg                 = 0017(16) { Command from internet I/F
,c_me_cp_task_abort          = 0018(16) { Command processor abort
,c_me_cp_task_stop           = 0019(16) { Command processor stopped
,c_me_command_err            = 001A(16) { Command-ME processing error
,c_me_load_cmd               = 001B(16) { Load command processor

    {================================================
    { Routing ME Intertask Message Workcode Definitions
    {================================================

,r_me_full_update_lcrds      = 0030(16) { Update Least Cost Routing Data Store
,r_me_part_update_lcrds      = 0031(16) { Partial update to LCRDS
,r_me_ridu_msg               = 0032(16) { Routing Information Data Unit message
,r_me_3a_nw_update           = 0033(16) { Routing 3A Network Update message

    {===============================================
    { Error ME Intertask Message Workcode Definitions
    {===============================================

,err_me_internet_error       = 0039(16) { Internet error message

    {===========================
    { Independent File Access ME
    {===========================

,ifa_wkcode                  = 0040(16) { independent file access initialization
```

```
     {====================================
     { Console Driver Workcode Definitions
     {====================================

,console$traffic                 = 0050(16) { Transmit message
,console$configuration           = 0051(16) { Startup configuration
,console$write_complete          = 0052(16) { Completion of transmission sequence
,console$read_complete           = 0053(16) { Message has been received
,console$read_correct            = 0054(16) { Message received for editing

     {===================================
     { Online Loader Workcode Definitions
     {===================================

,dlc$load_abs_delay              = 0060(16) { Load absolute module
,dlc$load_abs_proceed            = 0061(16) {
,dlc$load_entry_point_delay      = 0062(16) { Load relocatable module
,dlc$load_entry_point_proceed    = 0063(16) {
,dlc$start_task_delay            = 0064(16) { Load relocatable module and
,dlc$start_task_proceed          = 0065(16) { Initialize as a task
,dlc$load_module_for_retain      = 0066(16) { Load module
,dlc$load_cmd_proc_delay         = 0067(16) { Load a command_processor
,dlc$load_cmd_proc_proceed       = 0068(16)

     {===================================================
     { DVM Intertask Message Command and Response Constants
     {===================================================

,dvm_response_base               = 0100(16) { Offset for dvm responses
,dvm_line_configure_res          = 0101(16) { Line configured status
,dvm_line_reconfigure_res        = 0102(16) { Line configuration response
,dvm_line_delete_res             = 0103(16) { Delete line response
,dvm_line_enable_res             = 0104(16) { Line enabled response
,dvm_line_disable_res            = 0105(16) { Line disabled response
,dvm_data_input_res              = 0106(16) { Input response
,dvm_data_output_res             = 0107(16) { Output response
,dvm_terminate_io_res            = 0108(16) { Line termination response
,dvm_line_status_res             = 0120(16) { Line status response
,dvm_trap_res                    = 0121(16) { DVM trap occurred
,dvm_timer_expired               = 0122(16) { DVM heart beat timer expired
,dvm_line_suspended              = 0123(16) { DVM has suspended service to an IP
,dvm_line_resumed                = 0124(16) { DVM has resumed previously suspended
                                            { service
,dvm_line_terminated             = 0125(16) { Service to a line has been terminated
,dvm_ip_dead                     = 0126(16) { Intelligent peripheral has reported
                                            { dead
,dvm_restart_ip                  = 0127(16) { Request restart IP service
,dvm_abort_ip                    = 0128(16) { Request abort IP service
,dvm_unexpected_interrupt        = 0129(16) { Unexpected interrupt

     {====================================================
     { HDLC Intertask Message Command and Response Constants
     {====================================================

