INTERFACE AND INTEGRITY FACILITY
DMERT/UNIX® RTR OPERATING SYSTEM
AT&T 3B20D COMPUTER

CONTENTS PAGE

CONTENTS PAGE

Printed in U.S.A.

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

1. GENERAL

1.01 This section provides general functional descriptions and information related to the various interface and integrity facilities provided by the duplex multienvironment real-time (DMERT) operating system or UNIX RTR operating system used in the 3B20D computer. Also included in this section is a description of the plant measurements system (PMS).

1.02 This document is reissued to include interface and integrity facility changes brought about by the introduction of the Central Output Mechanism Feature (NI.043) for UNIX RTR Release 1 that provides the capability to put UNIX RTR output messages in a central database. This feature has the dual aim of centralizing output messages to facilitate converting their English text to other natural languages for applications with non-English-speaking customers and to allow the manipulation of the alarm level and message class associated with the output message. The Central Acknowledgement Mechanism Feature (NI.043C), an add-on to the Central Output Mechanism feature, provides the centralizing of input message acknowledgements. Revision arrows are used to emphasize the significant changes. The Equipment Test List is not affected. The specific reasons for reissue are listed below:

(a) Change Part 5 to provide a description of the Central Output Mechanism Feature (NI.043), UNIX RTR Release 1 and the add-on Central Output Mechanism Feature (NI.043C)

(b) Add Figures 6, 7, and 8 to provide a pictorial representation of the central output message data base (OMDB) on disk

(c) Add Figure 9 to show the layout of a .ofmt file in the OMDB

(d) Add Figures 10 and 11 to show the UNIX RTR OMDB key assignments and application key assignments, respectively

(e) Add Figure 12 to show the coordinator of the Spooler Output Process (CSOP) high level design with the OMDB

(f) Add Table C to show the OMDB key file layout

(g) Change the original Part 5 to Part 6.

This document applies to DMERT generic 1 and 2 and UNIX RTR Release 1 (formerly DMERT generic 3). This document also has the UNIX RTR craft consistency feature.

1.03 The operating system software integrity subsystem is flexible enough to handle all aspects of software integrity (ie, audits, defensive check errors, and overload conditions). In DMERT generic 1, the system integrity monitor (SIM) is responsible for overload conditions, the audit manager (AUDMGR) is responsible for audit control, and defensive check errors are left to the discretion of the associated processes. Effective with DMERT generic 2 and UNIX RTR Release 1, the SIM is responsible for scheduling and dispatching all audits and for handling overload conditions that could affect system integrity. All essential software integrity data is stored in the equipment configuration database (ECD).

1.04 The craft interface subsystem is composed of a number of hardware, firmware, and software entities that provide a maintenance person with communication and/or control over the computer. Communication is in the form of terminal messages and/or virtual panel displays (and other types of displays).

1.05 The following hardware components support the craft interface subsystem:

- Maintenance teletypewriter peripheral controller (MTTYPC)
- Emergency action interface (EAI)
- Power switch
- Maintenance terminal
- Maintenance printer (receive-only printer)
- Optional additional TTY peripheral controllers (TTYPC) and attached craft interface terminals.

Figure 1 shows the configuration of the craft maintenance interface hardware.

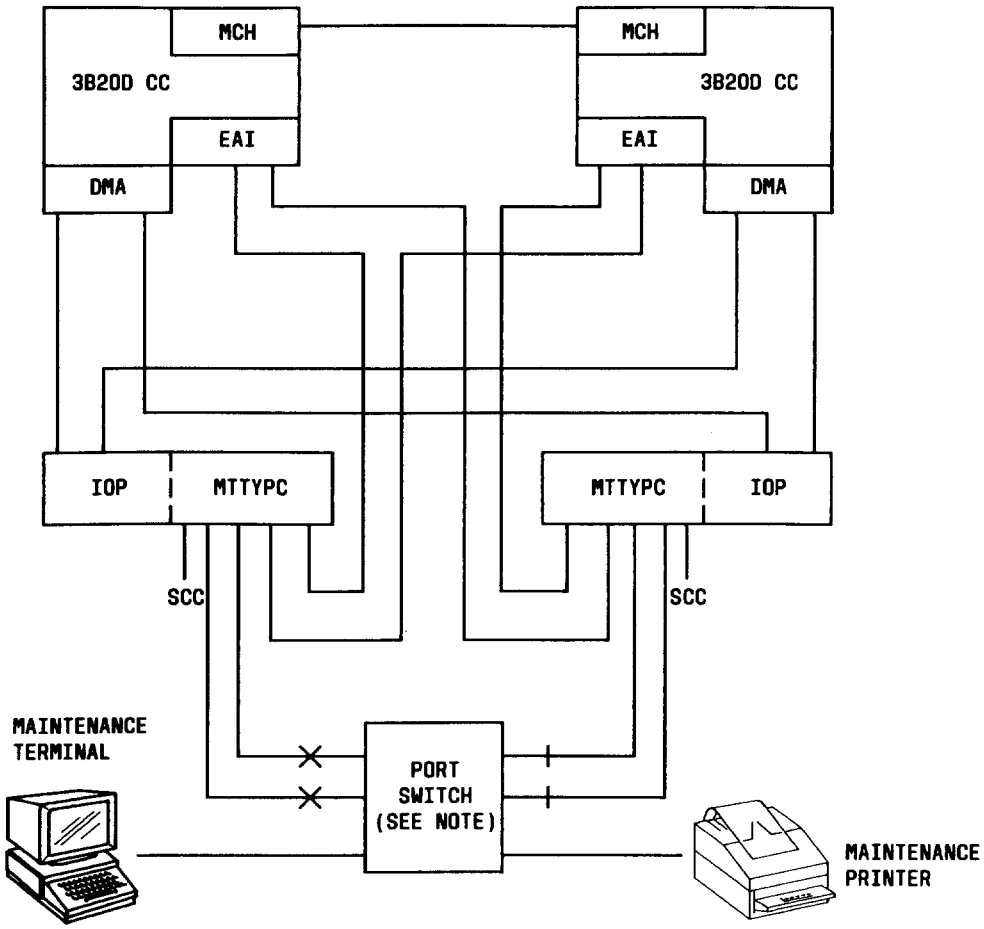
1.06 The craft interface subsystem is composed of the following major software entities:

- EAI firmware—Provides communication to/from the computer, executes EAI commands, and performs audits and self-tests
- Craft Interface Handler (a portion of the input/output processor)—Controls the information flow between the MTTYPC, any other TTYPCs with craft interface terminals, and the rest of the operating system (except for EAI in DMERT generics 1 and 2)
- Program Documentation Standards (PDS) Shell—Parses input TTY messages and invokes the appropriate processes to handle the requests
- Man Machine Language (MML) shell for DMERT generic 2 and UNIX RTR Release 1—Parses input TTY messages and invokes the appropriate processes to handle the requests
- Output Spooler—Routes output messages to the appropriate terminals, printers, and/or log files
- Display Administration Process (DAP)—Generates virtual panel or graphic displays for the maintenance terminal
- Real-Time Status Reporting Process— Monitors system status and reports the current status to the maintenance terminal
- Power Switch Administration—Includes the power switch monitor and scan administrator; provides monitoring and control over requests related to the power switches.

2. SYSTEM INTEGRITY

SYSTEM INTEGRITY MONITOR (DMERT GENERIC 1)

2.01 The SIM is responsible for the software integrity of DMERT operating system. The application integrity monitor (AIM) is responsible for the



LEGEND:
CC - CENTRAL CONTROL UNIT
DMA - DIRECT MEMORY ACCESS
EAI - EMERGENCY ACTION INTERFACE
IOP - INPUT/OUTPUT PROCESSOR
MCH - MAINTENANCE CHANNEL
MTTYPC - MAINTENANCE TELETYPEWRITER PERIPHERAL CONTROLLER
SCC - SWITCHING CONTROL CENTER
NOTE: THERE ARE TWO DISTINCT SWITCHES
1) FOR MAINTENANCE TERMINAL
2) FOR MAINTENANCE PRINTER

Fig. 1—Craft Maintenance Interface Configuration

integrity of all application software. The SIM performs the following functions:

- Receives software fault reports and initiates corrective action
- Creates and is responsible for the fault recovery of the DMERT operating system boot processes
- Serves as the main interface for the exchange of integrity information between the DMERT operating system and all applications
- Resets the hardware sanity timer and administers the application sanity timer
- Creates integrity processes

- Maintains a record of errors.

The SIM is a kernel boot process that is initiated by the DMERT kernel immediately after the error interrupt handler (EIH) is initialized during a system bootstrap. The SIM executes at priority level 13, which makes it the highest priority DMERT process excluding the EIH, generic access package, and processes executing in a critical region (ie, processes that execute at priority levels 14 and 15).

A. DMERT Interfaces

2.02 The main function of SIM is to receive software integrity exception data (ie, information regarding software faults) from DMERT operating system software and initiate corrective action.

2.03 DMERT-owned audits, overload detection checks, DMERT boot processes, and other processes detect software integrity faults and report them to SIM.

2.04 ***Interface to DMERT-Owned Audits:*** All audits report status information (success or failure) directly to the AUDMGR. The AUDMGR is responsible for the management of failure data and failure thresholds for all audits. When a failure threshold for a particular audit is exceeded, the AUDMGR sends an audit integrity exception message to the SIM. The message identifies the process which detected the failure. Audit failures which can be corrected by the audit are not reported as failures to SIM, but are recorded in a system error file. The corrective action taken by SIM depends on the process that caused the failure. In the event that the AUDMGR sends an invalid message, an audit is run on the AUDMGR. If failures which exceed the failure threshold are detected during the audit of the AUDMGR, then the SIM will terminate and recreate the AUDMGR process. DMERT audits that were active at the time are rescheduled. However, application-specified audits will not be rescheduled, but the SIM will notify the AIM of this event via a message.

2.05 ***Interface to DMERT Boot Processes:*** The SIM is responsible for the creation and fault recovery of the following DMERT operating system boot processes:

- File manager
- Disk driver
- Process manager
- Error interrupt handler
- Capability manager
- Scheduler
- Utility manager
- Memory manager.

2.06 When a system initialization is initiated, the SIM is notified via an event by the DMERT kernel. All boot processes are then expected to send the SIM an initialization status message (ie, success or failure) within 30 seconds. Also, the boot processes will inform the SIM of any integrity faults encountered during initialization. Thus, basic DMERT integrity can be defined as the reception of all expected successful initialization status messages from DMERT boot processes by the SIM.

2.07 The initialization status message contains the process number of the boot process and an EAI output message. The SIM computes the EAI step number based on the boot process number. The SIM then forwards the EAI output message, along with the step number, to the EAI.

2.08 As each successful initialization status message is received by the SIM, an output message is sent by SIM to the EAI. The EAI displays the message on the maintenance terminal as a success processor recovery message (PRM).

2.09 If a boot process cannot successfully initialize, a failure initialization status message is sent to notify the SIM of system error. The appropriate information is sent to the EAI to be displayed on the maintenance terminal as a failure PRM. The SIM requests another system initialization.

2.10 If the SIM does not receive initialization status messages from all boot processes within 30 seconds, the SIM will request another system initialization. No prior notification of this initialization will be given to the applications.

2.11 The SIM does not determine the scheduling of the boot processes. The application should ensure that the DMERT boot processes are not suspended from running by their own processes. This

division of responsibility is to give as much decision-making power to the application as possible. The decision to make an initialization of any degree should be made by the application.

2.12 ***Interface to the Overload Monitors:*** When the SIM receives an overload indication (via a fault or message from a system process or overload monitor), the SIM will inform the AIM of the overload condition.

2.13 ***Other Interfaces:*** The SIM also interfaces to portions of the craft interface subsystem for the exchange of integrity information with the applications and maintenance persons. Since these DMERT interfaces are involved in the application interfaces, they are described as such.

B. Application Interfaces

2.14 The SIM serves as the main interface for the exchange of integrity information between the DMERT operating system and all application integrity processes. The SIM will process application requests and instructions. Most messages sent to applications by DMERT will be handled by the SIM. The SIM interface to the application is via the AIM. The AIM is a kernel process, running at the same priority as SIM. In addition, the SIM will handle messages from the maintenance terminal via the craft interface subsystem.

2.15 ***System Initialization Message Requests:*** An application may request a copy of the initialization message from the last system initialization by sending a message to the SIM. The SIM is a memory resident data structure built by various initialization processes during initialization. The SIM contains:

- A copy of the requested initialization parameters (if the initialization was the result of a program request)
- The manual request parameters from the EAI
- The value of the real-time clock when the initialization was initiated
- The value of the initialization level counter
- An indication of the source of the initialization (hardware, craft, program request, etc.)
- The value of the application initialization level counter.

Integrity Instructions

2.16 The SIM receives DMERT system integrity commands from the craft interface PDS shell (PDSHL) in the form of messages. These commands pertain to audits, the overload system, and other integrity functions. The SIM either interprets and processes the instructions or forwards the message to the appropriate process.

2.17 ***Interface to the Audit Manager:*** The SIM provides a maintenance person/application process interface to the AUDMGR. Audit command messages from either the craft interface PDSHL or application processes are forwarded to the AUDMGR. The commands for audits include:

- Suspending future execution of an audit or all audits
- Rescheduling and resuming execution of a suspended audit
- Requesting the immediate execution of a suspended audit
- Requesting the immediate termination of an audit or all audits.

2.18 ***Interface to Craft Interface Subsystem:*** Audit and overload commands from the PDSHL are sent to the SIM, which then forwards the messages to the appropriate processes. Error recording is handled by the SIM using the output spooler.

C. Corrective Actions

2.19 The corrective actions taken by the SIM depend upon the failing process. All processes in DMERT are either essential or nonessential. The DMERT boot processes are essential processes. All other processes are generally considered nonessential by the SIM.

2.20 ***Essential Process Action:*** When the SIM receives an audit failure report from an essential process, the SIM requests a system initialization.

The SIM uses an initialization level of 1 with no application initialization level specified. Since the system initialization strategies are based upon previous initialization history, the SIM allows the built-in fault recovery strategies to determine the actual level of initialization.

2.21 ***Nonessential Process Action:*** The corrective action initiated by the SIM in response to an audit failure report from a nonessential process is to send the process manager a message to terminate the faulty process. The process manager terminates the process and sends a termination message to the process responsible for creating the faulty process.

D. Sanity Timers

2.22 The SIM is responsible for resetting the computer hardware sanity timer and administering the application sanity timer.

2.23 ***Hardware Sanity Timer:*** The integrity of the SIM and the EIH, as well as any application processes executing at levels 13 to 15, are maintained by the hardware sanity timer. The hardware sanity timer of the on-line control unit (CU) is set to time out in 1000 milliseconds. The hardware sanity timer of the off-line CU (if available) is set to time out in 1600 milliseconds (maximum). It is the responsibility of the SIM to reset the hardware sanity timers every 800 milliseconds (ie, the processes executing at levels 13 to 15 must execute in less than 200 milliseconds). If a hardware sanity timer times out, then the integrity of the SIM is questionable and a system initialization will occur. Fault recovery of the SIM is the responsibility of computer microcode and is part of the hardware fault recovery strategy. The SIM is not responsible for its own integrity (with the exception of resetting the timers). The AIM should be placed at the same execution level as the SIM. Thus, the sanity of the application monitor is dependent upon the hardware sanity timer in a similar manner as the SIM and EIH.

2.24 ***Application Sanity Timer:*** The purpose of the application sanity timer is to insure that application processes receive a share of the scheduled processor execution time. The application sanity timer is activated by the AIM via a message to the SIM. The message specifies the time interval (in milliseconds). Within that time interval, an application process must send the SIM an application sanity event. If the timer expires before the event arrives, the SIM will assume application software insanity and cause a system initialization.

E. Creation of Integrity Processes

2.25 The SIM is responsible for the creation and subsequent restart (in case of termination) of critical DMERT processes. These processes include:

- Audit manager
- Overload monitor
- User-level process monitor.

The user process monitor is used to monitor application-user processes.

F. Error Reporting and Logging

2.26 The SIM reports error conditions to the maintenance terminal via the EAI or the output spooler. The SIM error logging strategy uses the craft interface output spooler to record all integrity error conditions.

2.27 ***EAI Interface:*** During system initialization, the SIM reports error conditions using PRMs via the EAI.

2.28 ***Output Spooler Interface:*** The SIM sends output error messages to the maintenance terminal, the maintenance printer, and the switching control center (SCC) data link via the output spooler. The SIM error message has the format

REPT SIMCHK a b

where "a" is the SIM error code and "b" is supplementary data. The supplementary data is binary information that may be examined to provide additional information. The occurrence of this message (and any supplementary data) is recorded in the SIM log file.

2.29 ***Error Logging:*** All system error conditions are recorded in the SIM log file. In addition to the SIM, the DMERT audits and overload monitor also record occurrences of errors in this file. The entries in the SIM log file are

- SIMER — SIM error conditions

- AUDER — audit system error conditions
- OVLDER — overload system error conditions.

G. UNIX Level Automatic Restart Process

2.30 The UNIX level automatic restart process (ULARP) is a UNIX process whose functions are to create and monitor critical user processes and to restart automatically any monitored processes that terminate.

2.31 ***Initialization of ULARP:*** During a bootstrap procedure, SIM creates a process named /prc/unix. In the pcreate message sent to the process manager, the field pm_chan is set to BOOTCHAN (defined in the header file const.h), which indicates to the UNIX system ***init*** function that this is a bootstrap procedure. The ***init*** function then executes the ULARP process. When SIM receives the acknowledgement message on the successful creation of the /prc/unix process, SIM logs the process ID and waits for am E_SENDMSG event (defined in the header file simdef.h) from ULARP. This event causes SIM to send ULARP a start-up message, type USTARTUP (defined in header file simmsg.h). This message contains SIMBOOT in the message text indicating a bootstrap procedure and other system information needed for a successful ULARP start-up.

2.32 ULARP executes the run command file (/etc/rc or /etc/mcrc). When it has completed execution of the run command file, it sends a message with type RCINFO (defined in header file simmsg.h) to SIM. This message informs SIM of the status (either SUCCESS or FAIL) of the execution of the run command file. This status will be given to ULARP when ULARP is restarted and when ULARP is instructed to reread its process name file.

2.33 ***ULARP Failure Reports:*** After ULARP has executed the run command file, ULARP creates and monitors critical UNIX level processes that are designed to run continuously. The pathnames of ULARP child processes are contained in process name file /etc/ularpfile. Upon execution, ULARP stores the pathnames of its child processes in an internal table. If one of the processes dies, ULARP attempts to restart it under most conditions. The craft interface integrity monitor (CMON), one of ULARP child processes, creates and monitors the craft interfaces processes. Their pathnames are contained in the cmon.p file.

2.34 SIM receives an event from ULARP, when ULARP encounters a failure condition. SIM reports these failures to the craft using PRMs.

2.35 ***ULARP Termination:*** SIM receives a death-of-child message on the ULARP process if either the UNIX system "init" sequence fails to create ULARP or if ULARP dies. When this message is received, SIM usually assumes that ULARP was prematurely terminated and will attempt to restart it. However, if the termination message is received before the E_SENDMSG event indicating a probable failure to execute ULARP, SIM will not attempt to restart ULARP. If ULARP is not restarted, SIM displays a PRM to indicate the reason for ULARP failure.

2.36 ***Restarting ULARP:*** SIM will attempt to restart the ULARP process under the following three conditions:

- ULARP is prematurely terminated, such as during a phase level 1. If this occurs, SIM receives a death-of-child message and immediately attempts to restart ULARP.
- The maintenance person enters the PDS/MML shell command INIT:ULARP on the maintenance terminal. When this command is entered, the event E_UNIT (defined in simdef.h) is sent to SIM. When this event is received, SIM checks ULARP process ID. If ULARP is running, a message with type UREREAD (defined in simmsg.h) is sent to ULARP, instructing ULARP to reread the process name file and attempt to start all monitored processes that are not executing. If ULARP is not running, SIM attempts to restart ULARP.
- The maintenance person enters the **CFT-INIT** option (command code 15) on the emergency action page (Fig. 2) of the maintenance terminal. The **CFT-INIT** option causes the EAI handler to send SIM the fault FLT_CFT. If ULARP is running, SIM terminates all craft processes and sends ULARP an UREREAD message. If ULARP is not running, SIM restarts ULARP.

```
NAME      TYPE      GENERIC                          <C>       05/02/81  07:50:02
SYS EMER  CRITICAL  MAJOR    MINOR     BLDG/PWR   BLDG INH  CKT LIM   SYS NORM
TRAFFIC   SYS INH     CU     CU PERPH    LINK

CMD:                                          ——— EMERGENCY ACTION PAGE ———
CU-0 [ACT] [RUN]                  MTTY 7
CU-1                              EAI-0 [ASW]     PRM-0 0100 3200 0000 0DDF 19 C2 10
SCCS                              EAI-1 [ASW]     PRM-1 00C0 0005 0001 0DDF 19 C2 0C

            SET CLR          CU-0 CU-1  SET CLR
10 FONL-0    20 21 PRI-DISK              [30] 31 BACKUP-ROOT [SET]      50 APPL
11 FONL-1    22 23 SEC-DISK              [32] 33 MIN-CONFIG [SET]       51 INIT
12 FONL-ACT  24 25 INH-TIMER [INH] [INH]  34 [35] INH-HDW-CHK          52 BOOT
13 CLR-FONL  26 27 PRM-TRAP               36 [37] INH-SFT-CHK          53 BOOT+ECD
                                          38 [39] INH-ERR-INT          54 BOOT+MEM
14 CLR-EAI   28 PRM-DUMP                  40 [41] INH-CACHE            55 LDTAPE-0
15 CFT-INIT                               42 [43] APPL-PARAM           56 LDTAPE-1
```

a. For DMERT generic 2 or UNIX RTR Release 1 with TN83

```
NAME      TYPE      GENERIC                    ttya-cdA MTTY0  05/02/81  07:50:02
SYS EMER  CRITICAL  MAJOR    MINOR     BLDG/PWR   BLDG INH  CKT LIM   SYS NORM
TRAFFIC   SYS INH     CU     CU PERPH  OS LINKS

CMD:                                          ——— EMERGENCY ACTION PAGE ———
CU 0 [ACT] [RUN]                  MTTY 7
CU 1                              EAI 0 [ASW]     PRM 0 0100 3200 0000 0DDF 19 C2 10
SCCS                              EAI 1 [ASW]     PRM 1 00C0 0005 0001 0DDF 19 C2 0C

            Set Clr          CU 0 CU 1  Set Clr
10 Fonl 0    20 21 Pri Disk              [30] 31 Backup Root [SET]      50 Appl
11 Fonl 1    22 23 Sec Disk              [32] 33 Min Config [SET]       51 Init
12 Fonl Act  24 25 Inh Timer [INH] [INH]  34 [35] Inh Hdw Chk          52 Boot
13 Clr Fonl  26 27 PRM Trap               36 [37] Inh Sft Chk          53 Boot+ECD
                                          38 [39] Inh Err Int          54 Boot+Mem
14 Clr EAI                                40 [41] Inh Cache            55 Ldtape 0
15 Cft INIT  Last Appl Param              42 [43] Appl Param           56 Ldtape 1
```

b. For UNIX RTR Release 1 with TN983

Fig. 2—Emergency Action Page

2.37 After a successful ULARP restart, SIM terminates all craft processes by using the termclass OST to send an E_ABORT signal to the class DC_CFT (defined in the header file class.h). Afterward, SIM sends ULARP a start-up message. The start-up message contains USTART (defined in simmsg.h) in the message text, indicating to ULARP that this is a restart procedure. The message also contains the status of the execution of the run command file, as reported to SIM by the previous ULARP process.