,hdlc_command_base               = 0200(16) { HDLC ssr command base
,hdlc_link_up_cmd                = 0201(16) { Physical link initialization
,hdlc_link_down_cmd              = 0202(16) { Physical link down
,hdlc_connect_cmd                = 0203(16) { Logical link initialization
,hdlc_link_idle_cmd              = 0204(16) { Logical link idle
,hdlc_disconnect_cmd             = 0205(16) { Logical link disconnect
,hdlc_low_buffer_cmd             = 0206(16) { Buffer levels are low
,hdlc_nrml_buffer_cmd            = 0207(16) { Buffer levels are normal
,hdlc_wake_up_cmd                = 0208(16) { NOP message to wake up SSR (used by 3A)
,hdlc_status_cmd                 = 0209(16) { Provide sender with status information
,hdlc_configure_cmd              = 020b(16) { Physical and logical link configuration
,hdlc_reconfigure_cmd            = 020b(16) { Same as configure_cmd
,hdlc_start_stats_cmd            = 020d(16) { Begin statistics collection
,hdlc_report_stats_cmd           = 020e(16) { Report statistics
,hdlc_stop_stats_cmd             = 020f(16) { Discontinue statistics collection
,hdlc_i_timeout_cmd              = 0220(16) { I frame time out
,hdlc_p_timeout_cmd              = 0221(16) { P/F recovery attempt time out
,hdlc_e_timeout_cmd              = 0222(16) { Error recovery attempt time out
,hdlc_ioc_timeout_cmd            = 0223(16) { IP response time out
,hdlc_a_timeout_cmd              = 0224(16) { Activity time out
,hdlc_ia_timeout_cmd             = 0225(16) { Inactivity time out
,hdlc_rty_ex_cmd                 = 0226(16) { Retry count exceeded
,hdlc_ret_ex_cmd                 = 0227(16) { Retransmit attempt count exceeded
```

```
    {======================================================
    { ESCI Intertask Message Command and Response Constants
    {======================================================

,esci_command_base              = 0300(16) { Command base for ESCI
,esci_startup_cmd               = 0301(16)
,esci_shutdown_cmd              = 0302(16)
,esci_suspend_cmd               = 0303(16)
,esci_resume_cmd                = 0304(16)
,esci_statistics_cmd            = 0305(16)
,esci_wakeup_cmd                = 0306(16)
,esci_switches_cmd              = 0307(16)
,esci_tdr_cmd                   = 0308(16)
,esci_diag_cmd                  = 0309(16)
,esci_nop_cmd                   = 030a(16)
,esci_dvmid_cmd                 = 030b(16)
,esci_dump_cmd                  = 030c(16)
,esci_xsub_cmd                  = 030d(16)
,esci_nures_res                 = 0320(16)
,esci_rcv_res                   = 0321(16)
,esci_xmit_res                  = 0322(16)
,esci_ststc_res                 = 0323(16)
,esci_switches_res              = 0324(16)
,esci_tdr_res                   = 0325(16)
,esci_diag_res                  = 0326(16)
,esci_nop_res                   = 0327(16)
,esci_dump_res                  = 0328(16)
,esci_xsub_res                  = 0329(16)

    {======================================================
    { System Ancestor Intertask Message Workcode Definitions
    {======================================================

,sa_start_task_for_user         = 0400(16) { Start a task on behalf of another task
,sa_reply                       = 0401(16) { Call start_system_task reply routine
,sa_dump_write                  = 0402(16) { Write data to dump file
,sa_dump_timer                  = 0403(16) { Time out dump processing
,sa_dump_close                  = 0404(16) { Close dump file
,sa_dump_restore                = 0405(16) { Start dump processing and restore task
,sa_dump_only                   = 0406(16) { Start dump processing (no restore)


    {====================================================
    { System Audit Intertask Message Workcode Definitions
    {====================================================

,sys_audit_checksum             = 0450(16) { Checksun system memory
,sys_audit_overflow             = 0451(16) { Check user stack pointer for overflow
,sys_audit_report_the_mpb_status = 0452(16) { Check battery and temperature

    {================================================================
    { Mainframe Channel Interface Intertask Message Workcode Definitions
    {================================================================

,mci_startup                    = 0501(16) { Specific MCI card
,mci_output_complete            = 0502(16) { PP has successful read
,mci_input_received             = 0503(16) { PP has successful write
,mci_data_available             = 0504(16) { Data is available for transfer
,mci_error_encountered          = 0505(16) { An error was found on a write
,mci_shutdown                   = 0506(16) { End processing
,mci_statistics                 = 0507(16) { Sender requests statistics
,mci_report_statistics          = 0508(16) { Announce statistics response
,mci_link_status_change         = 0509(16) { New link status
,mci_timer_expiration           = 050a(16) { Response timer has expired
,mci_log_message                = 050b(16) { Message is to be logged
,mci_failure_detected           = 050c(16) { Failure detected
,mci_run_diagnostics            = 050d(16) { Run diagnostics

    {=======================================================
    { Initialization ME Intertask Message Workcode Definitions
    {=======================================================