AUDITS (DMERT GENERIC 1)

2.38 Audits detect and locate certain classes of errors. Normally, audits will attempt to correct software errors when discovered. The activation and the results of running each audit are recorded. Audits, controlled by the AMGR, are invoked by:

- Fault recognition and recovery programs used by either the AIM or the SIM
- Periodic checking of system software to detect latent faults
- The maintenance person from the maintenance terminal.

A count is maintained for each type of audit error detected.

A. Audit Manager

2.39 The AUDMGR is a supervisor process which serves as a scheduler and request administrator for audits. In order to provide a uniform control structure for the scheduling and result reporting of audits, the AUDMGR maintains certain data about the various audits in a tabular format. However, it does not maintain information on the function or structure of each individual audit. An entry in the AUDMGR data table defines attributes of an audit such as name, priority, and thresholds and provides memory space for counters.

2.40 The AUDMGR is created at bootstrap by the SIM. The AUDMGR then sends the SIM a message to acknowledge creation and then initializes the data table.

2.41 The DMERT AUDMGR is capable of initiating various audits. It does this on a routine basis according to priority. Scheduling of routine audits is on a relative priority basis rather than an absolute basis. The audit system does not guarantee that certain routine audits will run during a given interval, but it does guarantee that certain audits will be run more frequently than others. Overload control of routine audits is done automatically by the operating system since the AUDMGR is a supervisor process. Overload control can also be maintained as audits are disabled by the SIM.

2.42 A demand audit is initiated either by a manual request from the maintenance terminal or by a software request. The AUDMGR guarantees that demand audits be initiated on a first-come, first-served basis before any other routine audit is scheduled. There is no guarantee that a demand audit will be run immediately or will be run to completion without being interrupted. An audit runs at the execution level of the process in which the audit code appears.

2.43 The AUDMGR performs the following functions:

- Schedules audits
- Initiates audits
- Records the results of audits
- Inhibits audits
- Reports the results of audits.

2.44 ***Audit Scheduling:*** An audit is scheduled for execution by placing the audit on an execution list. The AUDMGR can accept requests for the scheduling of single or multiple audits. Several audits scheduled as a group in a specific order are called sequenced-mode audits. The AUDMGR must be provided the following information in order to schedule an audit:

- The process number of the process containing the audit code
- The relationship of the audit to a series of audits linked together (for sequenced-mode audits)
- The maximum values for error counts

- A timer for the clearing of counters after audit completion.

2.45 ***Audit Initiation:*** Audits are initiated asynchronously; i.e., the AUDMGR does not wait for the completion of each individual audit. Sequenced-mode audits are initiated in the specified order. However, the application can have sequenced-mode audits executed synchronously. In this case, the AUDMGR waits for completion of each audit in the sequence before initiating the next. The initiation of sequenced-mode audits can be aborted on request from the application. The AUDMGR can also add or delete a specific audit from a sequenced-mode chain of audits upon request.

2.46 ***Audit Results:*** Audits return a status to the AUDMGR upon completion. Audit-corrected errors are reported to audit control which maintains a record of the number of errors and maximum allowable values of these correctable errors. Software errors which cannot be corrected by the audit are also reported to audit control. The AUDMGR maintains counts of the occurrences of these types of errors and maximum allowable values associated with the counts.

2.47 ***Inhibit Audits:*** The AUDMGR can temporarily inhibit the execution of a specified audit on request. The AUDMGR can inhibit all audits, audits in a specific group, or audits with a specific priority. Also, the AUDMGR can reinstate the execution of inhibited audits on request. Status reports of all active and inhibited audits can be requested by the crafts person.

2.48 ***Reporting Audit Results:*** The AUDMGR does not report the result of each individual audit to the process requesting the report. The AUDMGR reports failure counts that exceed thresholds to the SIM. The report identifies the audit or group of audits that have failed and the actual number of failures recorded. After reporting, the counters are cleared. Other conditions for counter clearing are system initialization, audit scheduling, audit inhibition or removal, and after the time interval specified in the scheduling request.

2.49 The AUDMGR handles, in order of priority, all reports of audit completion, all demand audits, all audit scheduling, inhibit and removal requests, craft interface requests, and all routine audits.

SYSTEM INTEGRITY MONITOR (DMERT GENERIC 2 and UNIX RTR RELEASE 1)

2.50 The SIM is primarily responsible for the operating system software integrity and integrity interface with application software. The SIM performs the following functions:

- Administering the computer hardware sanity timer and the application sanity timer
- Ensuring the integrity of the operating system bootstrap initializations and generating boot process PRMs
- Creating and monitoring the ULARP and cooperating with it during INIT:ULARP and CFT-INIT procedures
- Monitoring the operating system overload conditions
- Administering and scheduling DMERT and UNIX RTR audits and initiating recovery action in response to audit errors
- Providing an interface between the operating system and the AIM.

The SIM is a kernel boot process initiated by the DMERT or UNIX RTR kernel immediately after the EIH is initialized during a system bootstrap. The SIM executes at priority level 13. The only operating system processes that execute at a higher level than SIM are EIH (level 15), generic access package (level 14), and processes temporarily executing in a critical region (level 15).

A. Bootstrap Initialization

2.51 SIM is responsible for ensuring that all essential boot processes are correctly initialized during a DMERT or UNIX RTR bootstrap process (phase levels 2, 3, and 4). Each boot process must send an initialization status message (SINITSTAT) to SIM within a specified time interval after SIM receives the E_INIT event from the DMERT or UNIX RTR kernel. The bootstrap processes are as follows:

- Error interrupt handler
- Disk driver

- File manager
- Process manager
- Scheduler
- Capability manager
- Memory manager
- Utility manager.

2.52 The operating system initialization interval is read from the SIM control record in the ECD. If SIM does not receive a status message that reports successful initialization from every essential boot process, SIM will output a failure (F) PRM and reboot the system at the end of the time interval. If software checks have been inhibited by the **INH-SFT-CHK** option (command code 36) on the emergency action page, SIM will not reboot the system.

2.53 ***Initialization Status Message:*** During the operating system intialization interval following a bootstrap, SIM must receive an initialization message from every essential boot process. SIM recognizes the essential boot processes by their fixed process numbers. SIM uses a global array of status flags (one for each process) to remember which processes have reported.

2.54 If SIM receives a valid status message with a status of FAIL, SIM will output all data entries in the message as F PRMs and request another system initialization. The panic code used in that initialization request will consist of two hexadecimal digits (1 followed by the PRM step number of the process that reported the initializtion failure).

2.55 When each status message is received with a status of SUCCESS, SIM sets the appropriate status flag (if it is an essential process) and generates an all-seems-well PRM with the appropriate step number. Nonzero data entries that are sent with SUCCESS messages are buffered for later output (after the operating system initialization interval is completed). Nonfatal error conditions detected by SIM during system initialization are also buffered for later output.

2.56 ***Initialization Check:*** SIM receives a time-out entry at the end of the initialization interval. SIM checks the array of status flags to verify that status messages have been received from all essential boot processes. If any boot process has not reported successful initialization, SIM outputs an F PRM with step number B, function code 00, and the eight process status flags (showing which one(s) did not report).

2.57 If all essential boot processes have reported successful initialization, SIM outputs an all-seems-well PRM with step number C. SIM then initiates the output of buffered data entries that were received with SUCCESS initialization messages as success (E) PRMs. These auxiliary PRMs are separated by intervals of 2.5 seconds in an attempt to ensure that all auxiliary PRMs be printed on the maintenance printer and transmitted to the SCC. As additional SUCCESS messages are received, the nonzero data entries are also placed into the SIM PRM buffer and output at 2.5-second intervals. This provides a mechanism for continuing to report the progress of system initialization beyond the end of the initialization interval.

2.58 ***AIM Initialization Check:*** After SIM has confirmed the successful initialization of the essential boot processes, SIM checks the value of the AIM initialization interval. If the AIM initialization interval is nonzero, then AIM must report to SIM within that interval. AIM will send SIM either an E_ASAN event or a message to activate the application sanity timer. If AIM has already activated the application sanity timer, SIM generates a boot progress PRM with function code 50. If AIM has not activated the application sanity timer, then SIM requests another timeout to occur at the end of the AIM initialization interval. If AIM has reported by the time SIM receives the timeout entry, SIM generates a boot process PRM. If AIM has not reported, then SIM generates an F PRM with function code 51 and requests another system initialization.

B. Sanity Timers

2.59 The SIM is responsible for resetting the computer hardware sanity timer and administering the application sanity timer.

Hardware Sanity Timer

2.60 SIM administers the hardware sanity timers in both control units (CUs) during normal system operation. SIM sets the hardware sanity timer in the on-line computer to time-out in 1000 milliseconds

and the off-line computer sanity timer to time-out in 1600 milliseconds. If a sanity timer times out, a maintenance reset function (MRF) will be generated by the computer hardware unless that signal has been disabled manually by selection of the **INH-TIMER** option (command code 24) on the emergency action page of the maintenance terminal. This inhibits only the MRF; this does not stop the sanity timer from counting. If the MRF has not been inhibited, a system initialization will occur.

2.61 SIM requests a timeout event every 800 milliseconds to reset the timers. This provides a 200-millisecond cushion that defines system insanity. More than 200 milliseconds of continuous execution by any process running at level 13 to 15 may cause the sanity timer to time-out with a resulting system initialization.

Application Sanity Timer

2.62 SIM provides a software sanity timer for use by AIM to ensure that application processes are sane and being scheduled by the operating system. This application sanity timer is activated by a message from AIM to SIM. This message specifies the timeout interval and the initialization levels to be used in a phase request if AIM fails to reset the sanity timer. Once the sanity timer has been activated, AIM must reset the sanity timer by sending SIM an E_ASAN event at least once in every timeout interval. If AIM fails to send this event during any timeout interval, SIM will generate a failure PRM and request a system reinitialization unless software checks have been inhibited by the **INH-SFT-CHK** option (command code 36) on the emergency action page of the maintenance terminal.

C. UNIX Level Automatic Restart Process (DMERT Generic 2)

2.63 ULARP is a UNIX process whose functions are to create and monitor critical user processes and to restart, automatically, any monitored processes that terminate.

2.64 ***Initialization of ULARP:*** During a bootstrap procedure, SIM creates a process named /prc/unix. In the pcreate message sent to the process manager, the field pm_chan is set to BOOTCHAN (defined in the header file const.h), which indicates to the UNIX system ***init*** function that this is a bootstrap procedure. The ***init*** function then executes the ULARP process. When SIM receives the acknowledgment message on the successful creation of the /prc/unix process, SIM logs the process identification (ID) and waits for an E_SENDMSG event (defined in the header file simdef.h) from ULARP. This event causes SIM to send ULARP a start-up message, type USTARTUP (defined in header file simmsg.h). This message contains SIMBOOT in the message text indicating a bootstrap procedure and other system information needed for a successful ULARP start-up.

2.65 ULARP executes the run command file (/etc/rc or /etc/mcrc). When it has completed execution of the run command file, it sends a message with type RCINFO (defined in header fie simmsg.h) to SIM. This message informs SIM of the status (either SUCCESS or FAIL) of the execution of the run command file. This status will be given to ULARP when ULARP is restarted and when ULARP is instructed to reread its process name file.

2.66 After ULARP has executed the run command file, ULARP creates and monitors critical UNIX level processes that are designed to run continuously. The pathnames of ULARP child processes are contained in process name file /etc/ularpfile. Upon execution, ULARP stores the pathnames of its child processes in an internal table. If one of the processes dies, ULARP attempts to restart it under most conditions. The CMON, one of ULARP child processes, creates and monitors the craft interfaces processes. Their pathnames are contained in the cmon.p file.

2.67 ***ULARP Failure Reports:*** SIM receives an event from ULARP when ULARP encounters a failure condition. SIM reports these failures to the craft using PRMs.

2.68 ***ULARP Termination:*** SIM receives a death-of-child message on the ULARP process if either the UNIX system "init" sequence fails to create ULARP or ULARP dies. When this message is received, SIM usually assumes that ULARP was prematurely terminated and attempts to restart it. However, if the termination message is received before the E_SENDMSG event indicating a probable failure to execute ULARP, SIM does not attempt to restart ULARP. If ULARP is not restarted, SIM displays a PRM to indicate the reason for ULARP failure.

2.69 ***Restarting ULARP:*** SIM will attempt to restart the ULARP process under the following three conditions:

- ULARP is prematurely terminated, such as during a phase level 1. If this occurs, SIM receives a death-of-child message and immediately attempts to restart ULARP.

- The maintenance person enters the PDS/MML shell command INIT:ULARP on the maintenance terminal. When this command is entered, the event E_UINIT (defined in simdef.h) is send to SIM. When this event is received, SIM checks ULARP process ID. If ULARP is running, a message with type UREREAD (defined in simmsg.h) is sent to ULARP, instructing ULARP to reread the process name file and attempt to start all monitored processes that are not executing. If ULARP is not running, SIM attempts to restart ULARP.

- The maintenance person enters the **CFT-INIT** option (command code 15) on the emergency action page of the maintenance terminal. The **CFT-INIT** option causes the EAI handler to send SIM the fault FLT_CFT. If ULARP is running, SIM terminates all craft processes and sends ULARP a UREREAD message. If ULARP is not running, SIM restarts ULARP.

2.70 After a successful ULARP restart, SIM terminates all craft processes by using the termclass OST to send an E_ABORT signal to the class DC_CFT (defined in the header file class.h). Afterward, SIM sends ULARP a startup message. The startup message contains USTART (defined in simmsg.h) in the message text, indicating to ULARP that this is a restart procedure. The message also contains the status of the execution of the run command file, as reported to SIM by the previous ULARP process.

D. UNIX Level Automatic Restart Process (UNIX RTR Release 1)

2.71 ULARP is a UNIX process whose functions are to create and monitor critical user processes and to restart automatically any monitored processes that terminate.

2.72 ***Initialization of ULARP:*** During a bootstrap procedure, SIM creates a process named /prc/unix. In the ***pcreate*** message sent to the process manager, the field pm_chan is set to BOOTCHAN (defined in the header file const.h), which indicates to the UNIX system ***init*** function that this is a bootstrap procedure. The ***init*** function then executes the ULARP process. When SIM receives the acknowledgment message on the successful creation of the /prc/unix process, SIM logs the process ID and waits for an E_SENDMSG event (defined in the header file simdef.h) from ULARP. This event causes SIM to send ULARP a start-up message, type USTARTUP (defined in header file simmsg.h). This mesage contains SIMBOOT in the message text indicating a bootstrap procedure and other system information needed for a successful ULARP start-up.

2.73 The run command files, ULARP's process file, and cmon.p input file for CMON have all been replaced by ECD records. The CMON program has been eliminated. ULARP executes and monitors its child processes and those formerly executed by CMON and executes but does not monitor each of the run commands.

2.74 ***ULARP Failure Reports:*** Error conditions are reported by spooler output messages if the spooler is running. If the spooler is not running, ULARP reports errors to SIM by means of SINITSTAT messages, which result in the output of PRMs to be output during a bootstrap or craft initialization procedure. All errors reported by ULARP are logged in the ULARPLOG log file.

2.75 ***ULARP Termination:*** SIM receives a death-of-child message on the ULARP process if either the UNIX system "init" sequence fails to create ULARP or ULARP dies. When this message is received, SIM usually assumes that ULARP was prematurely terminated and attempts to restart it. However, if the termination message is received before the E_SENDMSG event indicating a probable failure to execute ULARP, SIM does not attempt to restart ULARP. If ULARP is not restarted, SIM displays a PRM to indicate the reason for ULARP failure.

2.76 ***Restarting ULARP:*** SIM will attempt to restart the ULARP process under the following three conditions:

- ULARP is prematurely terminated, such as during a phase level 1. If this occurs, SIM receives a death-of-child message and immediately attempts to restart ULARP.

- The maintenance person enters the PDS/MML shell command INIT:ULARP! on the maintenance terminal. When this command is entered, the event E_UINIT (defined in simdef.h) is sent to SIM. When this event is received, SIM checks ULARP process ID. If ULARP is running, SIM sends the E_UINIT event to ULARP. This instructs ULARP to reread the database records and attempt to start any child processes which are not executing. If ULARP is not running, SIM attempts to restart ULARP.

- The maintenance person enters the **CFT-INIT** option (command code 15) on the emergency action page of the maintenance terminal. The **CFT-INIT** option causes the craft interface handler to send SIM the fault FLT-CFT. If ULARP is running, SIM sends the E_CFTERM event, warning that a craft initialization is going to take place. ULARP replies with an E_KILCFT event to SIM, instructing SIM to kill the craft processes. SIM terminates the craft processes and then sends an E_RSTCFT event to ULARP telling ULARP to restart the craft processes. At this time, ULARP restarts craft processes only. When all craft processes have been restarted, ULARP sends an E_CFTCOMP event to SIM indicating that the craft initialization is complete. Until SIM receives the completion event, SIM will not initiate any new craft initialization procedure. If ULARP is not running, SIM restarts ULARP, terminates the craft processes, and sends ULARP a USTARTUP message with the value CFTSTART. SIM will not kill any craft processes if disk limp mode is in effect.

2.77 If ULARP dies and is restarted, SIM does not terminate craft processes unless the restart is in response to a craft initialization request. SIM sends ULARP a startup message USTARTUP, with one of two values; USTART or CFTSTART. USTART results in ULARP attempting to adopt its former child processes. If an adopt call returns failure, ULARP will attempt to restart that process. CFTSTART result in ULARP restarting craft processes and attempting to adopt other child processes. If the adopt fails, ULARP will restart the process at this time.

AUDITS (DMERT GENERIC 2 AND UNIX RTR RELEASE 1)

2.78 The DMERT generic 2 and UNIX RTR Release 1 audit control systems are composed of the following:

- The System Integrity Monitor process is responsible for scheduling and dispatching all audits and for initiating audit error recovery procedures

- The Equipment Configuration Database (ECD) stores all software integrity information used by SIM. SIM uses software integrity control, audit control, and audit instance records to administer audits

- Audits that verify, correct system data structures, and recover lost system resources. Audits may reside in long-lived or transient processes

- Library functions that provide the interface between audit processes and the software integrity subsystem

- The System Integrity Output Formatter (SIOF) process provides a consistent output message format when reporting audit results to maintenance personnel

- Manual standard PDS/MML input commands (AUD, INH:AUD, OP:AUD, etc.) are provided to query and request software integrity services

- Requests from other processes to run audits or block audits

- Periodic status reports output a periodic REPT AUDSTAT message to inform maintenance personnel when audits are inhibited or blocked

- A Plant Measurements interface allows counts of audit attempts and failures to be stored in the plant measurements database.

2.79 ***Audit Identification:*** All audits are categorized by the type of data audited (ie, file manager, message buffer, and ECD). Each audit category is referred to as a family and each family has a name that is up to six characters long. The audit name should be mnemonic for the name of the system resources or facility to be audited. A member number is used to distinguish among audits in the same family.

2.80 In DMERT generic 2, each audit control record contains one or more audit instances and each audit instance may have a name that is up to 19 characters long. Audit instances maintain information specific to a particular occurrence of data to be audited. Because of ECD record size limitations, only four audit instances are permitted per audit control record. An audit that requires more than four audit instances is forced to use multiple audit control records.

2.81 In UNIX RTR Release 1, each audit control record is associated with one or more audit instance records. Each audit instance may have a name that is up to 19 characters long. Audit instances maintain information specific to a particular occurrence of data to be audited.

2.82 An audit with only one audit instance is identified by the audit name and audit member number of the audit. If an audit has more than one audit instance, the audit name, audit member number, and audit instance name identify the audit.

2.83 ***Correcting Audits:*** All DMERT and UNIX RTR audits should be capable of correcting any errors that audits detect. By default, audits should execute in error correcting mode. Audits must be flexible enough to execute in error detecting mode only. The default error correction mode of each audit is stored in the audit record.

Audit Control and Scheduling

2.84 When scheduling audits, SIM is required to limit the number of routine, manual, and software requested audits that may execute simultaneously. SIM is also required to control the amount of CU time audits use.

2.85 ***Audit System Initialization:*** Initialization of the audit control system after a system bootstrap requires SIM to access the system integrity control record, each audit and audit instance record in the ECD, and each audit record in the plant maintenance database. If any step in the initialization procedure fails, SIM outputs a message to inform the user. Whenever audit system initialization has failed, SIM recognizes the ALW:AUD:ALL input command as a request to attempt to reinitialize the subsystem and begin running routine audits.

2.86 ***Audit Segmentation:*** The audit control system supports segmented and nonsegmented audits. Segmented audits relinquish the CU after a predetermined amount of time. Most kernel level audits should execute in a segmented mode. When scheduled routinely or by a software request, a kernel level audit will execute one audit segment each time it is dispatched by SIM. When a segmented audit is requested manually, more than one segment may be executed each time it is dispatched.

2.87 Nonsegmented audits run to completion and report to SIM the amount of CU time used. All UNIX and supervisor level audits must be nonsegmented. Nonsegmented kernel level audits must compute and report the amount of CU time it uses. For UNIX and supervisor level audits, the audit interface library functions compute and report the CU time usage.

2.88 ***Audit Dispatching:*** Audits are dispatched using messages (messages accompanied by an E_AUD event and events). Most kernel level audits and all UNIX and supervisor level audits must be message dispatched. Effective with UNIX RTR Release 1, messages accompanied by an E_AUD event are provided as a performance improvement for kernel level audits only. Kernel level audits are the only event dispatched audits. Only the message queue audit is event dispatched.

2.89 ***Audit Execution Modes:*** The audit control mechanism supports routine, manual, software, and demand execution modes. Routine audits are executed at a given frequency or at specified times during normal system operation. Manual audits are manually requsted by maintenance input commands. Software audits are requested by processes other than maintenance input commands. Demand audits are demanded by SIM as a result of a system error.

2.90 The audit control system allows an audit to execute in any of the execution modes, depending upon the audit invocation. The permitted execution modes of each audit are specified in the ECD record for that audit.

2.91 ***Blocking Audit:*** The execution of audits can be blocked using explicit and implicit blocking. Explicit blocking occurs when a process specifically requests the blocking of an audit instance by calling the aud_block library function. This prevents the audit instance from being executed in any mode. If all the instances of an audit need to be blocked, the aud_block function must be called for each instance. Implicit blocking is used when scheduling audits. SIM ensures that no two audits with the same name are run simultaneously and no two instances of the same audit are run simultaneously.

2.92 ***Inhibiting Routine Execution of Audits:*** Inhibiting audits prevents them from being scheduled to run routinely or by software request. Separate inhibit states are provided for the following:

- All audits
- All instances of a specific audit
- A specific instance of an audit.

The INH:AUD and ALW:AUD commands are used to control audit inhibits. Setting or resetting the master inhibit state for all audits turns routine audit scheduling off and on. The inhibit states of individual audits or audit instances are not affected by setting or resetting the master inhibit state. Manual audit inhibits will not be allowed automatically by SIM. A periodic output message, REPT AUDSTAT, reports the inhibit status of audits when the system is running in full configuration.

2.93 ***Frequency Group Audits:*** Each frequency group audit is assigned a frequency labeled A through H, where A is the highest frequency and H is the lowest frequency. Each audit in a higher frequency group executes twice as often as an audit in the next lower frequency group. The frequency group of each audit instance is specified in the audit ECD record in DMERT generic 2 or in the audit instance record in UNIX RTR Release 1. Because the frequency group is instance dependent, different instances of the same audit may be assigned to different frequency groups.

2.94 ***Timed Audits:*** Audits that must be run at given times of the day should use the timed audit facility. Each audit instance contains information that allows the audit to be scheduled to execute at a specified hour and day. Different instances may be scheduled to run at different times or on different days. A given instance cannot be both a timed instance and a frequency group instance. However, the same audit may have one or more timed instances and one or more frequency group instances. When scheduling timed audits, schedule no more than one hour of work to be done in a single hour-long interval.

Audit Library Functions

2.95 SIM uses messages and events to dispatch audits as specified in their audit control records. Audit processes interface with SIM via audit interface library functions. The library functions perform the following:

- Assist audits in reporting errors and termination status to SIM
- Accumulate total error counts
- Determine how many error reports may be forwarded to SIM for output by the SIOF
- Advise an audit when an error correction limit is reached.

For audits that report supplementary error data, the library functions provide the supplementary data interface with SIOF. For kernel and special kernel audit processes, the library functions assist in reading the audit ECD record (if necessary).

2.96 The library functions implement two kinds of interfaces with SIM, depending upon the execution level at which the process is running. In processes below the level of SIM, the library functions use operating system trap (OST) calls to communicate with SIM. In processes running at or above the level of SIM, the library functions send messages to SIM.

2.97 The appropriate craft interface commands make OST calls to SIM to initiate the requested action. Other system processes call library

functions which send messages to SIM to request that audits be run or audit instances be blocked. SIM sends reply messages back to the requesters. The SIOF makes OST calls to SIM to retrieve data for audit output messages. SIOF is created and monitored by ULARP. In UNIX RTR Release 1, SIM inhibits the routine scheduling of audits if SIOF is not executed successfully at bootstrap time or terminates and is not restarted by ULARP.

OVERLOAD MONITOR

2.98 The overload monitor is responsible for monitoring the resources of the operating system and the reporting of overload conditions to the SIM. The overload monitor is actually a portion of the SIM, although under most conditions the overload monitor functions as an independent entity.

Overload Detection and Reporting

2.99 The overload monitor, with few exceptions, is not responsible for overload detection. As resources are allocated to DMERT or UNIX RTR processes, checks are made for overload conditions. As processes detect overload conditions, they are reported to SIM, which reports them to AIM and generates REPT SIMCHK output messages which are recorded in the system integrity log file.

2.100 The operating system detects overload conditions for the following resources and reports them by sending faults to SIM:

- Message buffer overflow
- Memory manager overflow
- Disk swap space overload
- Segment descriptor table overflow
- Dispatcher control table overflow
- File manager input request queue
- Disk file controller job queue.

2.101 ***Nonswappable Main Memory:*** The memory manager has a limit to the amount of main memory that can be filled with nonswappable processes. When creation of a new kernel process is requested, the memory manager checks that the allocation of memory to that process will not exceed that limit. The memory manager will monitor the amount of nonswappable main memory that is allocated. When the amount allocated reaches 80 percent of the maximum, the memory manager will send the fault OV_MEMLOW to SIM. When the amount allocated is reduced to 60 percent of the maximum, the memory manager will send the fault OV_MEMCLR to SIM. When creation of a kernel process fails due to insufficient memory, the memory manager will send the fault OV_MEMKPFL to SIM.

2.102 ***Nonswappable Module 1 Memory (UNIX RTR Release 1):*** There is a limited amount of memory available in the module 1 memory. The memory in this module is not swappable, thus having the potential to be overloaded. The memory manager will monitor the amount of module 1 memory in use. When the amount allocated reaches 80 percent of maximum, the memory manager will send the fault OV_MEM1LOW to SIM. When the amount allocated is reduced to 60 percent of the maximum, the memory manager will send the fault OV_MEM1CLR to SIM. If a request for module 1 memory cannot be granted because of a lack of pages, the fault OV_MEM1FUL is sent to SIM.

2.103 ***Disk Swap Space:*** The amount of swap space on disk currently in use is monitored to detect overload. When the amount used reaches 80 percent of the maximum space allocated, the memory manager will send the fault OV_SWAPLOW to SIM. When the amount is reduced to 60 percent of the maximum space allocated, the memory manager will send the fault OV_SWAPCLR to SIM.

2.104 ***Segment Descriptor Table Entries:*** Creation of any new process requires a minimum of three entries in the segment descriptor table. The maximum number of segment descriptor table entries required by a process is 128. Most processes use less than six entries. The number of entries in the segment descriptor table will be monitored to detect overload. When the number of free entries falls below 50, the memory manager will send the fault OV_SDELOW to SIM. When the number of free entries has recovered to 100 or more, the fault OV_SDECLR will be sent to SIM. When the creation of a new process or a new segment of an existing process fails due to insufficient free segment descriptor table entries, the fault OV_SDEPRFL will be sent to SIM.

2.105 ***Dispatcher Control Table Overload:*** The number of entries in the dispatcher control table will be monitored to detect overload. The DMERT and UNIX RTR kernel will track the number of entries in the dispatcher control table. When the entries in the dispatcher control table reach 70 percent of the maximum, the kernel will send the fault OV_DCTOVLD to the overload monitor. When the dispatcher control table entries are 100 percent allocated, the kernel will send the fault OV_DCTCRIT to the overload monitor. When the entries are reduced to 50 percent, the kernel will send the fault OV_DCTOK to the overload monitor.

2.106 ***File System Monitor:*** File system overload conditions are detected by a UNIX process that runs under ULARP. File System Monitor (FSMON) monitors the number of free blocks in each mounted file system. FSMON attempts to predict when a file system might run out of free blocks based upon the rate of decrease as a percentage of the number remaining. The file manager reports to SIM the name of any file system that is overflowing or that might soon overflow. The file manager sends an FSOVWARN message to SIM if the file system appears to be in danger of overflowing within 2 hours. The file manager sends an FSOVCRIT message to SIM if the file system has no free blocks left or free inodes. When data is removed from a file system that overflowed, the file manager sends SIM an FSOVCLR message. SIM forwards file system overload reports to AIM and generates REPT FS OVERFLOW output messages with the file system names.

2.107 ***Disk Driver Overload:*** The disk driver administers a separate job queue for each disk file controller (DFC) in the system. Overload on a DFC depends upon the number of jobs waiting for that DFC and the sizes of those jobs. A DFC will be considered overloaded if its internal queue of 64 jobs becomes full. When the DFC internal queue is reduced to 50 jobs, the overload condition is considered cleared. A DFC will also be considered overloaded if incoming jobs wait too long because pending jobs are too big. When the waiting time for new jobs reaches 7 seconds, the controller will be considered overloaded. When the waiting time is reduced to 5 seconds, the overload condition will be considered cleared.

2.108 The DFC will send the fault OV_DFCOVLD to the overload monitor when either of the above conditions occur on a DFC. Another fault will be sent if a second DFC becomes overloaded. When both of the above conditions are cleared for a DFC, the disk driver will send the fault OV_DFCOK to the overload monitor.

Real Time Overload Monitors

2.109 Process lockout conditions are detected by SIM with the help of two related monitor processes. KLMON is a kernel process that executes at level 3, which is the lowest level available to regular kernel processes. SUOVPRC is a supervisor process that executes at a fixed priority level below the priority at which ordinary UNIX processes execute. The KLMON and SUOVPRC report by sending events to SIM periodically in order to verify that processes are being scheduled by the operating system.

2.110 The KLMON process stops reporting to SIM when some other kernel process at or above level 3 (but below level 13) is using up all the CU time. This is called kernel level lockout. A SUOVPRC stops reporting to SIM when another supervisor or UNIX process is using too much CU time. This is called supervisor level lockout. SIM reports these lockout conditions by sending faults to itself and to AIM in exactly the same way that other overload conditions are reported.

2.111 Each KLMON detects a less severe type of real-time overload by measuring the time interval between its own timeout entries. KLMON reports an overload condition to SIM when its timeout entry is delayed by more than a specified amount.

E. Message Buffer Overload Recovery

2.112 The AIM is responsible for initiating the bulk of overload recovery action. The only overload condition for which SIM takes any recovery action is message buffer overload. Also, the SIM will notify applications when critical system resources exceed predetermined critical levels.

2.113 The DMERT and UNIX RTR kernel provides OSTs to recover message buffers from processes in order to relieve overload conditions. SIM uses the OSTs in the recovery procedures as follows:

- Count the message buffers that are on the input queues of selected processes
- Flush messages from the input queues of selected processes

- Terminate selected processes and free their message buffers.

SIM uses the spy library functions to obtain information from the system concerning message buffer allocation and the addresses and sizes of data structures included in a selective panic dump. If a kernel-level lockout condition exists when message buffer overload occurs, SIM requests a phase 1 reintialization in an attempt to break the lockout. If none of the attempted recovery actions succeed, SIM will repeatedly request a phase 1 until the DMERT fault recovery strategy escalates the initialization level to a phase 2 bootstrap. If this occurs, a selective panic dump will be written to disk. After the system has recovered, the data in the panic dump can be used to help determine what caused the overload.

3. PLANT MEASUREMENTS

3.01 The PMS provides a means by which the application can obtain measurements regarding the long-term performance of various entities of the DMERT or UNIX RTR operating system and the 3B20D computer. The computer is composed of the following hardware units:

- Control Unit (CU)
- Disk File Controller (DFC)
- Moving Head Disk (MHD)
- Input/Output Processor (IOP)
- Emergency Action Interface (EAI)
- Switching Control Center (SCC) Link
- Maintenance TTY Peripheral Controller (MTTYPC)
- Maintenance Terminal
- TTY Controller (TTYC)
- Teletypewriter (TTY)
- Magnetic Tape Controller (MTC)
- Magnetic Tape (MT)
- Maintenance Printer (receive-only printer)
- Direct User Interface Controller (DUIC)
- Direct User Interface (DUI)
- Synchronous Data Link Controller (SDLC)
- Synchronous Data Link (SDL)
- Scanner/Signal Distributor Controller (SCSDC).

3.02 The plant measurements provide information that can be used for:

- Determining when equipment needs servicing or replacing
- Determining appropriate thresholds for audits and fault recovery
- Indicating overall system reliability
- Evaluating the performance of the office and the craft.

A. Plant Measurement Structure

3.03 Essentially, the PMS is composed of the following entities:

- Measurement database/PMS library
- Plant daemon
- Operating system interfaces
- Application interface.

Measurements Database

3.04 In DMERT generic 1, the PMS measurement database has been implemented as a public library composed of two portions. The first portion is the database access primitives (Table A) used to access the data in the data portion of the measurement database. The second portion is the measurement data consisting of sets of measurements. In DMERT generic 2 and UNIX RTR Release 1, the access primitives and measurement data are maintained in two distinct entities (PMS library and measurement databases, respectively). The PMS library has been implemented as a public library containing database access primitives. The public library is locked in

primary memory to provide fast execution of the database access primitive.

3.05 ***Database Access Primitives:*** The database access primitives (functional calls) are composed of low level access (LLA) primitives that use a master outline to provide operating system and application access to the measurement data. The LLA primitives provide a means for preserving data integrity by allowing only one operating system process to update a record at a given time. These access primitives are accessible from both supervisor and kernel processes. User level processes require an additonal UNIX system function call in order to access the public library.

3.06 ***Measurement Data:*** The measurement data consists of sets of measurements, or counts, which are stored in the form of records. To retrieve information from the database, an application must read the individual set(s) for which it is interested in obtaining the information.

Plant Daemon

3.07 The plant daemon is a nonkillable kernel process of level 3 which is created during bootstrap. In software, a process that is referred to as a "daemon" controls information of other processes with unusual effectiveness. Since it is a kernel process, it is locked in primary memory. The plant daemon performs the following functions:

- Manages the measurements database
- Ensures that the measurement database is locked in primary memory
- Computes maintenance usage.

Operating System Interfaces

3.08 Several operating system processes use the access primitives to store performance measurement data in the measurement database. These operating system processes (followed by the type of information stored) are as follows:

(a) ***Error Interrupt Handler:*** This process keeps track of the source and level of all system initializations. It also maintains the record of each level of processor error interrupts.

(b) ***Alarm Control Process (DMERT Generic 1 and 2):*** This process controls the record maintained for each type of system alarm (except for power alarms).

(c) ***Coordinator of the Spooler Output Processor (UNIX RTR Release 1):*** This pro-

TABLE A

MEASUREMENTS DATABASE ACCESS PRIMITIVES

PRIMITIVE	DESCRIPTION
pl_attrec	Reads the specified record (if present) or creates a new record (used by operating system processes only)
pl_delete	Deletes the specified record (used by operating system processes only)
pl_init	Attaches the measurements database and data dictionary to the calling process (used by operating system and application processes)
pl_read	Reads the contents of the measurements database and copies all records into a specified buffer area (used by application processes only)
pl_update	Writes the contents of a specified buffer into the specified measurements database record (used by operating system)

cess controls the record maintained for each type of system alarm (except for power alarms).

(d) ***Power Switch Monitor:*** This process keeps a record of power alarm counts.

(e) ***Audit Manager (DMERT Generic 1):*** These processes keep a record of all audit attempts and failures.

(f) ***System Integrity Monitor (DMERT Generic 2 and UNIX RTR Release 1):*** These processes keep a record of all audit attempts and failures.

(g) ***Communications Protocol Handler:*** This process maintains a record of the unauthorized access attempts on individual data links.

(h) ***Configuration Control Process:*** This process keeps track of the number of instances and amount of time a unit is in the nonactive state. It also records faults and errors reported against specific units.

(i) ***Processor Control Process Audit:*** This process keeps track of the state of the CU.

Application Interface

3.09 The application interface consists of two plant function calls which are contained in the public library. They are the ***pl_init*** and ***pl_read*** function calls. Basically, the ***pl_init*** function call returns connection and set information used in the ***pl_read*** function call to read the database.

3.10 ***PL_INIT Function:*** The ***pl_init*** function call should be implemented as part of the application's initialization sequence. It is used to attach a process to the plant measurements database for the life of the process. If the application is a user level process, the UNIX system plib function call should be included in the initialization sequence prior to the ***pl_init*** call. The ***pl_init*** function call returns, through the function parameters, a copy of the set identifier record and information used on subsequent ***pl_read*** calls.

3.11 ***PL_READ Function:*** The ***pl_read*** function call is used to retrieve data from the database, i.e., to copy the records contained in the plant measurement set defined by the application and passed on the call into a buffer area. If successful, the ***pl_read*** call also returns a count of the number of records that were read from the database. This count makes it possible to see if additional records were added to or deleted from a particular measurement set. Only the records contained in the specified plant measurement set are returned by the ***pl_read*** function call. This allows the application the option of omitting those sets of records for which it has no need.

B. Measurements

3.12 The PMS provides eight basic record sets of information. These record sets are as follows:

- System initializations
- Alarms
- Audits
- Processor error interrupts
- Data links
- Data link groups
- Equipment information
- Set identifier.

System Initializations

3.13 System initializations can be initiated manually (requested by the maintenance person) through a software fault or through a hardware fault. There are also four different levels of severity associated with each initialization. The range of levels is from 1 to 4 (level 0 is transparent to the operating system and is not counted). Initialization levels 1 through 3 can be initiated in any of the three ways mentioned above. Level 4 initializations must be manually invoked by the maintenance person. There are ten possible combinations for which a count will be provided. Each count will represent the number of initializations made for a particular combination. The following list indicates the different types of system initialization combinations for which counts are maintained.

- Software initiated—level 1

- Software initiated—level 2
- Software initiated—level 3
- Hardware initiated—level 1
- Hardware initiated—level 2
- Hardware initiated—level 3
- Manually requested—level 1
- Manually requested—level 2
- Manually requested—level 3
- Manually requested—level 4.

3.14 Initializations are counted after they occur by a post-mortem dump process (Recovery Message Formatter). In the event of a rapid series of initializations, this process is able to read from memory the information pertaining to the first, second, third, and last initialization. Thus, in the case of rolling MRFs, only data concerning four initializations is recorded.

Alarms

3.15 There are three basic types of system alarms. They are critcal, major, and minor alarms. A count of the number of occurrences is made for each type.

3.16 Specific counts of power alarms are also kept. Since critical power alarms are not currently possible, separate counts will be maintained only for major and minor power alarms. The counters associated with each basic type of alarm (critical, major, and minor) are under the control of the alarm control process (DMERT generics 1 and 2) or CSOP (UNIX RTR Release 1). The PMS maintains control of the power alarm counts.

Audits

3.17 In DMERT generic 1, there are nine entities which are audited periodically and on request by the maintenance person. A count is maintained of each attempted audit and how many times an audit fails. A count of audit errors will not be maintained. An audit error is considered to have occurred when the number of audit failures exceeds a specified threshold. Audit counts maintained by the PMS are as follows:

- Audit manager audit attempts (DMERT generic 1)
- Audit manager audit failures (DMERT generic 1)
- Message buffer audit attempts
- Message buffer audit failures
- Scheduler audit attempts
- Scheduler audit failures
- Memory manager audit attempts
- Memory manager audit failures
- File manager audit attempts
- File manager audit failures
- Hardware audit attempts
- Hardware audit failures
- Application ECD audit attempts (DMERT generic 2 and UNIX RTR Release 1)
- Application ECD audit failures (DMERT generic 2 and UNIX RTR Release 1)
- ECD (disk copy) audit attempts (DMERT generic 2 and UNIX RT Release 1)
- ECD (disk copy) audit failures (DMERT generic 2 and UNIX RTR Release 1)
- Incore ECD audit attempts (DMERT generic 2 and UNIX RTR Release 1)
- Incore ECD audit failures (DMERT generic 2 and UNIX RTR Release 1)

3.18 In DMERT generic 2 and UNIX RTR Release 1, a PMS database record is created and maintained by the SIM process for each audit control record in the ECD using the same audit name and member number. This includes all audits provided by the DMERT or UNIX RTR system, plus any audits

added by the application to be administered by the DMERT/UNIX RTR audit control system. Each audit record in the PMS database contains an attempts counter and a failures counter for that audit. These counters are updated by SIM every time the audit is run.

3.19 All of the audit counters are controlled by the AUDMGR process (DMERT generic 1) and the SIM process (DMERT generic 2 and UNIX RTR Release 1).

Processor Error Interrupts

3.20 There are three levels of severity of processor error interrupts. The range of levels is from 0 to 2. Level 0 corresponds to on-line hardware errors. Level 1 corresponds to off-line error interrupts. Level 2 corresponds to software and memory management exceptions. Counts of the number of error interrupts which occur are maintained for each level described above. The EIH process records the counts associated with each level of processor error interrupt.

Data Links

3.21 A count is maintained of the number of unauthorized attempts to access an individual data link. Data link traffic information is also recorded. These counts are maintained by the communication protocol handler.

3.22 Initially, no records of this type will be in the database. Records in this set are created dynamically whenever an unauthorized access or traffic activity is detected. The record is identified by the complex name and number and the individual unit name and number as defined in the ECD.

Data Link Groups

3.23 A count is maintained of the number of occurrences of an input/output buffer overflow within a data link group. An overflow is considered to have occurred if a request is made for an input/ouput buffer and none are available. This count is maintained by the communication protocol handler. The record is identified by the logical device record (record type mdct) of the ECD.

Equipment Information

3.24 Maintenance measurements are recorded for each individual unit of restorable hardware. The measurements that are collected for equipment are the most involved of the PMS, both in terms of the number of units involved and the type of information collected.

3.25 Equipment information measurements can be broken down into two basic categories. They are maintenance usage information and forced active information.

3.26 ***Maintenance Usage Information:*** An error occurring in a piece of equipment can be real or possibly a fluke (a transient error). To determine whether or not a problem is real, thresholds are established in the ECD for each restorable unit. Within a specified time period, if the total number of errors which have occurred in a unit reaches the threshold of the unit, then that unit is faulted. The number of errors that contribute to a unit's threshold are considered transient errors. A count of these errors is maintained in the PMS.

3.27 It is sometimes necessary that a unit be placed in a nonactive state. The nonactive state for a unit may be requested for the following reasons:

- Routine maintenance by the automatic diagnostic process
- Manually initiated by the maintenance person
- Unit is faulted due to threshold being exceeded.

3.28 A separate record of both the amount of time and the number of instances a unit is in the nonactive state is maintained by the PMS for each of the reasons mentioned above. Thus, to find the total amount of time a unit was in the nonactive state and the total number of instances the unit was placed in the nonactive state, it is necessary to add together the three separate time records and instance counts, respectively. The configuration control process handles these counts.

3.29 ***Forced Active Information:*** As mentioned previously, when a unit is faulted, a request is made to place that unit in the nonactive state.

However, if the unit is essential in nature and a backup is not available, the unit cannot be placed in the nonactive state without jeopardizing the system. Two alternatives to the nonactive state request are available. For any essential piece of equipment without an available backup, a fault may be relieved by a system initialization. The second alternative applies only if the faulted piece of equipment is a CU. This unit can be manually forced active by the maintenance person.

3.30 The PMS maintains counts of the amount of time and the number of instances that a CU is placed in the forced active state. This state is also indicated on the common processor display on the MTTY.

3.31 Records concerning the states of restorable units are controlled by the configuration control process routines, which inform the PMS of changes in the state of a unit. The records maintained for the forced active state of the CU are controlled by the processor control process audit process which informs the PMS of the state of the CU.

3.32 The following list depicts the equipment information records maintained per unit:

(a) Transient errors—total number

(b) Maintenance usage information

- Automatic diagnostic process request—number of instances
- Automatic diagnostic process request—amount of time
- Manual request—number of instances
- Manual request—amount of time
- Threshold fault request—number of instances
- Threshold fault reqeust—amount of time.

(c) Forced active information—number of instances and amount of time for manual requests only. This information currently pertains only to the CU.

3.33 As with the data link record set, the database will be initialized with no records in the equipment information record set. The records will be created whenever a unit is taken out of service.

3.34 In DMERT generic 1, this set can have a maximum of 100 records. All counts in these records are preserved until a new version of the public library is brought into core after a level 2 or higher system initalization. All counts are cleared after such an initialization; thus, all data collected since the last initialization is lost. Once the entire capacity of the equipment information portion of the PMS database is used, attempts to create additional records results in a PL_RCRT_ERROR message being returned from the ***pl_attrec*** function. No additional records can be created until the PMS database is reinitialized with a level 2 or higher system intialization.