,ime_pdu                        = 0601(16) { 3A indication parameters
,ime_transient_timer_expired    = 0602(16) { Transient task timer expired
,ime_init                       = 0603(16) { Transient task initialization
,ime_last_itm                   = 0604(16) { Transient task's last message
,ime_request_empty_itm_q        = 0605(16) { Request for ime_last_itm [*verify?*]
,ime_inactive_timer_expired     = 0606(16) { Main task timer expired
```

```
    {=====================================================
    { XEROX Transport Intertask Message Workcode Definitions
    {=====================================================

,xt_transmit                   = 0700(16) { Transmit delayed data
,xt_retransmit                 = 0701(16) { Retransmit normal data
,xt_expedited_retransmit       = 0702(16) { Retransmit expedited data
,xt_inactivity                 = 0703(16) { Send a probe or kill the connection
,xt_cid_timer                  = 0704(16) { Kill previously disconnected con'ctn
,xt_incoming_data_to_cep       = 0705(16) { Process packet for a connection
,xt_incoming_data_to_sap       = 0706(16) { Process packet for a sap
,xt_local_disconnect           = 0707(16) { Kill connection due to local action
,xt_set_up_timer               = 0708(16) { Set up connection timer

    {=====================================================
    { CDCNET Statistics Manager Message Workcode Definitions
    {=====================================================

,csm_issue_statistics_req      = 0800(16) { Request statistics to be reported
,csm_process_timer_req         = 0801(16) { Process statistics timer call

    {=================================================================
    { Operator Support Application Intertask Message Workcode Definitions
    {=================================================================

,osa_from_operator             = 850(16) { Command indication from operator
,osa_from_transport            = 851(16) { Indication from transport
,osa_from_internet             = 852(16) { Indication from internet
,osa_terminate_osa             = 853(16) { Kill OSA
,osa_configure_osa             = 854(16) { Initialize OSA
,osa_cmd_response_time_expired = 855(16) { Command time limit expired
,osa_cmd_proc_cmd_indication   = 856(16) { Command notice to OSA command processor
,osa_broc_response_time_expired = 857(16) { Broadcast command time limit expired
,osa_alarm_data                = 858(16) { Alarm indication from Dep  Alarm ME
,osa_format_message            = 859(16)  { Formatting workcode


    {==========================================================
    { K Display Supervisor Intertask Message Workcode Definitions
    {==========================================================

,kdisp_wkcode                  = 888(16) { Used to bring up k_display_supervisor

    {=============================================================
    { Log Support Application Intertask Message Workcode Definitions
    {=============================================================

, lsa_log_request_workcode       = 900(16) { Request for logging
, lsa_log_connect_retry          = 901(16) { Retry due to transport connect failure
, lsa_alarm_connect_retry        = 902(16) { Retry due to transport connect failure
, lsa_log_directory_indication   = 903(16) { Logging directory indication
, lsa_alarm_directory_indication = 904(16) { Alarming directory indication
, lsa_log_transport_indication   = 905(16) { Logging transport indication
, lsa_alarm_transport_indication = 906(16) { Alarming transport indication
, lsa_log_formatting_workcode    = 907(16)

    {======================
    { SSR Intertask Message
    {======================

,ssr_init_ok_workcode           = 980(16) { SSR initialization completed ok
,ssr_init_error_workcode        = 981(16) { SSR initialization error
,ssr_init_start_port_service_err = 982(16) { SSR start port service error
,ssr_init_queue_cim_command_err = 983(16) { SSR queue cim command error
,ssr_shutdown_error_workcode    = 984(16) { SSR shutdown error
,ssr_shutdown_ok_workcode       = 985(16) { SSR shutdown ok workcode
,ssr_reset_timer_req_workcode   = 986(16) { SSR reset request workcode
,ssr_timeout_workcode           = 987(16) { SSR timed out workcode
```

```
    {====================================================================
    { Configuration Status Reporter Intertask Message Workcode Definitions .
    {====================================================================

,csr_report_time               = 1050(16) { Rime to report configuration status

    {================================================
    { Clock ME Intertask Message Workcode Definitions
    {================================================