3.35 In DMERT generic 2 and UNIX RTR Release 1, the equipment information record set can have a maximum of 150 records. The PMS database is periodically (approximately every 15 minutes) copied to disk. A level 2 or higher system initialization restores the PMS database from its last disk copy. Thus the only data lost upon reinitialization is that which was collected since the last time the PMS database was copied onto disk. As with DMERT generic 1, attempts to create additional records after record capacity has been reached result in a PL_RCRT_ERROR message. Additional records can be created only after a level 2 or higher system initialization has reinitialized the PMS database.

Set Identifier

3.36 The set identifier record is the last record set in the plant measurements database. This record contains a set identifier for each set of measurements in the database (eg, system initializations, alarms, audits, etc.). These identifiers allow unique accessing of each individual set of measurements.

3.37 The set identifier record also contains a field that is written by the plant measurments daemon with the time that the database was brought into core. This allows an appliction to determine whether or not the database was initialized since the last time data was read from it by that application.

4. CRAFT INTERFACE

HARDWARE

4.01 The following paragraphs provide general descriptions of the hardware components of the craft interface subsystem.

A. Maintenance TTY Peripheral Controller

4.02 The MTTYPC provides access to the normal input/output channels of the IOP and the EAI (Fig. 1). The MTTYPC supports interfaces to the maintenance terminal, maintenance printer (which is a receive-only printer), and SCC.

4.03 For DMERT generic 1 and 2, the MTTYPC provides two independent channels. Each channel has two ports (Table B). For UNIX RTR Release 1, the MTTYPC provides four independent channels, one of which has two ports (Table B). Because MTTYPCs operate autonomously, conflicting commands from different MTTYPCs are executed on a first-come, first-served basis by the EAI. Coordination is required between the local office and the SCC system (SCCS).

MTTYPC/EAI Communications

4.04 Each MTTYPC communicates with both EAIs over a full-duplex serial communication link at 9600 baud. Communication with the EAIs is in the form of upper case American Standard Code for Information Interchange (ASCII). These ASCII messages represent either commands to the EAIs or acknowledgments/responses from the EAIs and status reports to the MTTYPCs.

4.05 Normally, the EAI does not initiate communication. The MTTYPC sends a command to the EAI. The EAI then checks the command, performs the requested actions, and returns an acknowledgement verifying the execution of the command. But for certain status changes, the EAI will initiate communication by signaling the MTTYPC via a control lead of the communication link. These status changes are:

- The RUN, ACTIVE, or EAI ENABLE status changed
- The EAI was unable to execute a command
- One or more EAI functions has arbitrarily changed state.

TABLE B

MTTYPC CHANNELS

HARDWARE	DMERT GENERIC 1 AND 2	UNIX RTR RELEASE 1
Maintenance Terminal	Channel 0	Channel 0
Emergency Action Interface 0	Channel 0 Port 1	Channel 2 Port 0
Emergency Action Interface 1	Channel 0 Port 1	Channel 2 Port 1
Switching Control Center	Channel 1 Port 0	Channel 3
Maintenance Printer	Channel 1 Port 1	Channel 1

Upon reception of a signal from the EAI, the MTTYPC will request status information from the EAI to determine the reason for the signal.

MTTYPC/Maintenance Terminal Communications

4.06 Each MTTYPC will communicate with the maintenance terminal over full duplex serial communications links. The local maintenance terminal link operates at 9600 baud, and the remote SCCS link operates at 2400 baud. The full 128-character ASCII set is used for communication with certain sequences reserved for control of the MTTYPC and the maintenance terminal. With the exception of control sequences, the MTTYPC will echo to the maintenance terminal.

MTTYPC Errors and Alarms

4.07 The MTTYPC reports to the input/output processor the following errors and communication failures:

- Parity fail
- Sanity fail
- Clock fail
- Routine diagnostic fail.

When these failures occur between the MTTYPC and the active CU, the failures are reported to special circuitry on the power monitor for pickup by the local office alarm grid. Communication failures between the MTTYPC and the CU or EAI are reported to the maintenance terminal and SCCS. A major alarm results when communication failures occur between the MTTYPC and the active CU. Communication failures between both MTTYPCs and the active CU result in a critical alarm. If a communiction failure occurs between the MTTYPC and the EAI, a report is sent to the active IOP which will determine the appropriate alarm level.

B. Emergency Action Interface

4.08 The EAI circuit pack provides a low-level maintenance access path from the maintenance terminal to the computer. The EAI is a small stand-alone microprocessor system. The major components of the EAI are:

- An Intel* microprocessor
- Read-Only Memory (ROM)
- Random-Access Memory (RAM)
- Two Universal Asynchronous Receiver Transmitter (UART) devices.

The EAI circuit pack is located in the processor frame and shares the power supply of the computer.

4.09 There are two EAIs, one associated with each computer. The EAI can function regardless of the current state of its associated computer. The EAI receives emergency action requests from the maintenance terminal or from a SCC via a MTTYPC port.

4.10 The EAI displays status information via light emitting diode (LED) indicators mounted on the faceplate of the EAI circuit pack. The following list describes these indicators.

- STATUS—Displays, as a single hexadecimal digit, the low four bits of the system status register
- RUN—Indicates that the associated central control (CC) is on-line
- ACTIVE—Displays the state of the processor on-line (PONL) signal from the associated computer
- FORCED ONLINE—Displays the state of the forced on-line (FONL) signal from the associated computer
- FORCED OFFLINE—Displays the state of the forced off-line (FOFL) signal from the associated computer
- EMERGENCY ACTION ENABLED—Displays the emergency action enabled (EAEN) status bit.

* Trademark of Intel Corporation.

C. Power Switch

4.11 The power switch circuit pack provides control over the power supply of the CC. This circuit pack is adjacent to the EAI circuit pack. The power switch administration displays status information via LED indicators mounted on the faceplate of the power switch circuit pack. Also mounted on the faceplate are the power switches. These indicators and switches are:

- ON switch—Turns the power converter on by initiating a power-up sequence
- OFF switch—Turns the power converter off by initiating a power-down sequence
- OFF indicator—Lighted when the power converter is off
- ALM indicator—Alarms lighted when the power converter is out of tolerance, a fuse is blown, an auxiliary circuit is faulty
- OOS indicator—Lighted when the power converter is out-of-service
- RQIP indicator—Lighted when a request initiated from one of the power switches is in progress
- ROS indicator—Lighted when the ROS/RST switch is in the Request out-of-service (ROS) position
- ROS/RST switch—Requests that the unit be taken out-of-service (ROS) or be restored to service (RST)
- ACO/T switch—Alarm cut-off or lamp test
- MOR switch—Manual override of the OFF/OOS interlock.

D. Maintenance Terminal

4.12 The maintenance person interacts with the computer primarily through the maintenance terminal. The maintenance terminal provides message input and output facilities. The maintenance terminal keyboard is used to enter PDS messages or MML (DMERT generic 2 and UNIX RTR Release 1) messages. The messages are found in the Input Message Manual, IM-4C000-01 or IM-4C002-01, respectively. The maintenance terminal also provides graphic displays of system status and alarms. The maintenance person selects input commands from the menu (list of commands) on each display page. The display screen of the maintenance terminal is split. The upper portion of the screen displays information necessary for the maintenance person to maintain the system while the lower portion displays a scroll of consecutive input and output messages.

Maintenance Printer

4.13 The maintenance printer is a Model 40 Data Terminal receive-only printer USOC 40P2F manufactured by Teletype Corporation (or equivalent). The printer is used to maintain a printed record of the communication to/from the maintenance terminal.

SOFTWARE

4.14 The following paragraphs provide general descriptions of the major software components of the craft interface subsystem.

A. Emergency Action Interface Firmware

4.15 The EAI provides a low-level maintenance access path from the maintenance terminal to the computer. The EAI software is contained in ROM. Thus, the software for the EAI cannot be destroyed due to power failure or accidental over-write. The primary purposes for the EAI are dead start and recovering from software insanity through manual intervention. The EAI performs the following functions:

- Executes EAI commands sent from the MTTYPC
- Handles recovery messages from the CC
- Performs self-initialization, if necessary
- Perform audits and detects EAI errors.

EAI Commands

4.16 The following paragraphs describe the commands that can be requested from the maintenance terminal via the EAI. These actions can be

performed regardless of the current state of the computer system.

4.17 ***Force Commands:*** The following commands are overrides that provide the basic emergency control over the associated CC:

- FONL (Force CC On-Line)—Forces the associated CC to the on-line state regardless of software attempts to switch (The other CC is forced off-line)
- FBDP (Force Primary Boot Device)—Forces usage of primary boot device (ie, disk) during subsequent system boots, regardless of software or firmware attempts to use secondary device (ie, magnetic tape)
- FBDS (Force Secondary Boot Device)—Forces usage of secondary boot device during subsequent system boots, regardless of software or firmware attempts to use primary device
- DTIM (Disable Sanity Timer)—Forces the sanity timer of the associated CC to be disabled; this inhibits subsequent switches.

4.18 ***Initialization Commands:*** The following functions relate to the initialization of the associated CC and the EAI:

- CLREAI (Clear EAI)—Resets the EAI to CC outputs and clears the EAI memory associated with these outputs
- EAIMRF (EAI Maintenance Reset Function)—Forces the associated CC to initialize (ie, MRF)
- INIP (Input Initialization Parameter)—Transfers 64 bits of the operating system initialization parameter from the MTTYPC to the EAI, where it is placed in EAI memory
- OUTIP (Output Initialization Parameter)—Transfers the contents of the EAI initialization parameter from the EAI to the MTTYPC.

4.19 ***Status Monitoring:*** The EAI provides a command that collects various information related to the status of the associated CC and the EAI itself. This command is:

(a) OUTSTAT (Output Status)—Returns a status message to the MTTYPC which contains

- RST—Indicates that the EAI has gone through a power-up restart
- RUN—Indicates that the associated CC is executing instructions from main store
- ACTIVE—Indicates that the associated CC is on-line
- ASW—Indicates that the EAI thinks that all seems well (ie, no internal faults detected)
- PRM—Indicates that a process recovery message has been received from the associated CC
- SPR—Indicates that a MRF has started
- PP22,PP23—Indicates the last active state of the pulse points from the CC
- IPB—Indicates that the EAI initialization parameter buffer contains nonzero data
- EAEN—Indicates that a force function to the associated CC is active or the initialization parameter is nonzero
- FBDP, FBDS, FONL, FOFL, DTIM, EAIMRF—Indicates the state of the corresponding EAI function node.

4.20 ***Processor Recovery Message:*** The following function allows the PRM to be accessed by the MTTYPC for display on the maintenance terminal:

- OUTPRM (Output Processor Recovery Message)—Transfers the 64-bit contents of the PRM buffer to the MTTYPC.

4.21 ***Recovery Message Handling:*** As a maintenance reset function (MRF) occurs in the associated CC, PRMs are generated by each step or process executed during the initialization. One or

more PRMs will be generated by the following steps or processes:

- Microboot
- Little boot
- Pinit
- Big boot
- DMERT kernel
- Error interrupt handler
- Disk driver
- IOP driver
- File manager
- Operating system processes
- Process manager
- SIM.

The PRM is composed of 16 hex digits that define the step or process sending the PRM, the initialization levels (operating system and application), the particular function within the step, and failure or progress information. Basically, there are two types of PRMs; All Seems Well (ASW) and Failure. The PRMs are sent to the associated EAI. The EAI forwards the message to the MTTYPC which in turn sends the PRMs to the maintenance terminal.

4.22 ***Self-Initialization:*** The EAI may need to perform a self-initialization. There are three initialization levels which are (in order of decreasing severity):

- Total Initialization—Executed during power-up or when both MTTYPC ports receive a BREAK; all RAM, registers, and flip-flops are cleared; the force outputs and UARTs are reset.
- Single-Break Initialization—Executed when a BREAK detected at only one port receives a BREAK; the UART and the area in RAM associated with the port are initialized.
- CLREAI Initialization—Executed when a CLREAI command is received; the force outputs and the area of RAM associated with these outputs are cleared.

4.23 ***EAI Audits:*** When the EAI is not executing MTTYPC commands or handling PRMs or diagnostic requests, the EAI performs audits and self-tests. The EAI will signal the MTTYPC when an error is detected during the self-tests or some of the audits. The audits performed are as follows:

(1) Output Audit—Compares the state of the outputs to the computer with an internal record of what the state should be; discrepancies will cause the EAI to signal the MTTYPC. The MTTYPC is responsible for taking the appropriate actions to recover the EAI or mark the EAI out of service. (This audit is designed to detect backplane shorts and EAI driver faults.)

(2) EAEN Audit—Checks the state of the force outputs and the initialization parameter; modulates the EAEN status bit and LED.

(3) Status Audit—Checks the state of the PONL signal and RUN logic; if a change occurs, the status word is updated and the MTTYPC is signaled.

(4) Data Link Audit—Checks the status of the ports to the MTTYPC; when an intercharacter time-out occurs or a port initialization exists, the UART of the port is reinitialized.

4.24 ***Error Detection:*** The EAI can detect the following types of operational errors:

- UART Initialization Fault—The EAI cannot initialize the UART and cannot exit the initialization loop. The MTTYPC recognizes this fault by the absence of an acknowledgement to a command.
- Command-Data Error—A parity error, framing error, or overrun error that occurs during transmission from the MTTYPC to the EAI. The EAI will return an error message to the MTTYPC.
- Command Syntax Error—A grammatically incorrect command from the MTTYPC; the

EAI will return an error message to the MTTYPC.

- Output Error—The EAI outputs do not agree with the expected states; the EAI will signal the MTTYPC via a control lead.

B. MTTYPC Handler

4.25 A portion of the IOP software is responsible for interfacing the MTTYPC to the operating system. This software performs the following functions:

- Restores the MTTYPC, when necessary
- Provides the interface to the special files
- Provides two separate paths for simultaneous input and output
- Provides direct memory access to and from application segments
- Provides terminal control access.

4.26 The application processes interface to the maintenance terminal are via three special files. These files are:

(1) /dev/tty—The terminal interface; used for ordinary input/output

(2) /dev/ack—Acknowledgment channel; used for acknowledgments to PDS messages

(3) /dev/cd—Control/Display interface; used for displays and control information.

C. Program Documentation Standards Shell (DMERT Generic 1 and 2)

4.27 The PDSHL supports the PDS maintenance terminal input language and MML language (DMERT generic 2) designed for ESS* switching equipment. The PDSHL is responsible for processing commands input from the maintenance terminals. Application programs are invoked by the PDSHL in response to a terminal command. The application programs are stored in files under several directories. Briefly, the actions of the PDSHL are as follows:

(1) Accept and parse commands from the terminals

(2) Perform directory search for application programs

(3) Create the application program and wait for its termination

(4) Determine if system is PDS or MML (DMERT generic 2).

In addition, the PDSHL will handle terminal signals that occur during any of these actions.

4.28 When the program is located, the PDSHL will attach the standard input of the application program to the terminal. This allows input from the terminal to the application program, if necessary. The PDSHL process uses conventions of the UNIX system to allow flexibility, simplicity, overall program modularity, and loose connectivity. The design of PDSHL process is based heavily on environment and support tools provided by the UNIX system. The PDSHL process, for example, provides an interface between the maintenance terminal, maintenance input request administrator (MIRA), and many other processes.

4.29 ***PDSGETTY:*** The PDSGETTY process provides the early initialization procedure for craft PDSHL terminals. This includes setting up standard input, standard output, and standard error file descriptors, initializing control display and spooler output capabilities, and executing the PDSHL.

4.30 The initialization procedure for a craft PDSHL terminal is the same as that used for a DMERT operating system terminal up to the point that the GETTY program is executed. The DMERT operating system allows for alternate GETTYs to be executed by the ***init*** process of the UNIX system by making appropriate entries in ECD on per-terminal basis. This allows maintenance terminals and terminals of the UNIX system to coexist on the same DMERT operating system. Once executed, the pdsgetty process needs specific information to determine the initial directory to be associated with a terminal, which shell to execute for that terminal, and

* Trademark of AT&T.

whether control display or spooler output capabilities are to be associated with that terminal. This information is contained in the ECD.

4.31 ***ECD Specifications for PDSGETTY:*** The ECD provides a GETTY record, defined as **struct getty_rec** in the header file (**lla/cft_rec.h**), to contain information needed by the PDSGETTY process. The PDSGETTY accesses the information in **getty_rec** through low level access (LLA) functions calls.

4.32 The pathname (.pname) file is in the initial directory that the PDSHL uses when attaching to a terminal. The .pname file is assumed by the shell to exist in the current directory. The PDSHL determines the current directory when first created by PDSGETTY. Thereafter, the PDSHL always uses the same directory upon request by client processes. Each line of the .pname file describes an environment for the PDSHL as follows:

> numeric label: PDS/MML search directory list:current directory:alternate shell $optional parameters

The PDSHL switches environments to parameters specified in the line when a client command for a line in the .pname file passes a return value to the PDSHL equal to the value in the numeric label field. Return values of zero from client command are ignored by PDSHL. Therefore, if environmental changes are desired, zero should not be used in the numeric label field. The first line in the .pname file is used by PDSHL as the default environment when the PDSHL is first created. A line in the .pname file may be continued with a backslash or terminated after any parameter with a new line.

4.33 The list of PDSHL search directories is searched by the shell. This list should be separated by blanks. The current directory parameter causes the PDSHL to execute a cd (change directory) using the current directory in the specified line. If on exit from a PDS/MML command a positive return code corresponds to an environment line with an alternate shell specified, the PDSHL will fork and execute the alternate shell in the specified directory. On exit from the alternate shell, the PDSHL will return to the previous environment.

4.34 The parameters following the dollar sign on the environment line are optional and are ignored by the PDSHL. An application may wish to add parameters to an environment by placing them after the dollar sign. A PDS/MML command created by PDSHL uses standard input, standard output, and standard error as set up by PDSGETTY.

4.35 ***Input Line Parser:*** The input to the PDSHL parser is the first line of the terminal command (ie, the command line). For the application command, the parser builds a list of the tokens that occur on the command line. For the PDSHL, the parser builds a list of character strings from the command line which is used to find the application program. The parser also provides syntactical and lexical error detection along with a minimal amount of error analysis. For example, the following command line is used to diagnose a CU:

> DGN:CU 0!

This line would be parsed into a list composed of a verb, delimiters, and identifications (ids), such as the following:

> "DGN" (verb)
>
> ":" (delimiter)
>
> "CU" (id 1[0,0])
>
> "0" (id 1[0,1])
>
> "!" (delimiter).

4.36 The PDSHL would use the first token of the command line (ie, the verb) in the form of a character string to locate the program responsible for executing this particular diagnostic. When found, the program is created and the PDSHL passes the list of five tokens to the program. The program would use this list to identify the specified operation to be performed. The parser for PDSHL is generated using the LEX and YACC programs. Semantic analysis is performed by the yyparse subroutine. The yylex subroutine is called by yyparse for lexical analysis.

4.37 The PDSHL implements a subset of standard PDS/MML syntax in the following two areas:

- Input delimiters
- Message diagnostics.

The ?A, ?I, ?D, and ?E are syntax error messages issued by PDSHL. The improper teletypewriter channel message (?C) feature is not supported by the PDSHL. The PDSHL input delimiters ("/" or "!") in PDS or ("?" or ";") in MML cause the terminal to be connected to the standard input of the application program. The connection permits the application program to read unparsed input from the terminal as needed.

4.38 ***Directory Search:*** The PDSHL has access to the file ".pname". This file contains a list of the pathnames of the directories that contain the application programs. The directory search algorithm used by the PDSHL is as follows:

(1) Checks the verb against the files in the first directory in the ".pname" file. If the verb matches the name of an executable file, the file is executed.

(2) If the verb matches a directory, checks the files within that directory against the first id. If the first id matches an executable file, the file is executed.

(3) If the first id matches a directory, then the files in that directory are matched against the second id. If an executable file is matched, the file is executed.

(4) If no match is found during the search in steps 1 through 3, then the procedure is repeated for the next directory listed in the ".pname" file.

(5) After all directories listed in the ".pname" file have been searched and an executable file has not been found, the PDSHL abandons the search and issues an error message.

4.39 ***Process Creation:*** The PDSHL invokes application programs by executing the "fork and exec" command of the UNIX system. The file name of the application program is supplied by the PDSHL as an argument to this command. Likewise, the list of tokens is also passed to the application program.

4.40 ***Signal Handling:*** When a signal is generated by the terminal or terminal handler, the signal is sent to all processes associated with the terminal. Each process then has one of the following options:

- Not catching the signal and terminating
- Catching and ignoring the signal
- Catching the signal and trapping to a subroutine.

The response by PDSHL to the signals varies from ignoring to dying upon receipt of the signal. Break and delete signals are caught by PDSHL. The hangup signals cause PDSHL to die forcing the terminal handler to bring up a new PDSHL for the terminal.

4.41 The terminal handler also originates the following time-out signals:

- Intercharacter time-out signal
- No acknowledgment time-out signal.

If the terminal handler waits more than the intercharacter time-out value (specified in the ECD) between characters, the intercharacter time-out signal is sent to PDSHL. A "?T" is sent to the terminal by PDSHL in response to an intercharacter time-out signal. The no acknowledgment signal depends on the acknowledgment time-out value in ECD. This signal is sent by the terminal handler to the PDSHL when the value in ECD is exceeded and no command acknowledgement is output to the standard error file descriptor. The PDSHL responds to a no acknowledgment signal by sending an "NA" to the terminal.

4.42 ***Program Documentation Standards Shell Library Functions (DMERT Generic 2):*** The PDSHL library functions have been combined into a craft library (libCFT). Basically, PDSHL reads a properly terminated input message and parses the input message into tokens. Then the appropriate client/application process with pointers to the parsed input message is invoked. Client and/or application processes have responsibilities (eg, acknowledgment time-out) to the PDSHL and IOP. The libCFT functions were developed for the following reasons:

- To aid client processes in meeting these responsibilities uniformly

- To provide client processes with tools for PDS/MML input message processing
- PDS/MML parsed input pointer development
- PDS/MML input data field/subfield data conversions
- Procedural functions to meet PDS/MML processing requirements within the 3B20D computer DMERT operating system environment
- Procedural functions to meet PDS/MML processing requirements within a remotely located SCC facility
- PDS/MML output message generation and destination routing.

Detailed information required by PDSHL client processes are defined in the standard header file for the PDS libraries (<cft/pdslib.h>).

4.43 Some of the functions described by libCFT are specifically designed for PDSHL clients and/or applications which handle multiline PDS input. The PDSHL is responsible for reading the initial portion of a multiline input command. The PDSHL client and/or application processes are responsible for reading subsequent lines of input using the functions in libCFT. In addition, there are function calls that allow user processes to interface directly with the craft interface output spooler. The libCFT is accessed by typing "sharelib:libCFT" in the specification (.b) file.

D. Craft Shell (UNIX RTR Release 1 Only)

4.44 The CFTSHL supports the PDS maintenance terminal input language and MML language designed for ESS switching equipment. The CFTSHL is also responsible for processing commands input from the maintenance terminals. Application programs are invoked by CFTSHL in response to a terminal command. The application programs are stored in files under several directories. Briefly, the actions of the CFTSHL are as follows:

(1) Determine if system is PDS or MML

(2) Accept and parse commands from the terminals using the input message catalog database

(3) Verify the existence and executability of client programs or perform a directory search for client programs

(4) Display acknowledgements to input messages as indicated in the input message catalog

(5) Create client programs and wait for the programs to terminate.

In addition, the CFTSHL will handle terminal signals that occur during any of these actions.

4.45 When the program is located, the CFTSHL will attach the standard input of the client program to the terminal. This allows input from the terminal to the client program, if necessary. The CFTSHL process uses conventions of the UNIX system to allow flexibility, simplicity, overall program modularity, and loose connectivity. The design of CFTSHL process is based heavily on environment and support tools provided by the UNIX system. The CFTSHL process, for example, provides an interface between the maintenance terminal, MIRA, and many other processes.

4.46 ***SHLGETTY:*** The SHLGETTY process provides the early initialization procedure for CFTSHL terminals. This includes setting up standard input, standard output, and standard error file descriptors, initializing control display and spooler output capabilities, and executing the CFTSHL.

4.47 The initialization procedure for a CFTSHL terminal is the same as that used for a UNIX RTR operating system terminal up to the point that the GETTY program is executed. The operating system allows for alternate GETTYs to be executed by the ***init*** process of the UNIX system by making appropriate entries in ECD on per-terminal basis. This allows maintenance terminals and terminals of the UNIX system to coexist on the same operating system. Once executed, the SHLGETTY process needs specific information to determine the initial directory to be associated with a terminal, which shell to execute for that terminal, and whether control display or spooler output capabilities are to be associated with that terminal. This information is contained in the ECD.

4.48 ***ECD Specifications for SHLGETTY:*** The ECD provides a GETTY record, defined as ***struct getty_rec*** in the header file (***lla/cft_rec.h***),

to contain information needed by the SHLGETTY process. The SHLGETTY accesses the information in ***getty_rec*** through LLA function calls.

4.49 The pathname (.pname) file is in the initial directory that the CFTSHL uses when attaching to a terminal. The .pname file is assumed by the shell to exist in the current directory. The CFTSHL determines the current directory when first created by SHLGETTY. Thereafter, the CFTSHL always uses the same directory upon request by client processes. Each line of the .pname file describes an environment for the CFTSHL as follows:

> numeric label: PDS/MML search directory list:current directory:alternate shell $optional parameters

The CFTSHL switches environments to parameters specified in the line when a client command for a line in the .pname file passes a return value to the CFTSHL that is equal to the value in the numeric label field. Return values of zero from client command are ignored by CFTSHL. Therefore, if environmental changes are desired, zero should not be used in the numeric label field. The first line in the .pname file is used by CFTSHL as the default environment when the shell is first created. A line in the .pname file may be continued with a backslash (\) or terminated after any parameter with a new line character (\n).

4.50 The list of CFTSHL search directories is used by the shell to locate/validate client process pathnames. This list should be separated by blanks. The current directory parameter causes the CFTSHL to execute a cd (change directory) using the current directory in the specified line. If on exit from a PDS/MML command a positive return code corresponds to an environment line with an alternate shell specified, the CFTSHL will fork and execute the alternate shell in the specified directory. On exit from the alternate shell, the CFTSHL will return to the previous environment.

4.51 The parameters following the dollar sign on the environment line are optional and are ignored by the CFTSHL. An application may wish to add parameters to an environment by placing them after the dollar sign. A PDS/MML command created by CFTSHL uses standard input, standard output, and standard error as set up by SHLGETTY.

4.52 ***Input Message Catalog Database:*** The CFTSHL uses the input message (IM) catalog database to validate the contents of input messages entered at a maintenance terminal. The IM catalog contains information about input message formats, keywords, valid argument values, and full pathnames of the client programs that execute the specified commands. The IM catalog also contains information which indicates whether the CFTSHL or a client program is to display the appropriate acknowledgement for each input message.

4.53 The CFTSHL operates in three different capability modes - full capability mode, partial capability mode, and limp mode. The full capability mode assumes that all UNIX RTR and application input messages are defined in the IM catalog database. The partial capability mode assumes that all UNIX RTR input messages are defined in the catalog but not all application input messages are defined in the catalog. If the IM catalog database cannot be accessed, the CFTSHL will operate in limp mode, which is a special case of the partial capability mode.

4.54 ***ECD Specifications for CFTSHL:*** The CFTSHL uses the following fields of the spooler information (splrinfo) record in the ECD:

- IM/OM syntax: This field can be set to PDS/MML. The CFTSHL and output spooler read this field to determine the proper I/O syntax.

- Full/Part capability mode: This field can be set to FULL or PART. The CFTSHL reads this field to determine the proper capability mode. Applicants who do not want to add their messages into the IM catalog must change the mode from FULL to PART using the recent change and verify system.

When it is initialized, the CFTSHL reports the capability mode and IM syntax it is using in the REPT CFTSHL TERMINAL IN SERVICE output message.

4.55 ***CFTSHL Help Facility:*** The Help facility of the CFTSHL provides assistance to users entering UNIX RTR PDS/MML input messages and can be used to complete or correct errors in input messages. If an input message is not defined in the IM catalog or if it contains invalid keywords or argument value(s), the CFTSHL will reject the input message with an appropriate error acknowledgement. If

the user does not understand or know what to do with the error acknowledgement, the user can get help from the CFTSHL by simply entering a question mark ('?'). The CFTSHL interprets the '?' at the end of a partial input message or following an error acknowledgement as a request for information about the format of the message. A second '?' is interpreted as a request to enter prompting mode. In the prompting mode, the shell will prompt the user for each keyword and argument in the message and provide information about the values that can be entered. When a complete input message has been constructed, the user may append to it, execute it, or cancel it. The help session is then completed; that is, help is provided for only one input message at a time.

4.56 ***Input Line Parser:*** The input to the CFTSHL parser is the first line of the terminal command (ie, the command line). For the client program, the parser builds a list of the tokens that occur on the command line. For the CFTSHL, the parser builds a list of character strings from the command line which may be used to find the client program. The parser also provides syntactical and lexical error detection along with a minimal amount of error analysis. For example, the following command line is used to diagnose a CU:

DGN:CU0!

This line would be parsed into a list composed of a verb, delimiters, and identifications (ids), such as the following:

"DGN" (verb)

":" (delimiter)

"CU" (id 1[0,0])

"0" (id 1[0,1])

"!" (delimiter).

4.57 If the command is not found in the IM catalog and the CFTSHL is operating in partial or limp capability mode, the CFTSHL would use the first token of the command line (ie, the verb) in the form of a character string to locate the program responsible for executing this particular diagnostic. When found, the program is created and CFTSHL passes the list of five tokens to the program. The program would use this list to identify the specified operation to be performed. The parser for CFTSHL is generated using the LEX and YACC programs. Semantic analysis is performed by the yyparse subroutine. The yylex subroutine is called by yyparse for lexical analysis.

4.58 The CFTSHL implements a subset of standard PDS/MML syntax in the following two areas:

- Input delimiters
- Message diagnostics.

The ?A, ?I, ?D, and ?E are syntax error messages issued by CFTSHL. The improper teletypewriter channel message (?C) feature is not supported by the CFTSHL. The CFTSHL makes no distinction between input delimiters ("/" or "!") in PDS or ("!" with craft consistency feature) or ";") in MML. For each delimiter, the terminal is connected to the standard input of the application program until a terminal output unlocking, acknowledgment occurs. The connection permits the application program to read unparsed input from the terminal as needed.

4.59 ***Directory Search:*** The CFTSHL has access to the file ".pname". This file contains a list of the pathnames of the directories that contain the client programs. If the input message is found in the input/output catalog, then the search directory is compared against the list of ".pname" directories. If the input/output catalog directory matches any ".pname" directory, then the CFTSHL verifies the existence and executability of the specified file. If the input/output catalog directory is not listed in the ".pname" file or if the process file specified in the input/output catalog does not exist or is not executable, then the input message is rejected. If the input message is not found in the input/output catalog and the CFTSHL is operating in partial capability mode or in limp mode, then the CFTSHL performs a directory search for the client program. The directory search algorithm used by the CFTSHL is as follows:

(1) Checks the verb against the files in the first directory in the ".pname" file. If the verb matches the name of an executable file, the file is executed.

(2) If the verb matches a directory, checks the files within that directory against the first id. If the first id matches an executable file, the file is executed.

(3) If the first id matches a directory, the files in that directory are matched against the second id. If an executable file is matched, the file is executed.

(4) If no match is found during the search in steps 1 through 3, the procedure is repeated for the next directory listed in the ".pname" file.

(5) After all directories listed in the ".pname" file have been searched and an executable file has not been found, the CFTSHL abandons the search and issues an error message.

4.60 ***Process Creation:*** The CFTSHL invokes application programs by executing the fork and exec commands of the UNIX system. The file name of the application program is supplied by the CFTSHL as an argument to this command. Likewise, the list of tokens is also passed to the application program.

4.61 ***Signal Handling:*** When a signal is generated by the terminal or terminal handler, all processes associated with the terminal are sent the signal. Each process then has one of the following options:

- Not catching the signal and terminating
- Catching and ignoring the signal
- Catching the signal and trapping to a subroutine.

The response by CFTSHL to the signals varies from ignoring to dying upon receipt of the signal. Break and delete signals are caught by CFTSHL. The abandon signals cause CFTSHL to abort the read command. The hangup signals cause CFTSHL to die forcing the terminal handler to bring up a new CFTSHL for the terminal.

4.62 The terminal handler also originates the following time-out signals:

- Intercharacter time-out signal
- No acknowledgment time-out signal.

If the terminal handler waits more than the intercharacter time-out value (specified in the ECD) between characters, the intercharacter time-out signal is sent to CFTSHL. A "?T" is sent to the terminal by CFTSHL in response to an intercharacter time-out signal. The no acknowledgment signal depends on the acknowledgment time-out value in ECD. This signal is sent by the terminal handler when the value in ECD is exceeded and no command acknowlegement is output to the standard error file descriptor. The CFTSHL responds to a no acknowledgment signal by sending an "NA" to the terminal.

4.63 ***Craft Shell Library Functions:*** The CFTSHL library functions have been combined into craft libraries (libCFT and libPARS). The libCFT contains functions and routines for clients and/or applications involved with maintenance input messages, handles multiline PDS/MML input and the generation of PDS/MML output messages. Basically, CFTSHL reads a properly terminated input message and parses the input message into tokens. Then the appropriate client/application process with pointers to the parsed input message is invoked. Client and/or application processes have responsibilities (eg, acknowledgment time-out) to the CFTSHL and input/output processor. The libCFT and libPARS functions were developed for the following reasons:

- To aid client processes in meeting these responsibilities uniformly
- To provide client processes with tools for PDS/MML input message processing.
- PDS/MML input message parsing
- PDS/MML input data field/subfield data conversion
- PDS/MML terminator syntax checking and uppercase conversions
- PDS/MML system installation determination
- PDS/MML language translation
- PDS/MML input message acknowledgement generation
- Procedural functions to meet PDS/MML processing requirements within a remotely located SCC facility

- PDS/MML output message generation and destination routing

- Attachment to the input message catalog database.

Detailed information required by CFTSHL client processes are defined in the standard header file for the PDS/MML libraries (cft/pdslib.h).

4.64 Some of the functions described by library libCFT are specifically designed for CFTSHL clients and/or applications which handle multiline PDS/MML input. The CFTSHL is responsible for reading the initial portion of a multiline input command. The CFTSHL client and/or application processes are responsible for reading subsequent lines of input using the functions in library libCFT. In addition, there are function calls that allow user processes to interface directly with the craft interface output spooler. The libCFT is accessed by typing "sharelib:libCFT" in the specification (.b) file. The libPARS is accessed by typing "−lPARS in the input section of the specification file.

E. Output Spooler (DMERT Generic 1)

4.65 The output spooler regulates the flow of output messages to maintenance terminals, maintenance printer, log files, data links, and other output devices. All output devices (regardless of type) are referred to as output files. The output spooler maintains a work queue for each output file. This work queue contains entries indicating the messages to be sent to the associated output file. The messages may or may not be sent based on the assigned priorities of the messages on the queue. Briefly, the functions of the output spooler are:

- Output message routing—Sends output messages to one or more output files.

- Output message flow control—Regulates the flow of messages to a specific output file to prevent the intermingling of the lines of several output messages sent at the same time to the same output file.

- Logging—Records the occurrences of different types of messages.

- Time stamping—Appends to the message, the time it was received by the output spooler; the time is specified in minutes past the hour (basic stamp) or full time and date (full stamp).

4.66 ***Message Classes:*** A certain type of message may need to be sent to more than one output file. For example, output messages pertaining to hardware removed from service may need to be sent to two particular terminals and one particular log file. The output spooler provides a mechanism that defines the output files and allows this association of a class of messages to an arbitrary set of output files. These associations are recorded in the map file.

4.67 ***Map File:*** The map file contains information defining device parameters, file parameters, and message classes. The pathname for this file is:

ft/spl/map file.p

The map file can be edited by the maintenance person to alter message classes or output file definitions, if necessary. The map file contains two types of entries:

(1) Output file definition entry—Every output file is defined by an entry in the map file. The output file definition entry contains

- A unique tag that represents the pathname of the output file

- Type of output file

- Whether or not output device is temporary

- Whether or not the messages sent to the output file should be sorted and output based on the priority of the messages

- Limits size of the internal queue associated with output file

- Specifies the forced line length of the output file

- Full pathname to the output file.

(2) Class definition entry—Message class to output file mapping is defined by class definition entries. This entry contains

- A class identifier (0 to 255); class 0 is reserved for the output spooler error log

- Whether alarms are to be handled by output spooler or ignored

- A list of the names of output files associated with this class of messages.

User Process Interface

4.68 User processes interface to the output spooler via a UNIX pipe. The output sent to the spooler consists of a user control string, the text of the message, and the end-of-transmission character.

4.69 The user control string (referred to as the spooler string) consists of ASCII characters. The handling of the text following the spooler string is based on the information within the spooler string. The important parameters of the spooler string are:

- A name identifying the output device or file pathname as defined in the map file (may be omitted if a class is specified)

- A class identifier as defined in the mapfile

- A priority-of-action indicator which is used to control audible alarms

- A time stamp option which is basic stamp, full stamp, or no stamp

- The priority of the output message, 0 (low) to 7 (high)

- An ident which is used to link an item in a log file with an entry in another file

- Type of entry on a log file

- The full pathname to an optional text file; text in this file is appended to the message

- Process identifier of the sending process.

4.70 The text of the message consists of an ASCII character string. The output spooler does not modify this field at any time. It is sent to the output file exactly as is. Following the text of the message is the end-of-transmission character which signifies the end of the message.

Output Spooler Structure

4.71 The output spooler (Fig. 3) is composed of four processes. These processes are:

- Spooler Input Processor (SIP)—Accepts spooler input messages and places them on the appropriate output file message queue

- Coordinator of the Spooler Output Processor (CSOP)—Keeps track of work queues, assigns work to the output processes, and notifies the alarm control process of alarms

- Spooler Output Processor (SOP)—Moves messages from the temporary storage message files to the output files

- Operating Queue Audit (OPAUD)—Routinely removes text files from the temporary storage message files.

4.72 ***Spooler Input Processor:*** The SIP accepts input messages from the PDSHL and application processes. The SIP performs the following actions for each message received:

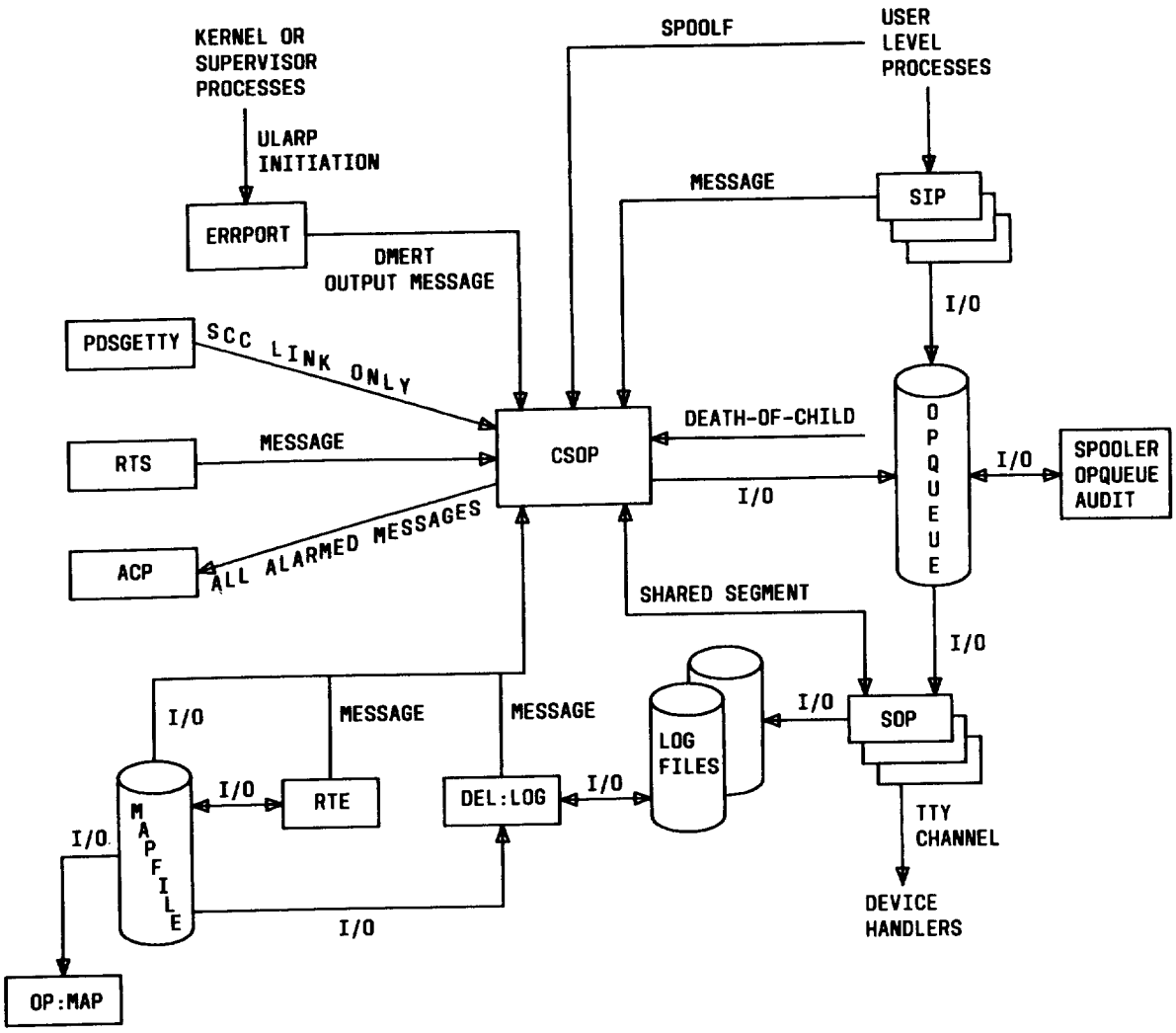
(1) Appends the priority-of-action field and time stamp to the beginning of the message

(2) Writes the message into an output message file for temporary storage. This file is located in one of eight directories depending on the priority of the message. (The name of the output message file used for temporary storage is determined by the SIP.) There is a directory for each message priority.

(3) Sends the CSOP a message containing the path name of the output message file and the class of the output message.

4.73 ***Coordinator of the Spooler Output Processor:*** The CSOP performs job assignment and overall coordination of SOPs. The CSOP maintains a master work queue that contains entries for every job sent from the SIP or by ***spoolf***. The actions taken by the CSOP in response to a message from the SIP are:

(1) Place the message from the SIP or ***spoolf*** in the master work queue.



LEGEND

ACP - ALARM CONTROL PROCESS
CSOP - COORDINATOR OF THE SPOOLER OUTPUT PROCESS
DEL:LOG - DELETE LOGFILE
ERRPORT - ERROR REPORT
OP:MAP - OUTPUT MAPFILE
OPQUEUE - OPERATING QUEUE
PDSGETTY - PROGRAM DOCUMENTATION STANDARDS GETTY
RTS - REAL TIME STATUS
SCC - SWITCHING CONTROL CENTER
SIP - SPOOLER INPUT PROCESS
SOP - SPOOLER OUTPUT PROCESS
ULARP - USER LEVEL AUTOMATIC RECOVERY PROCESS

Fig. 3—Spooler Interface and Utilities (DMERT Generic 1)

(2) Assign the job to the appropriate output file work queues based on the message class information defined in the mapfile. Depending on the priority option specified in the mapfile, the job is placed in the appropriate output file work queues based on:

- Time of message arrival (first-in/first-out basis)
- Priority of message, then by time of message arrival.

(3) Create SOPs for the output files that do not currently have one.

(4) Assign the first job on an output file work queue to the appropriate SOP by passing a message to the SOP; the message contains the path name of the output message file.

(5) Upon receiving an acknowledgment message from an SOP, the CSOP removes the appropriate job entry from the appropriate output file work queue.

(6) Send a message to an SOP assigning the next job.

(7) Determine if the last message sent completes the job. If the message has been sent to all output files in the message class (ie, none of the output file work queues associated with the message class contain an entry for the message), then the entry for the message is removed from the master work queue and the output message file is then removed from the appropriate priority directory. If the text of the message was obtained from a USER-FILE, an attempt is made to remove this file also.

(8) Repeat actions 5 through 7 until all messages are output.

4.74 ***Spooler Output Process:*** In general, there will be one SOP in existence for each output file. The SOP writes the message file specified by the CSOP to its output file. If the output file is not a log file, the SOP will remove the spooler string from the message before sending it. When the job is completed, the SOP will send an acknowledgement to the CSOP.

4.75 ***Operating Queue:*** The pathname of OPQUEUE (operating queue) is /cft/spl/opqueue. The OPQUEUE directory is used internally by the spooler for temporary storage of text for spooling. Old test files are routinely removed from OPQUEUE by an OPQUAD (OPQUEUE audit). The OPQUAD pathname is /cft/spl/ opquad. OPQUAD is initiated by the CRON (clock daemon) every 12 hours to read through the spoolers OPQUEUE. All files over 6 hours old are discarded via UNLINK.

F. Output Spooler (DMERT Generic 2 and UNIX RTR Release 1)

4.76 The output spooler regulates the flow of output messages to maintenance terminals, maintenance printers, log files, data links, and other output devices. All output devices (regardless of type) are referred to as output files. The output spooler maintains a work queue for each output file. This work queue contains entries indicating the messages to be sent to the associated output file. The messages may or may not be sent based on the assigned priorities of the messages on the queue. Briefly, the functions of the output spooler are as follows:

- Output message routing—Sends output messages to one or more output files.
- Output message flow control—Regulates the flow of messages to a specific output file to prevent the intermingling of the lines of several output messages sent at the same time to the same output file.
- Logging—Records the occurrences of different types of messages.
- Alarm control UNIX RTR Release 1 only—Provides an optional mechanism for the control of major, minor, and critical alarms.
- Time stamping—Appends to the message the time it was received by the output spooler; the time is specified in minutes past the hour (basic stamp) or full time and date (full stamp).