,ck_sync_clock                 = 1060(16) { Synchronize clock
,ck_sync_complete              = 1061(16) { Synchronization attempt complete
,ck_stop_independent_clock     = 1062(16) { Cancel independent clock on this system
,ck_disconnect_connection      = 1063(16) { Disconnect this connection
,ck_start_clock                = 1064(16) { Start independent clock on this system
,ck_clock_started              = 1065(16) { Independent clock started
,ck_clock_stopped              = 1066(16) { Independent clock stopped
```

DI Reset Codes G

Table G-1 lists all DI reset codes, in numerical order. For more information on the DI reset codes, refer to appendix B of the CDCNET Network Analysis manual.

Table G-1. DI Reset Codes

Major Category	Code Name	Code Number	Component Generating Reset Code
Hardware	power_up_reset	00(16)	MPB ROM
	manual_reset	02(16)	MPB ROM
	halt_memory_fault	03(16)	MPB ROM
	dead_man_time_out	04(16)	MPB ROM
Software	load_software_too_big	10(16)	Initialization Bootstrap
	improper_first_module	11(16)	Initialization Bootstrap
	unsatisfied_external	12(16)	Initial Loader
	sysconfig_not_loaded	13(16)	Initial Loader
	post_load_routines_not_found	14(16)	Initial Loader
	reset_at_end_of_quiesce	15(16)	Initialization Bootstrap
	unrecognizable_object_text	16(16)	Initial Loader
	duplicate_entry_point	17(16)	Initial Loader
	task_error_no_recovery_proc	18(16)	System Ancestor
	task_error_exceed_max_recovers	19(16)	System Ancestor
	task_error_unrecoverable	1A(16)	System Ancestor
	no_configuration_file_obtained	1B(16)	Configuration Procurer
	configuration_file_read_error	1C(16)	Configuration Procurer
	not_enough_memory_for_buffers	1D(16)	Loader
	identification_record_expected	1E(16)	Loader
	unexpected_idr_encountered	1F(16)	Loader
	premature_eof_on_file	20(16)	Loader
	absolute_length_too_large	21(16)	Loader
	invalid_object_text_version	22(16)	Loader
	invalid_module_kind	23(16)	Loader
	invalid_module_attribute	24(16)	Loader
	invalid_section_ordinal	25(16)	Loader
	duplicate_section	26(16)	Loader
	invalid_section_kind	27(16)	Loader
	invalid_allocation_alignment	28(16)	Loader
	invalid_offset	29(16)	Loader
	storage_allocation_failed	2A(16)	Loader
	undefined_section	2B(16)	Loader
	reference_outside_of_section	2C(16)	Loader
	invalid_address_kind	2D(16)	Loader
	invalid_number_of_bytes_spanned	2E(16)	Loader
	transfer_sym_entry_pt_not_found	2F(16)	Loader
	parameter_verification_error	30(16)	Loader
	loader_table_not_found	31(16)	Loader
	kill_system_with_dump	32(16)	KILL_SYSTEM command
	kill_system_without_dump	33(16)	KILL_SYSTEM command

(Continued)

Table G-1. DI Reset Codes *(Continued)*

Major Category	Code Name	Code Number	Component Generating Reset Code
	stop_executive	34(16)	Executive - S/W error
	module_checksum_is_invalid	35(16)	System Audits
	software_dead_stop	36(16)	Dead stop - S/W error
	smm_double_bit_error	37(16)	Executive - H/W error
	ac_low_error	38(16)	Executive - H/W error
	temperature_shutdown_error	39(16)	Executive - H/W error
	reset_from_debugger	3A(16)	Hardwired in Debugger
	overflowed_stack	3B(16)	Executive/System Audits
	system_data_not_found	3C(16)	Initial loader
	boot_file_media_mismatch	3D(16)	Boot startup code
	cybil_detected_error	3E(16)	CYBIL runtime routines

Index

Index

Comments (continued from other side)

Please fold on dotted line;
seal edges with tape only.

FOl

FOLD

FOl

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technology & Publications Division
ARH219
4201 N. Lexington Avenue
Arden Hills, MN 55126-6198

We value your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

Who are you?	**How do you use this manual?**
☐ Manager	☐ As an overview
☐ Systems analyst or programmer	☐ To learn the product or system
☐ Applications programmer	☐ For comprehensive reference
☐ Operator	☐ For quick look-up
☐ Other ______	

What programming languages do you use? ______

How do you like this manual? Check those questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?

Comments? If applicable, note page and paragraph. Use other side if needed.

Would you like a reply? ☐ Yes ☐ No

From:

Name	**Company**
Address	**Date**
	Phone

Please send program listing and output if applicable to your comment.

CONTROL DATA