Output Class Definition

4.77 The spooler uses the ECD for assigning output messages to terminals. The mapping may be changed by recent change/verify, but a fallback de-

fault mapping will always be retained. An output message to the spooler may have no output class if outputted on a terminal in response to an input command. Both output message class and terminal on which the input command occurred or only one may be specified to the spooler. The ECD defines the device and file parameters and classes for the spooler. The output device and class mapping are defined in the ECD. The ECD must be created before CSOP can be executed. For each output file specified in the ECD, CSOP creates a SOP process. The pathname of SOP is /cft/spl/sop. Changes to the ECD may be made via recent change/verify.

4.78 ***Class Definition Records:*** The class definition (classdef) records are a set of records defining the 256 possible classes. Not all classes need be defined. The classdef record contains the following fields.

(a) ***Class Definition:*** A number between 0 and 255 specifying class definition.

(b) ***Alarm Flag:*** The alarm flag specifies if alarms associated with output messages directed to the class are forwarded to the SC/SD and alarm control process (ACP) for DMERT generic 2 or DAP for UNIX RTR Release 1 alarm functions.

(c) ***Logical Name List:*** A list of logical device names. The device record supplies definitions for each logical device.

4.79 ***Device Definition Records:*** The device definition (device) records are spooler device specification which defines a spooler output device. The device record contains the following fields:

(a) ***Logical Name:*** A one to eight character name. Legal references to an output device in the message control string are the logical name or a class containing the logical name.

(b) ***Device Name:*** Except for log files, this is the path name that the UNIX system uses to reference the output device (ie, "/dev/rop0"). For log files, it is all but the last character of path name.

(c) ***Sequence Numbering:*** If sequence numbering is enable ("y"), output messages sent to this device contain a sequence number indicating the order that the spooler received the output message request.

(d) ***Log Files and Regular Files:*** A flag specifying whether the output device is to be treated as a log file for a regular file ("y" for log file, "n" for regular file). The entries sent to a log file contain header information used by OP:LOG and DEL:LOG maintenance commands to process log files. This flag also enables split files. When a log file grows beyond the log file overflow limit, it is closed and another log file is opened. The names of the two files are formed by appending an ASCII one (1) or zero (0) to the end of the device name. Regular files are any spooler output device other than log files.

(e) ***Temporary Spooler Output Processes:*** If an SOP encounters an output error and this flag specifies a "temporary" SOP, the SOP exits. Messages sent to this device are lost until the CSOP receives a SOP restart request for that device. The restart request comes from a getty for the output device. An SOP that is "permanent" does not die on an output error; processing continues normally except that the current output message is lost.

(f) ***Priority:*** This option causes the output queue for a device to be kept in priority order rather than chronological order.

(g) ***Force Time Stamp:*** If this field is set to msgon or msgoff, it overrides the requested time stamp of an output message for the specifed device. If the field is set to ***dontcare***, the message stamp is used. The option is overridden if the force time stamp option in the miscellaneous spooler information (splrinfo) record is set to msgon or msgoff.

(h) ***Enable Multiple Messages per Write:*** If this flag is set, an SOP blocks as many messages into an output buffer as possible before writing the buffer out to its device. This option only has an effect if the output queue for an SOP has more than one output message in it. An output buffer contains a partial message if the output message is larger than block size. Messages contained in the same output buffer are separated by the message trailer string. If forced line length is nonzero and the output file is not a log file, the multiple messages per write feature are affected.

Tab expansion and additional new line codes in the output message throw off the character counter. This impacts only those applications which cannot tolerate message spanning write ( ) buffers.

(i) ***Queue Size:*** This field specifies the maximum number of output messages that are queued in CSOP for this device.

(j) ***Block Size:*** The size of an output buffer used by the write( ) function to output to the device.

(k) ***Forced Line Length:*** If this number is non-zero, lines greater than the specified limit are broken with a new line character. Forced line length has no effect on log files.

(l) ***Log File Overflow Limit:*** If the output device is a log file, this specifies the overflow limit for a log file half, specified in number of bytes.

(m) ***Message Header String:*** This is a 0 to 15 character string that precedes every output message on the device. To conform to MML output standards, this string is a new line character since MML requires a blank line between output messages.

(n) ***Message Header String:*** This is a 0 to 15 character string that separates every output message contained in the same output buffer. Its purpose is to aid those applications who previously relied on there being one message per output buffer to parse spooler output.

4.80 ***Miscellaneous Spooler Information Record:*** A splrinfo record defines miscellaneous information of the spooler. The splrinfo record contains the following fields:

(a) ***IM/OM Syntax:*** This field can be set to "MML" or "PDS". The craft spooler and shell read this field to determine the proper input/output syntax.

(b) ***Force Time Stamp:*** If this field is set to msgon or msgoff, it overrides the requested time stamp of output messages for all devices. If the field is set to ***dontcare, the force time stamp specification for each device is used.***

User Process Interface

4.81 The output to the spooler consists of an ASCII string in the form of a user control "ucs" string followed by the text to be spooled. ASCII string is commonly referred to as the spooler string. The "ucs" must be preceded and followed by dollar signs ($).

4.82 The format of the spooler string is ucs text, where "ucs" consists of one to nine positional parameters separated by commas (,). Fields may be omitted, but delimiting commas must remain, except trailing commas. An output message sent to ***spool*** or ***spoolf*** may contain the following fields.

- A name identifying the output device or file pathname as defined in the ECD (may be omitted if a class is specified)
- A class identifier as defined in the ECD
- A priority-of-action indicator which is used to control audible alarms
- A time stamp option, which is basic stamp, full stamp, or no stamp
- The priority of the output message, 0(low) to 7(high)
- An ident which is used to link an item in a log file with an entry in another file
- Type of entry on a log file
- The full path name to an optional text file; text in this file is appended to the message
- Process identifier of the sending process
- Text consists of ASCII characters which will be passed to the output file.

4.83 The spooler (Fig. 4 for DMERT generic 2 or Fig. 5 for UNIX RTR Release 1) consists of the following executable modules:

- CSOP—Accepts spooler input messages and places them on the appropriate output file message queue. Also keeps track of work queues, assigns work to the output processes, and notifies the alarm control process of

alarms (in DMERT generic 2) or control alarm functions (in UNIX RTR Release 1).

- SOP—Moves messages from the temporary storage message files to the output files.

The spooler also consists of output configuration definitions in the ECD. During normal processing, the modules are initiated by one of the following primary stimuli:

- ***Spoolf***—Waiting for a message from the parent process
- CSOP—Waiting for input from ***spool/ spoolf*** library functions
- SOP—Waiting for a message from CSOP to output to its device.

4.84 The ***spoolf*** function accepts input messages from the PDSHL (DMERT generic 2) or CFTSHL (UNIX RTR Release 1) and application processes. The ***spoolf*** performs the following actions for each message received:

(1) Appends the priority-of-action field and time stamp to the beginning of the message.

(2) Writes the message into an output message file for temporary storage. This file is located in one of eight directories depending on the priority of the message. (The name of the output message file used for temporary storage is determined by the ***spoolf***.) There is a directory for each message priority.

(3) Sends the CSOP a message that contains the pathname of the output message file and the class of the output message.

4.85 The CSOP performs job assignment and overall coordination of SOPs. The CSOP maintains a master work queue that contains entries for every job sent from the ***spoolf***. The actions taken by the CSOP in response to a message are:

(1) Places the message from the ***spoolf*** in the master work queue.

(2) Assigns the job to the appropriate output file work queues based on the message class information defined in the ECD. Depending on the priority option specified in the message, the job is placed in the appropriate output file work queues based on the following:

(a) Time of message arrival (first-in first-out basis)

(b) Priority of message; then by time of message arrival.

(3) Creates SOPs for the output files that do not currently have one.

(4) Assigns the first job on an output file work queue to the appropriate SOP by passing a message to the SOP; the message contains the path name of the output message file.

(5) Upon receiving an acknowledgement message from an SOP, the CSOP removes the appropriate job entry from the appropriate output file work queue.

(6) Sends a message to an SOP assigning the next job

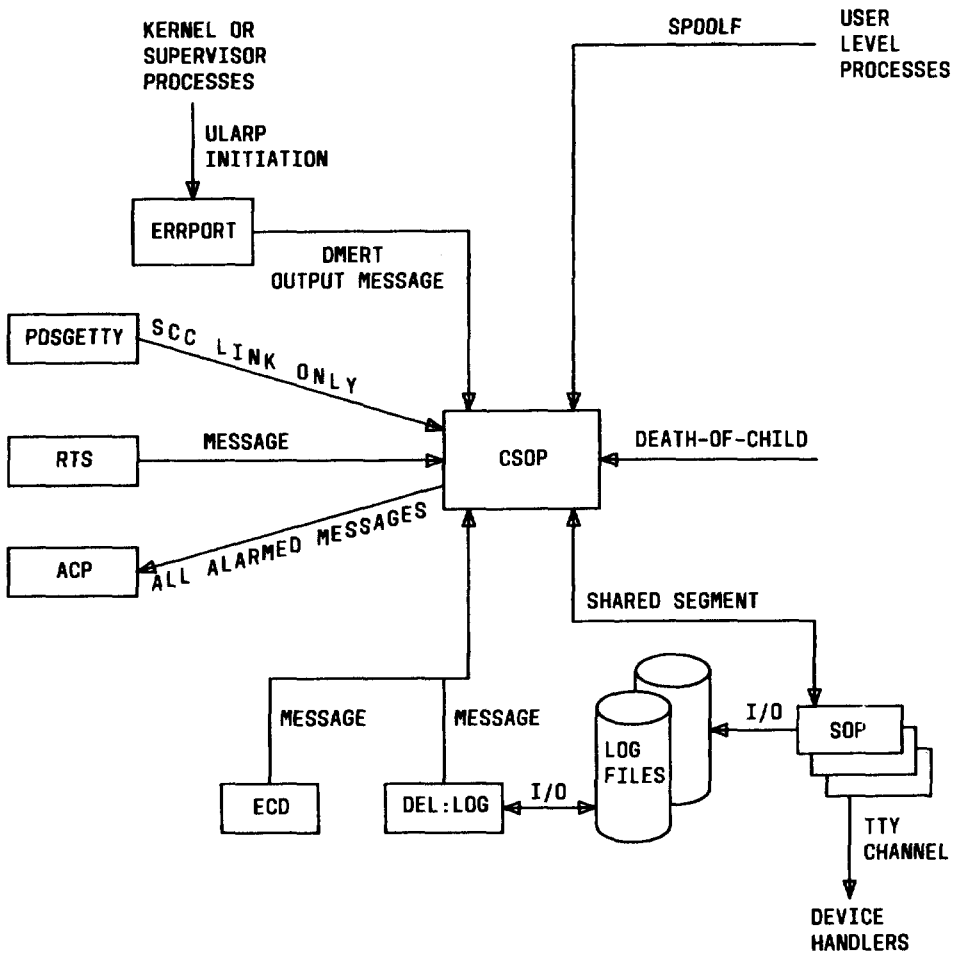
(7) Determines if the last message sent completes the job. If the message has been sent to all output files in the message class (ie, none of the output file work queues associated with the message class contain an entry for the message), then the entry for the message is removed from the master work queue and the output message file is removed from the appropriate priority directory. If the text of the message was obtained from a user-file, an attempt is made to remove this file also.

(8) Repeat Steps 5 through 7 until all messages are output.

4.86 In general, there will be one SOP in existence for each output file. The SOP writes the message file specified by the CSOP to its output file. If the output file is not a log file, the SOP will remove the spooler string from the message before sending it. When the job is completed, the SOP will send an acknowledgment to the CSOP. The library functions are the interfaces that exist to the output spooler.

G. Display Administration Process

4.87 The display administration process (DAP) provides a bidirectional graphic interface

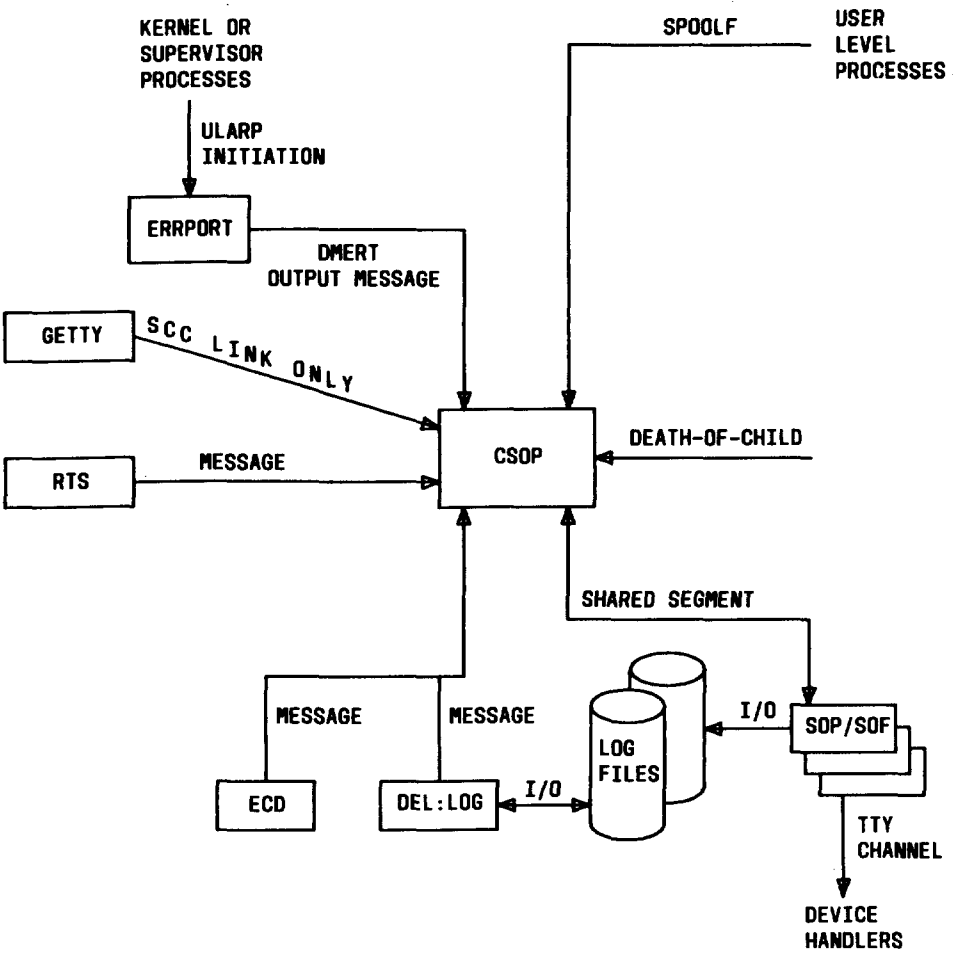


LEGEND

ACP - ALARM CONTROL PROCESS	POSGETTY - PROGRAM DOCUMENTATION STANDARDS GETTY
CSOP - COORDINATOR OF SPOOLER OUTPUT PROCESSES	RTS - REAL TIME STATUS
DEL:LOG - DELETE LOGFILE	SCC - SWITCHING CONTROL CENTER
ECD - EQUIPMENT CONFIGURATION DATABASE	SOP - SPOOLER OUTPUT PROCESS
ERRPORT - ERROR REPORT	ULARP - UNIX LEVEL AUTOMATIC RESTART PROCESS

Fig. 4—Spooler Interface and Utilities (DMERT Generic 2)

between operating system/application processes and the maintenance terminal. The primary function of DAP is to generate virtual panel images (referred to as pages) and display these images on the screen of the maintenance terminal. A page is a set of functionally grouped software-controlled indicators. Interactive modifications to these displays (ie, changing the state of an indicator, selecting an option from a menu, etc.) are referred to as " pokes". The DAP accepts information regarding pokes that occur and modifies the graphic display to coincide with the requested change. In addition, the DAP will initiate the request by notifying the appropriate process to carry out the request. The application processes can request changes on a page via messages to the DAP. Thus, the DAP provides a means of requesting



LEGEND

CSOP - COORDINATOR OF SPOOLER OUTPUT PROCESSES
DEL:LOG - DELETE LOGFILE
ECD - EQUIPMENT CONFIGURATION DATABASE
ERRPORT - ERROR REPORT
RTS - REAL TIME STATUS
SCC - SWITCHING CONTROL CENTER
GETTY - PDS/MML GETTY OR DIALOGUE GETTY
SOP - SPOOLER OUTPUT PROCESS
SOF - SPOOLER OUTPUT FUNCTION (PART OF THE DIALOGUE SHELL)
ULARP - UNIX LEVEL AUTOMATIC RESTART PROCESS

Fig. 5—Spooler Interface and Utilities (UNIX RTR Release 1)

information, changes to the system, diagnostics, etc., by a means other than the typical terminal message. The DAP can display up to eight pages on one maintenance terminal. Basically, DAP:

- Allows interactive graphics (panel type displays) that can be modified through cursor control or light pen
- Allows design of screen images (eight pages maximum) by the application
- Allows flexible page design
- Supports independent maintenance terminal communities (ie, one maintenance terminal for diagnostics, one for restorations, etc.).

4.88 Application processes can control a variety of attributes associated with graphic displays via the DAP interface. An application process can:

- Attach a maintenance terminal
- Remove a maintenance terminal
- Initialize a page
- Display a page on a specific maintenance terminal
- Erase a page displayed on a specific maintenance terminal
- Change the state, poke characteristics, or variable text of an indicator
- Request the state, poke characteristics, or variable text of an indicator.

4.89 The DAP interfaces between user processes running under the DMERT or UNIX RTR operating system. The page descriptor files (PDFs) are generated on an off-line support computer using page descriptor file generator and states processes/tools. The PDFs are data structures that instruct DAP on the placement of patterns on the maintenance terminal screen to form display pages. The PDFs also provide information which is based on the fixed or dynamic status of display variables to DAP concerning the handling of control inputs. The page descriptor language (PADL) is used to build source files containing one or more display page descrip ions. These source files are used to generate the PDFs. The PADL is a language that simplifies the construction of displays.

4.90 A page can contain up to 128 indicators. Each page has a state (on/off), a set of display attributes (shape, color, intensity, and legend), and two control attributes (acknowledgment and action). The variables associated with an indicator are state, text, and poke characteristics. When displaying the indicator on the maintenance terminal, these software variables are options. Indicators can be composed of one or more of the following basic shapes:

- Keys
- Rectangles
- Lines
- Text fields.

The placement of these shapes on the screen is fixed in PDF. However, two visual characteristics (color and text) can reference indicator variables. These characteristics may change when the referenced variable changes.

H. Real Time Status Report

4.91 The real-time status (RTS) process reports hardware status changes of the computer to the maintenance terminal via PDS/MML compatible terminal messages and DAP display pages. For DMERT generics 1 and 2, the RTS process places a time stamp message in the log files of the form:

REPT:TIME month/day/year, hours:minutes:seconds

This allows log files to be searched for entries that fall within specified times.

4.92 Effective with UNIX RTR Release 1 craft consistency feature, the time stamp message will be a separate entity. The time stamp message will be controlled by the clock daemon (CRON) mechanism time-of-day scheduler. This will allow each application to change its timing parameter or suppress it entirely. Due to restrictions on CRON, the time stamp message will not be executed when the system is operating in minimum configuration.

4.93 The RTS process consists primarily of interfaces to various entities of the DMERT system. These interfaces are:

- A status-reporting port interface from the hardware drivers for receiving status changes
- An equipment configuration database access interface for retrieving status information of a unit control block (UCB) via low level access system function calls
- A spooler interface for outputting unit status messages that conform to PDS/MML via ***spoolf*** library function

- A DAP interface for updating the common processor display page and accepting menu inputs from that page

- An application interface to allow applications to receive unit status changes.

4.94 Upon creation, the RTS process will access the ECD and the current read status of all units associated with the system display page from the UCB in the ECD. The DAP display page is initialized and spooler messages are sent via the spooler to indicate the current status of the common hardware units.

4.95 When an operating system driver reports a status change, the RTS process retrieves the current status of the associated hardware from the ECD. Then, the display page is updated (via the DAP) and a PDS/MML message is sent to the maintenance terminal (via the output spooler). Also, the status change is reported to the application via the application interface.

4.96 When a "poke" input is received from the DAP (ie, a menu item on the display page was selected), the process that owns the page forwards the request to MIRA. The application is also notified of the request via the application interface.

4.97 The RTS owns the common processor display page. This page is composed of a set of indicators. An RTS indicator has the following attributes:

(a) Unit name (such as CU, IOP, etc.)

(b) Major state, which can be:

- Active (ACT)

- Growth (GROW) DMERT generic 2 and UNIX RTR Release 1

- Initialization in progress (INIT)

- Out of service (OOS)

- Standby (STBY)

- Unavailable (UNAV)

- Unequipped (UNEQ)

- Unit off-line (OFL).

(c) Optional minor state, which can be:

- Automatic diagnostic (AUTO)

- By Pass (BYP)

- Fault (FALT)

- Error recognition inhibited (INH)

- Manual (MAN)

- Unit forced active (FRCL)

All of the major and minor states may not be applicable for a given hardware unit.

Power Switch Monitor

4.98 The power switch monitor is informed of changes in power switch states via the scanner/signal distributor (SC/SD) administrator. A group 1 power switch has three scan points and two distribute points which are organized into a logical SC/SD group. The power switch monitor uses a transition table to determine the appropriate action to be taken for each scan change. A scan point change results in one or more of the following actions:

- Generates a remove/restore request and sends the request to MIRA

- Operates the power switch lamps

- Sends a message to the output spooler.

5. CENTRAL OUTPUT MECHANISM FEATURE

5.01 The central output mechanism feature (NI.043) is a UNIX RTR Release 1 feature that provides the capability to put UNIX RTR output messages in a central database. This feature has the dual aim of centralizing output messages to facilitate converting their English text to other natural languages for applications with non-English-speaking customers and to allow the manipulation of the alarm level and message class associated with the output message. The Central Acknowledgement Mechanism (NI.043C), an add-on to the central output mechanism feature, provides the centralizing of input message acknowledgements.

5.02 These features are provided for the 5ESS Switch Export project for customers outside the United States who want output messages in languages other than English. Having the output messages in a central place makes it practical for them to provide this capability. The customers also require the flexibility to customize the message class to which an output message is routed, and what alarm, if any, should be associated with the message.

5.03 The major aspects of the central output mechanism feature are:

- Specifying all UNIX RTR spooler output messages in a central output message database (OMDB)
- Add format and spool capabilities using the OMDB to the existing spool-only capabilities in the UNIX RTR output message spooler
- New craft input commands to update and examine the message class and alarm level associated with each output message defined in the OMDB.

5.04 The major impacts of this feature are:

- New software tools and administrative procedures for building and maintaining the OMDB
- The need to update the OMDB whenever a UNIX RTR spooler output message is added, changed, or deleted
- Addition of a new field to the ***spirenfo*** ECD record and RC/V form
- Addition of a new field update type to the SG database and a new MSGS template for the OMDB
- Changes to several global header files, including ***spooler.h, fault.h, cft/pdslib.h, lla/sg.h, fmgr/segcodes.h,*** and ***libs.h***
- New format-and-spool interface functions added to libCFT and liberrpt
- Extensive changes to the output message spooler (CSOP)
- New craft input commands to manipulate the OMDB
- Updates to several UNIX RTR documents, including the ***Input and Output Message Manuals***, Volumes 1, 2, 3, and 4 of the ***UNIX RTR Programmer's Manual***, and the ***Craft Interface Users Guide***
- Conversion of all UNIX RTR spooler output message-generating software to use the new mechanism.

5.05 The major aspects and impacts of the Centralized Acknowledgement feature are:

- Specifying all UNIX RTR acnowledgement messages in a central acknowledgement database (ACKDB)
- New software tools and administrative procedures for building and maintaining the ACKDB
- The need to update the ACKDB whenever a UNIX RTR acknowledgement message is added, changed, or deleted
- Changes to the global header files fmgr/segcodes.h and ***libs.h***
- New acknowledgement functions added to libCFT
- Updates to several UNIX RTR documents
- Conversion of all UNIX RTR acknowledgement message-generating software to use the new mechanism.

A. Output Message Database

5.06 The OMDB is a binary file which contains all the information necessary to display output messages from the 3B20D computer system. When the CSOP is started up, it will read the disk copy of the OMDB into its address space. While reading the database in (and also when processing formatting requests), segments of the database will be selectively removed and added to CSOP. The CSOP accesses this information in order to format and spool

an output message to a device. Each message in the OMDB contains the following information:

- Message class
- Alarm level
- Handling priority
- Timestamp option
- Message, table header, and/or dump line text
- Variable translation and enumerations.

All the above fields are specified by the creator of the output message (See paragraph 5.16). All other fields in the database are created by the database builder (***3bobld***) and are used for administrative purposes.

OMDB Structure on Disk

5.07 The OMDB, as it resides on disk, is shown pictorially in Fig. 6. The disk OMDB consists of six sections. All sections except the OMDB header are dynamic in size during the building procedure. In the disk file, however, they have been concatenated to avoid wasted space.

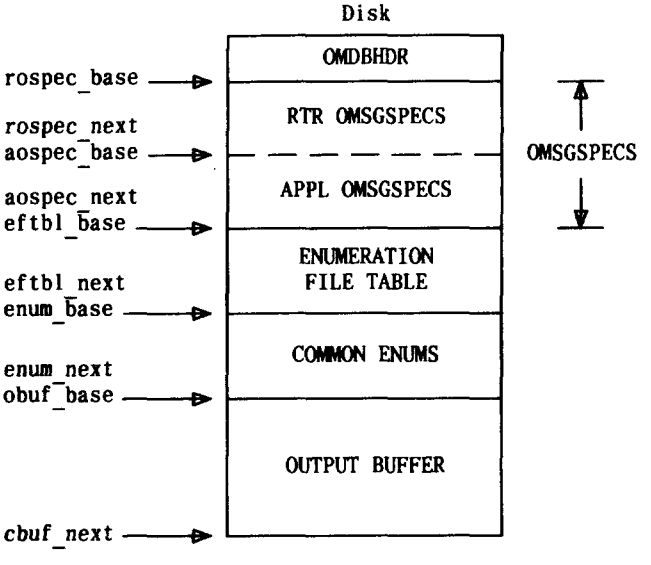


Fig. 6—OMDB as Created on Disk

5.08 ***OMDB Header:*** The first section is an administrative header which contains points to the beginning and end of each of the other database sections. This section of the database has a fixed size (size of OMDBHDR) and is always located at offset 0 in the disk database (see Note). The header also contains the first and last UNIX RTR and application keys. These fields are used to verify the validity of keys when accessing the OMDB.

Note: The OMDB header contains absolute addresses. All other addresses are actually relative offsets from one of these header points.

5.09 ***UNIX RTR Output Message Specification Records:*** The second section of the database is an array of output message specification records (OMSGSPEC structures). Each OMSGSPEC structure contains the message class, alarm level, handling priority, and timestamp option for a UNIX RTR message. Also in this structure is a pointer to the message text (found in the output buffer section), the size of the message text, and a count of the number of variables in this output message. Only UNIX RTR message specifications are contained in this section. The size of this section depends on the largest UNIX RTR key in use (up to FIRST_APPL_KEY-1). Any unused keys less than the largest used key will cause gaps.

5.10 ***Application Output Message Specification Records:*** This section contains application output message specification records. These records have the same structure as the UNIX RTR specification records. UNIX RTR and application output message specifications are kept separate in order to minimize the administrative problems of database key allocation. In the disk file, the smallest allowed application key (FIRST_APPL_KEY) follows immediately after the largest used UNIX RTR key. The largest used application key determines the size of this section. Unused keys will cause gaps.

5.11 ***Enumeration File Table:*** This section is organized as an array of enumeration file table (EFTBL) structures. Each EFTBL entry contains a common enumeration filename and a pointer to the enumeration data within the common enumeration section. Private enumerations do not have a filename associated with them, so they do not reside in this section.

5.12 ***Common Enumerations:*** Common enumerations are enumerations which are not unique to any one individual output message. The values and strings of these enumerations are contained in this section. Only one copy of each of the enumerations is maintained in this section, so all output messages requiring one of these enumerations share the same copy of the enumeration data. This section is organized as consecutive enumeration data elements, where an enumeration data element is an array of ENUM structures followed by all the enumeration strings for an enumeration variable.

5.13 ***Output Buffer:*** This section contains the text for an output message, translation information for each formatted variable, and local enumeration data. The layout of the output buffer section is shown in Fig. 7. Every message in the database has an output message specification (OMSGSPEC) entry, located in either the UNIX RTR or application output message specification section. Each OMSGSPEC entry contains a pointer in the output buffer section to the ASCII text for a message. The VAR entries for the message begin immediately after the message text and can be easily found by using the ***obuf_sz*** field from the OMSGSPEC.

5.14 The VAR entries are organized as an array of VAR structures. Each structure contains the following fields:

- ***translation*** — The type of translation to be performed on this variable
- ***data_in_bytes*** — A Boolean bit field indicating whether the ***field_sz*** and ***field_off*** fields are in bytes or bits. This field is provided to increase the performance of output message formatting; formatting bytes and words is faster than formatting bit fields
- ***c_enum*** —A Boolean field indicating whether an enumeration is a common (value = 1) or a private (value = 0) enumeration. When formatting a message, this field actually indicates whether the enumeration data can be found in the common enumeration section or in the output buffer section.
- ***field_sz, field_off*** — These two fields indicate where in the user's output message request the data for the variable is located. The ***field_off*** indicates the offset into the user's

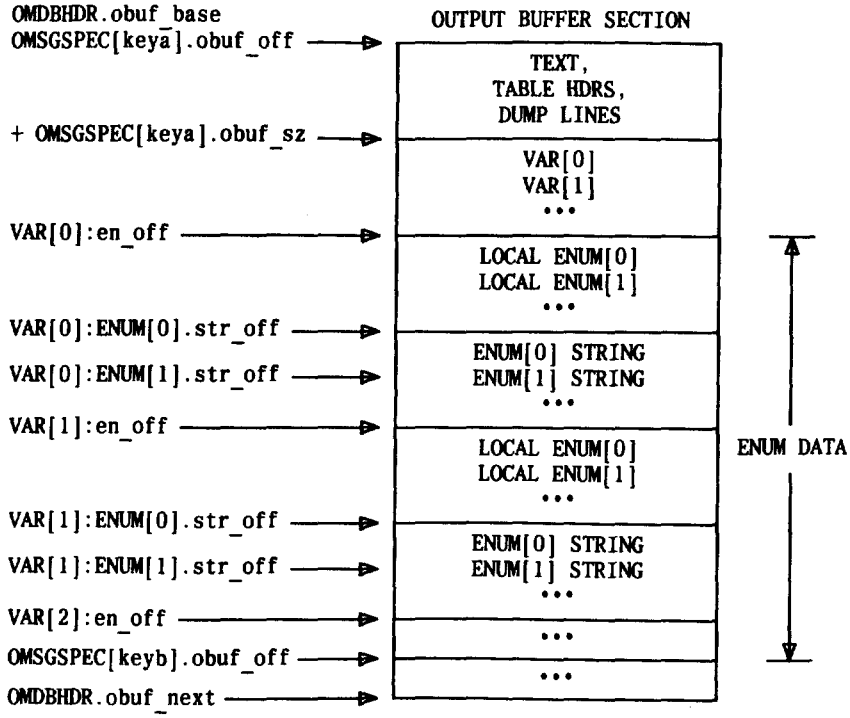


Fig. 7—Layout of Output Buffer Section

data and ***field_sz*** indicates the size of the data field. These two fields are in bytes or bits depending on the value of the ***data_in_bytes*** field.

- ***np_value*** — A nonprinting value for a variable. This is specified by the user in a .ofmt file.

- ***elmnt_sz*** — Dump element size is the size (in bytes) of the structure which will be dumped as part of a dumpline. This field is used as an increment to find the next occurrence of a dump element.

- ***en_off*** — Offset to the enumeration structures. If the enumeration is a common enumeration (***com_enum*** = ***1***), the offset is relative to the enum_base field, otherwise it is relative to the obuf_base field.

- ***prnt_fld_sz*** — Size of the data field, when printed. The ***prnt_fld_sz*** specifies the minimum number of characters that will be printed. If the printed data value requires more than ***prnt_fld_sz*** characters, ***prnt_fld_sz*** is effectively ignored.

- ***appl_fld*** — A reference point for applications to put their own special fields in the VAR structure.

5.15 Immediately following the VAR entries are the local enumerations. The ***enum_off*** field of the VAR structure points to an array of ENUM structures. The text string for an enumeration is pointed to by the ***str_off*** field in the ENUM structure. The strings for local enumerations are placed after the last ENUM structure for a variable.

B. OMDB Structure in Memory

5.16 The incore OMDB is shown pictorially in Fig. 8. The common enumerations are stored in an unnamed segment; and the header and output message specifications are placed in a named segment. This segment is named so that the ***gomdb( )*** library function can gain access to the message class and alarm level data stored in this segment. The common enumerations are contained in an unnamed segment, because CSOP is the only process which requires access to the common enumeration data. Since these enumerations are used by many messages, the enumerations are kept in their own segment, separate from the output buffer segments, so that this segment can always be resident incore. [Segments are variable in size from 1 to 64 pages (2048 bytes to 128K bytes)]. The output buffer segment(s) are unnamed segments. Since only one process, CSOP, needs access to the actual message text, naming the segments is not required. The output buffer segments will be created at adjoining virtual addresses in CSOP's address space. As shown in Fig. 8, the header points to the beginning and end of each section.

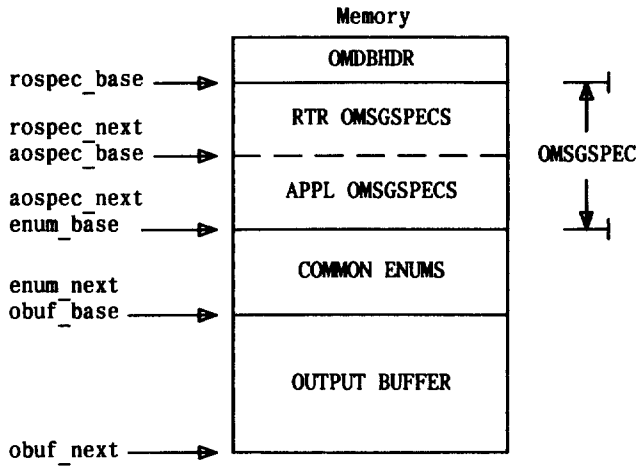


Fig. 8—Incore OMDB

5.17 The incore copy of the OMDB is organized similar to the disk OMDB. The differences between the two databases are:

- The incore OMDB contains different pointers in the OMDBHDR. These pointers contain virtual addresses which specify the beginning and end of each of the other incore OMDB sections.

- The UNIX RTR and application OMSGSPEC records are concatenated together and treated as one section.

- The enumeration file table (EFTBL) section is only used when building the database. Since it is not needed when accessing the database, this section is not part of the incore OMDB.

- The output buffer section is aligned on a segment boundary, so that segments of the database can be added and removed easily.

C. OMDB Generation Tools

Output Message Definition Files

5.18 Special purpose files called output message definition files contain a description of the presentation of the report at the craft interface and the translation rules needed to transform the data needed by the formatter into its external form. A description of any data needed by the formatter to complete a report must be provided as standard "C" data definitions in header files, which are included in the message definition files.

5.19 ***.ofmt Files:*** The introduction of the Central Output Mechanism necessitated the introduction of new ***.ofmt*** files for the UNIX RTR. Some changes were made to the existing ***.oty*** files of the 5ESS. The output message definition files — ***.ofmt*** — to be used within the UNIX RTR have the layout shown in Fig. 9. All lowercase bold names indicate reserved keywords. All uppercase names indicate develop-supplied data. Entries enclosed in [] are optional. Entries of the form (x ¦ y) indicate that either x or y must be chosen but not both.

Intermediate Build Products — *.om* Files

5.20 The process of building an OMDB from message definition files is a 2-stage one. First, the ***.oty*** or ***.ofmt*** files are built into intermediate products called ***.om*** files. The ***.om*** files are then built into the database itself. A ***.om*** file is an ASCII file

```
[include <      >]
[include "      "]

# The opening brace may be placed either on the same line as
# "output" or on the following line (in the first character
# position)

output ( { |
  } )
       key      KEYVAL
       time     TSTAMP
       class    MSG_CLASS
       alarm    PRIORACT
       outprior HNDLPRIO

# Choose either a reference to a common definition
#       or the complete definition itself
     ( common  COMMON_KEY  |
# At least one of the following three lines must be present
       if the complete definition is given
     [ prototype MSG_TEXT ]
     [ tblhdr    HDR_TEXT ]
     [ dumpline DUMP_TEXT ]
     [ field  ITM_NAME TRANSTYP [ VARFLD ] [ SIZE_OR_NP ] ]

                 • (number of variable fields)
                 •
           [ ( enum ENMID ENMSTR |
include <ENUM_FILE> ) ] )
}
[ output ( { |
  } )
   •
   •
   •
} ]
```

Fig. 9—Layout of .ofmt File

which can contain many identically formatted entries, each entry corresponding to one ***output*** { } structure from the ***.ofmt*** or ***.oty*** file. The layout of a single entry is as follows:

key class timestamp alarm priority
(common key |
text - line(s)
item - count
item - spec(s))

The ***.ofmt*** File Parsing Tool (***ofmtparse***)

5.21 The ***.ofmt*** file processing tool, ***ofmtparse***, performs the first step towards building an OMDB. It accepts an input of one or more ***.ofmt*** files (in the current directory) and creates as output an equivalent number of ***.om*** files, also in the current directory. The tool can be invoked via UNIX-level command on any UNIX RTR development system, as well as via the official build procedure. The UNIX-level command is:

ofmtparse [−d] <.ofmt file> [<.ofmt fle> ...]

The generated ***.om*** files have the same names as the ***.ofmt*** files, except that the ".ofmt" is replaced by "om". The −d option requests the tool to preserve intermediate human-readable files for debugging purposes.

5.22 The tool first checks the correctness of the command line. There must be at least one ***ofmt*** file, and the only valid option is −d. Each other file must be in the current directory, its name must end with the characters "ofmt", and not exceed 14 characters in length. The tool also checks that the user has the environment variable "OFC" defined. It must point to the official node to allow access to various header files during the compilation phase. For each input file, the tool goes through four main phases:

(1) Processing of include files

(2) Parsing of data

(3) Compilation of generated .c files

(4) Execution of object program.

5.23 The main goal of the first two phases is to generate a "C" program, consisting of a main routine, a supporting header file, and a number of functions, each of which deals with just one output structure from the ***ofmt*** file.

5.24 The compilation phase first gathers all of the generated .c files. There is one main file, named after the input ***ofmt*** file, which contains the main() function, plus one file for each key found in the input file. A compiler command string is constructed and then offered to the ***system()*** function for execution.

5.25 The final phase of the program simply calls the ***system()*** function with the "a.out" file generated during the compilation phase. The output of this is a ***.om*** file corresponding to the original ***.ofmt*** file.

5.26 All error messages generated by the program are sent to ***stderr***. In general, the occurrence of an error will not cause the program to abort. Where possible, the program will continue processing until the end of the parsing phase. No attempt is made to compile and execute the generated program if errors were detected.

5.27 One warning message (rather than an error message) is generated if the user's alarm level and handling priority appear to be incompatible. This, in isolation, will not prevent the generation of an ***.om*** file.

Output Message Database Builder (*3bobld*)

5.28 The final stage in the process of OMDB generation is the building of the OMDB itself from ***.om*** files. The building tool can accept one or more ***.om*** files as input and will generate one binary database file, called ***omdb***, as output. The tool can be invoked via a UNIX-level command on any UNIX RTR development system, as well as via the official build procedure. The UNIX-level command is:

3bobld [−uxs] <.om file> <.om file>

5.29 The −u option allows the user to indicate that this build will include updated .om entries (i.e. duplicates of existing database entries), which should be accepted by the builder without causing the builder to abort, and which are intended to replace the current ones. This option is intended to be used by developers to test a changed output message without having to rebuild the entire OMDB.

5.30 The −x option causes the builder to generate an additional file of output. This file, called ***omdb.xref***, contains a cross-reference between each database key and the text of the related output report. It is an ASCII file and is intended to be forwarded to the appropriate documentation department to form part of the OM manual. Other information included in the cross-reference is message class and alarm level for each message.

5.31 The −s option indicates to the builder to print statistics about the current OMDB. These statistics are printed to ***stdout*** and contain information about each of the sections of the OMDB, the keys not used in the OMDB, and the size of the disk and incore OMDBs.

5.32 The builder allows for the generation of full or incremental builds, i.e., it can start from scratch or it can add to an existing database. This is possible because each section of the database generated from any build grows dynamically with the addition of new keys and data.

5.33 If errors are detected during the build process, the builder adopts one of two strategies based on the presence or absence of the −u option. If the −u option is not specified, the majority of errors will cause an appropriate error message to be generated, and the build will stop after the status quo has been restored. However, if the −u option is specified, a duplicate key error that would have been considered fatal without the −u option can be accepted and the build will continue. In such a case, an appropriate warning message is generated. Error and warning messages are generated on the standard error channel, ***stderr***.

OMDB Key Assignment Administration

5.34 An overview of the UNIX RTR OMDB key assignment is shown in Fig. 10. To request a new output message key, an interactive script is available on all UNIX RTR development machines. This script, ***get_omdb_key***, will prompt the developer for subsystem, ***.ofmt*** file name, and identifying text of the output message. The identifying text should be taken from the first text line in the ***.ofmt*** file (i.e., prototype, dumpline, or tblhdr line) for that message. The first 30 characters entered as identifying text and the ***.ofmt*** file name will be recorded in the administrative key file for reference. The ***get_omdb_key*** will determine the developer's machine using ***uname*** and his/her login using ***logname***. This script will then send a request to the UNIX RTR key administrator via ***nusend*** to assign the next OMDB key available, passing the subsystem name, identifying text, and return electronic address.

5.35 The ***assignkey*** program will be executed under the key adminstrator's login when it receives a developer's request for a new key. The ***assignkey*** will determine the next available key by examining the administrative key file (***omdb.key***), mark that key as used, fill in that key's entry with the subsystem and identifying text of the output message. The UNIX RTR keys will be assigned consecutively from 0 to MAXRTRKEY. MAXRTRKEY is defined as FIRST_APPL_KEY-1, and FIRST_APPL_KEY is defined in ***omdb.h***. The date the key was as-

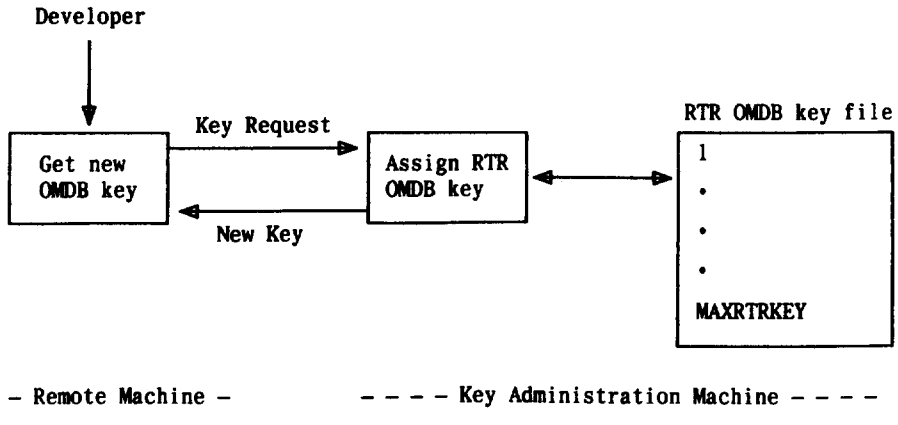


Fig. 10—UNIX RTR OMDB Key Assignment

signed will be determined from the system date and also noted in the file. The layout of the key file is shown in Table C. To everyone besides the key administrator, the key file will be read-only. The new key will be sent back to the requesting UNIX RTR developer electronically using the machine name and login determined by the remote script ***get_omdb_key***.

5.36 It is the developer's responsibility to make note of the new key and use that key in his/her ***.ofmt*** file and in the corresponding format-and-spool call.

5.37 In addition to updating the administrative key file, ***assignkey*** will log the request, including the electronic return address, in a separate log file. This will provide the key administrator with a record of all requests and any error messages produced during a key assignment attempt.

5.38 An overview of the application OMDB key assignment is shown in Fig. 11. To request a new output message key, an interactive script is available on all application machines. Applications using this script must customize the script with the machine address and login of their key administrator (where the ***assignappkey*** tool is installed). Applications must provide their own subsystem names (or other identifiable entities) to the ***get_app_key*** script. This script will prompt the application for subsystem, .ofmt file name, and identifying text of the output message. The identifying text should be taken from the first text line in the .ofmt file (i.e., prototype, dumpline, or tblhdr line) for that message. The first 30 characters entered as identifying text and the .ofmt file name will be recorded in the administrative key file for reference. The ***get_app_key*** will determine the application's machine using ***uname*** and his/her login using ***logname***. This script will then send a request to the key administrator via ***nusend*** to assign the next OMDB key available, passing the subsystem name, identifying text, and return electronic address.

5.39 The ***assignappkey*** program will be executed under the key administrator's login when it receives a request for a new key. The ***assignappkey*** will determine the next available key by examining the administrative key file (***omdb.keys***), mark that key as used, and fill in that key's entry with the subsystem and identifying text of the output message. The keys will be assigned consecutively from MAXRTRKEY+1 to MAX_KEY. (MAX_KEY is defined in ***omdb.h.***) The date the key was assigned will be determined from the system date and also noted in the file. The layout of the key file is shown in Table C. To everyone except the key administrator, the key file is read-only.

5.40 In addition to updating the administrative key file, ***assignappkey*** will log the request, including the electronic return address, in a separate log file. This provides the key administrator with a record of all requests and any error messages produced during the key assignment attempt.

5.41 Because of the potential for reapplying craft message class and alarm level updates to the wrong messages after a field update of the OMDB, ***keys should not be reused***. For example, if a craftsperson had changed the alarm level of message 101 to INFO (because the content of the message text associated with that key was not deemed worthy of a higher alarm) and the message key 101 was reused in a successive load to reference a truly critical alarm, the re-APPLY of the craft changes would result in

TABLE C

OMDB KEY FILE LAYOUT

KEY	DATE	SUBSYSTEM	MESSAGE TEXT	.OFMT FILENAME
0	850101	cft	REPT CSOP IN SERVICE	csop.ofmt
1	850501	cft	REPT OP LOG COMPLETE	oplog.ofmt
2	850502	pudrv	SET IODRV	setiodrv.ofmt
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Note: Headings do not appear in the actual key file.

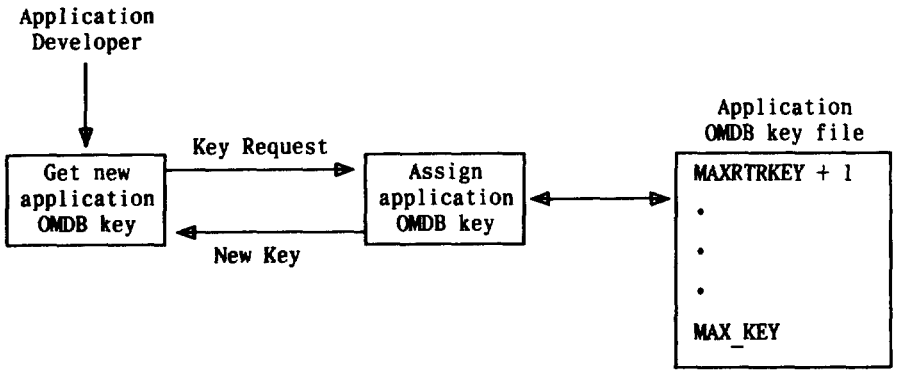


Fig. 11—Application OMDB Key Assignment

the alarm level of that message being changed to INFO.

5.42 It is possible, however, at some time in the future there may be a need or desire to reuse keys. Applications must make their own decision about their policy on the reuse of keys with the realization of the impact of reusing a key within the same load. A "load" must be defined by the application; it does not necessarily mean a single load. Any "load" (including individual loads packaged together) that contains a deletion of a message and reuse of the same key could be disastrous.

E. CSOP and SOP

5.43 CSOP is the process responsible for the new tasks of initializing and accessing the OMDB, formatting of output messages, and a part in the change capability. Fig. 12 shows the flow in CSOP of these new tasks as well as existing CSOP tasks that play a part in this feature.

5.44 Initialization within CSOP must allocate memory for the active OMDB and populate the memory from the OMDB disk file. After initialization, CSOP enters a loop of receiving interprocess messages. Two new message types will be received by this loop.

5.45 Existing spool requests (See b in Fig. 12) will continue to be processed in the current manner. The new format-and-spool requests (see a in Fig. 12) will cause a read of the active OMDB. The new output message will be formatted using the test from the OMDB and parameters from the client process. The request will be passed on to CSOP spooling and treated the same as if a spool-only request had been made.

5.46 The second new message type that CSOP will recognize is for requests to update, activate, output, or apply the logged changes to the OMDB (see c in Fig. 12). The CSOP will be the only process to access either the active OMDB or the OMDB disk file. Therefore, much of the work for the change capability is done in CSOP, not in the input commands. To handle the change capability requests, CSOP will access the appropriate OMDB copy and then continue processing. Update and apply will result in writing the OMDB disk file. An activate request will cause the active OMDB to be repopulated. Output for the OP:OMDB command will be given to the formatting-and-spool process.

5.47 Three new text buffers are provided to pass message text from CSOP to the SOP or the SOF. Changes are provided in the SOP process and to the SOF allowing the SOP/SOF to manage the new buffers.

5.48 ***OMDB Initialization:*** Before CSOP can begin processing any output messages, it must first create the incore OMDB. CSOP maintains a field, ***max_db_segs_incore***, which is a copy of the ECD field (***omdb_mode***). This field indicates how many segments of the OMDB should be kept in CSOP's address space at any given time. The ***max_db_segs_incore*** field is used during CSOP initialization and whenever data for a message is being referenced in the output buffer section. Upon successful

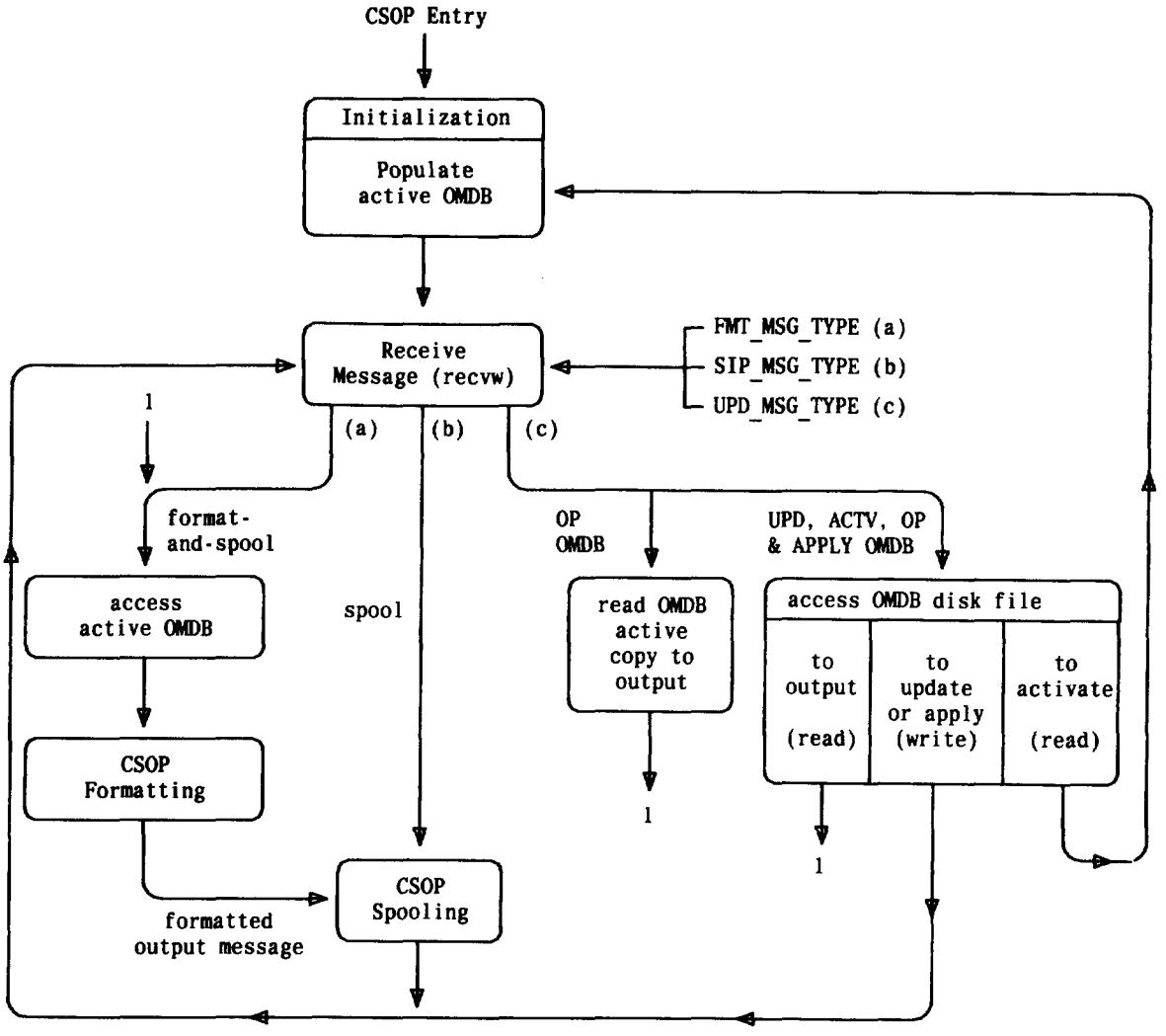


Fig. 12—CSOP High Level Design with OMDB

creation of the incore OMDB, the following message is printed:

REPT CSOP IN SERVICE
OMDB SEGMENTS:

TOTAL	ACTIVE
a	b

—***a*** gives the total number of segments for the OMDB.

—***b*** gives the number of segments that are active (incore).

5.49 ***OMDB Access Routines:*** In order for CSOP to access the data in the various sections of the incore OMDB, five internal interface routines are required. These are described in the following paragraphs. The ***get_ospec()*** routine is the only access routine to check the validity of any of its arguments. All other routines assume valid arguments are being passed.

1. ***get_ospec***(key) — The ***get_ospec()*** routine finds the OMSGSPEC corresponding to the requested key. It does this by comparing the key with the last UNIX RTR key and with the first and last application key. By doing these comparisons, the validity of the key is checked.

If it is valid, then the OMSGSPEC is found by doing a simple pointer calculation. After locating the OMSGSPEC, the function also verifies the OMSGSPEC is in use. If it is not in use, BAD_Key is returned to the caller. If the OMSGSPEC is not in use, a pointer to this OMSGSPEC is returned to the user. When CSOP is operating in primitive mode, because the incore OMDB was not able to be created, ***get_ospec()i*** returns NO_OBUF.

2. ***get_obuf***(&OMSFSPEC) — The ***get_obuf*** function takes as an argument an OMSGSPEC pointer, and after performing some administrative actions, returns to the caller a pointer to the output buffer for this OMSGSPEC.

3. ***get_nxt_var*** — The ***get_nxt_var()*** macro returns a pointer to the next VAR structure to be processed. The macro simply increments the ***nxt_VAR*** variable.

4. ***reset_var(VAR_value)*** — This macro is used to reset the ***nxt_VAR*** variable to a previous value (the first argument). This routine is useful when VAR structures must be processed mulitple times as in the case when formatting dumplines.

5. ***find_enum(&VAR,enum_value)*** — The find function takes two arguments, a VAR pointer and an enumeration value. The function searches the VAR pointer's enumeration structures for a match of the specified enumeration value. If a match is found, a pointer to the enumeration string is returned to the caller. This function must also determine whether the enumeration is private or common, so the ***com_enum*** field in the VAR structure must be taken in to account before any searching is done. NULL is returned to the caller when no match is found.

F. Formatting

5.50 The new function is CSOP, ***fmtmsg***, will process the format-and-spool requests. Input to ***fmtmsg*** is a structure of type ***somb_req*** which contains the input from the client. This structure is in ***spooler.h***. The message text and additional information necessary to format the message must be taken from the OMDB.

5.51 When the formatting is complete, ***fmtmsg*** will call ***sipmsg*** (the CSOP function that handles the spooling). ***sipmsg*** expects to receive a struture of type ***som_req*** (also in ***spooler.h***). The ***fmtmsg*** will set up a copy of ***som_req*** to pass to ***sipmsg***.

5.52 The first step in ***fmtmsg*** is to access the OMSGSPEC for the message key from the client via the ***get_ospec*** routine. The message key is in the ***Key*** field in ***splomdb_args*** (in ***spooler.h***). Error conditions that can be returned by ***get_ospec*** are BAD_KEY and NO_OBUF.

5.53 If BAD_KEY is returned, an OMSGSPEC could not be found for this message key. An error message will be given and all processing for this format-and-spool request will be terminated. The error message is:

REPT CSOP FORMATTING

BAD KEY PASSED BY CLIENT

KEY: a PID: b UTILITY ID: c

— a contains the invalid key.

— b contains the process id of the requesting process.

— c contains the utility id of the requesting process.

5.54 If NO_OBUF is returned, the named segment cannot be accessed, i.e. no OMSGSPECs can be accessed. To fill in a ***som_req*** structure for spooling of the primitive mode message, information passed by the client will be used. For information that can only be found in the OMSGSPEC, default values will be used.

5.55 Given that an OMSGSPEC is found, ***fmtmsg*** will proceed with setting up a ***som_req*** structure with the administrative information. Next, ***fmtmsg*** must access the text of the message and do the formatting. The ***get_obuf*** function will be used to obtain the text. The function will return a pointer to a character string which contains all the text for this message key, null terminated. The NO_OBUF will be returned if a problem occurs in accessing the segment which contains the message text for this key. The request will be processed in primitive mode

using the message class and alarm level already set in ***som_req***.

G. Change Capability

5.56 The Change Capability involves a new UNIX-level process for the input commands and changes to CSOP for access to the OMDB. The new process for the input commands is described in paragraph 5.68. The design changes for CSOP is described as follows.

5.57 To accommodate the interface between the input command process and CSOP, a new message type and new structures are added to spooler.h. The new message type is UPD_MSG_TYPE and the structures involved are: struct, update, struct output, and struct change_req. The input command will populate the change_req structure and pass it to CSOP in an interprocess message of the type UPD_MSG_TYPE.

5.58 The CSOP will assume that the information passed in the change_req structure has been validated by the input commands. However, validation of the key by the input command only insures that the key is within the valid range; further validation will be done by CSOP as it accesses the OMDB.

5.59 The CSOP is responsible for giving an indication of the result and letting the craftsperson know that the command has completed. To do this, CSOP will call its own format-and-spool processing to output most messages. Since the output messages that report a failure to access the OMDB will occur when CSOP cannot access either the disk or active copy, those messages will not be in the OMDB but rather in an "essential messages" header file and will be spooled.

H. RC/V Capabilities

5.60 The incore OMDB is affected if the ***omdb_mode*** field in the spooler information record is changed. A change to this field will not necessarily have an immediate impact, but CSOP will update to the new value as it is processing format-and-spool requests.

5.61 When the ***omdb_mode*** field is changed to the incore ECD, RC/V will send a message to CSOP after a successful TREND. The CSOP will update the ***max_db_seg_in-core*** variable. If the value of ***max_db_seg_in-core*** has been decremented, the number of OMDB output buffer segments incore will be adjusted immediately. If the value has been incremented, the adjustment will not be made until the next format-and-spool request is made and ***get_ospec*** is called.

I. Disk Independent Operations

5.62 A new OST, ***dio_notify***, is added to the operating system subsystem to inform the system when it is in the disk simplex mode. A call to this OST will inform the operating system that the calling process wants to be informed about DIO state changes. CSOP will call ***dio_notify*** during its initialization.

5.63 When disk simplex mode is entered, the operating system will send a fault (received by CSOP as a signal). Only processes that have called ***dio_notify*** will receive this notification. The CSOP will then attempt to bring any OMDB segments incore that are not currently there in anticipation that disk limp mode could occur. The CSOP cannot wait until disk limp mode as the OMDB segments would not be available at all if they were not already incore. Applications must engineer their memory requirements, allowing for the maximum number of OMDB segments, in order to guarantee that all segments can be brought incore during disk simplex operation.

5.64 If a format-and-spool request is made for a message whose text is not in one of the incore segments, the message will be output in primitive mode. If primitive mode is not acceptable, the only alternative is to run with all OMDB segments incore at all times.

5.65 The CSOP will receive notification via the fault/signal mechanism when the system is returned to disk duplex mode. The CSOP will return to the state it was in before the notification of disk simplex mode.

J. Format-and-Spool Interface Functions

5.66 In order for processes and the kernel to output messages which are contained in the OMDB, five new interface library functions are provided. The purpose of all but one of these functions is to convert the client process's arguments into a ***somdbl_req*** structure format and send this structure in a message to CSOP. The client processes must pass one

argument to these functions, the address of a ***splomdb_args*** structure. The ***splomdb_args*** contains information on where to find the user's data, the size of the data, and fields which are used to convey information which would previously be passed as part of the user control string. All fields of the ***splomdb_args*** structure must be filled in by the client, since this structure will be passed intact to CSOP. The fifth interface function is provided so that any craft process can obtain the message class and alarm level for a given key.

5.67 ***splomdb_args Fields:*** The ***splomdb_args*** structure contains all the information necessary for initiating an OMDB spool request.

K. Change Capability Input Commands

5.68 Since CSOP is responsible for accessing the OMDB to do the work associated with the change capability input commands, this leaves little work besides parsing for the input command process-(es). Therefore, a single new process is provided to handle all four new input commands:

(1) UPD:OMDB:KEY = a:DATA, {MSGCLS = b ¦ ALARM = c};

(2) ACTV:OMDB;

(3) OP:OMDB:{DISK ¦ ACT}:KEY = a;

(4) APPLY:OMDB;

This new process will group together all craft interfaces to the OMDB.

5.69 In response to the craftsperson typing one of the new input commands, and after successfully parsing and validating the command, the craft shell will fork and execute the command process. The command process is also responsible for doing parsing and parameter validation, so if the craft shell is in IM catalog-limp mode and cannot fully validate the input message, only expected values will be passed to CSOP. This process, ***updomdb,*** will determine which activity is desired: update message class or alarm level (UPD), refresh CSOP's active copy of the OMDB (ACTV), output one or more OMDB entries (OP), or apply previous updates during field update (APPLY). For UPD or OP, ***updomdb*** will check that the value(s) entered for KEY are numeric and with the valid range for an OMDB key. For UPD, this process will check that if MSGCLS is specified, the value is numeric and within the valid range for message classes (0-255). If ALARM is the specified keyword on the UPD command, ***updomdb*** will check that the value specified is one of the following: CRIT, MAJ, MIN, MAN, ACT, INFO, or VAR. For OP, the command process will determine if the craftsperson wishes to reveiw the DISK or ACTIVE copy of the OMDB. Any invalid values will result in an appropriate error acknowledgement.

5.70 If the command entered was syntactically correct, the command process will set the ***cmd_type*** flag in the message structure ***change_req*** indicating the desired operation (update, activate, output, or apply), and fill in any data associated with the UPD or OP operations in the appropriate structure (struct update for UPD or struct output for OP). These structures are defined in ***spooler.h***. The dialogue information must be taken from the spooler environment variable and also placed in the ***change_req*** structure.

5.71 A maximum of 32 keys can be sent to CSOP for processing at one time for an UPD or OP command. If fewer than 32 keys are specified on the command line, the command process will fill in a value of −1 in the key array to mark the end of the keys being passed for the current request. After filling in the necessary information in the ***change_req*** message structure, the command process will issue an "IP" acknowledgement for update, activate, or apply requests or a "PF" for an output request.

5.72 The ***updomdb*** will then send an interprocess message of type UPD_MSG_TYPE (also defined in ***spooler.h***) to CSOP using ***sendpw()***. If the ***spendpw()*** is not successful, indicated by a return value of −1 from the ***sendpw()***, ***updomdb*** will call ***splomdb()*** to format and spool the following output message:

a OMDB ABORTED
UNABLE TO SEND REQUEST TO CSOP

— a is the verb of the command being executed: ACTV, APPLY, OP, or UPD.

The CSOP is responsible for issuing a completion report for successful commands, or reporting errors it detects while attempting to do the requested operation.

L. OMDB Field Update Procedure

5.73 A new OMDB field update procedure is provided because different steps are required to install a new OMDB than for existing products. Changes to implement this procedure are provided in the field update ***mkmsg*** source and in the SG database.

5.74 During a field update, the OMDB is not merely copied onto the 3B20 computer system; after installation into /cft/spl/omdb, CSOP must be forced to read this disk file and refresh its incore copy. The command ACTV:OMDB exists to do precisely this, therefore, this command must be included in the MSGS file. To prevent the loss of craft message class and alarm level updates made to the previous version of the OMDB disk file, the command APPLY:OMDB must be included in the PERM phase of the MSGS file before an ACTV:OMDB. The APPLY:OMDB cannot be done as part of the APPLY phase because of the nature of the command (it accesses /cft/spl/omdb which is the file being field updated) and the way in which the field update works.

5.75 In summary, the steps required for the OMDB during various phases of a field update are:

APPLY phase

—Save a copy of /cft/spl/omdb.
—Install new OMDB file in /cft/spl/omdb.
—Execute ACTV:OMDB.

BACKOUT phase

—Restore /cft/spl/OMDBLOG.
—Execute ACTV:OMDB.

PERM phase

—Execute APPLY:OMDB.
—Execute ACTV:OMDB.
—Remove saved file.

Since a MSGS template is generated according to field update type, the new field update type for the OMDB is added to the SG database.

M. Input Message Acknowledgements

The Acknowledgement Database

5.76 The database is a table of variable length acknowledgements separated by null characters. Each unique acknowledgement has a numeric key associated with it which is used to build the database but is not part of this database. The very beginning of the database is an array of variable size, ordered by key, of the offsets of the various text entries. The first two bytes of the database contain the number of bytes required to store the keys for RTR acknowledgement messages. The following two bytes of the database contain the number of bytes required to store the key values of application messages. The first four bytes of the database thus determine the beginning address of the text entries which will immediately follow this array. This makes it possible to reference an acknowledgement without any searching or hashing to keep the performance requirements intact. In addition, the only unused space in the database is the offset entries of unused keys. Careful administration of the keys will help avoid unused keys in the database. The array will allow maximums of 1000 keys for UNIX RTR and 2000 keys for applications. The space between the maximum RTR key and the first application key will be closed.

5.77 At run-time the database resides in a named segment, specifically segment 45, which is initialized by DAP, a Craft Shell, or a Dialogue Shell, whichever is created first. If the shells and DAP never successfully initialize after a boot, the Acknowledgement Database (ACKDB) will not be available. The segment is part of every shell's address space, ensuring that as long as one can enter input messages, the database is accessible in main memory. (Note that the ACKDB is shared by all CFTSHLs, DLGSHLs, DAP, and all client processes.)

5.78 It should be noted that since DAP is DIO essential and DAP has the database in its address space, in effect the ACKDB is DIO essential.

5.79 The two character acknowledgement codes and supplemental information have separate entries in the acknowledgement database to provide flexibility to the user and to conserve database space. A complete acknowledgement consists of a two character code and up to three sections of explanatory text. This corresponds closely to the PDS and MML standards, which prescribe certain sections of infor-

mation for certain types of acknowlegements. The standard set of acknowlegements provided includes text for each of these sections.

5.80 The disk copy of the database resides in ***/cft/shl/ackdb.***

Acknowledgement Format Files

5.81 The format files for the database, called "***.afmt***" files, consists of lists of acknowledgement specifications. An example of a "***.afmt***" file might be:

```
#include "defines.h"

#define BADCAT   24

key    2
text   "NG"

/*This message indicates that the IMCATLG
is unavailable */

key    BADCAT
text   "BAD IMCATLG"
```

5.82 Comments are enclosed in /* */. Constants may be #defined to integer values in either the ***.afmt*** file or in a #included file. A new global header file, ***acks.h***, is added which contains #defines of key numbers for the most common acknowledgements.

5.83 An acknowledgement specification is two lines, the key and the text. Each specification is followed by one or more blank lines. For UNIX RTR, there is one format file per subsystem, each located in a new directory (***afmt***) one level below the subsystem directory.

Acknowledgement Database Building Tools

5.84 The ***afmtparse*** will accept an input file (the file name must end with ***.afmt***) and call the C preprocessor to resolve all #defined constants and remove comments. Then, ***afmtparse*** will remove the quotation marks from the text strings. It will output a file named the same as the input file except "***afmt***" will be replaced by "***am***".

5.85 The following ***.am*** file corresponds to the sample ***.afmt*** file:

```
key    2

text   NG

key    24
text   BAD IMCATLG
```

It is suggested that applications also limit each subsystem to one ***.afmt*** file since the names of all ***.am*** files must fit on the command line for ***3babld.*** In the UNIX RTR Project, the ***.am*** files are kept in a new global directory, "***ack***", and is available on all development machines.

5.86 The builder (***3babld***) will accept ***.am*** files as arguments, insert each text line contiguously into the database in the next available location, and place that relative location into the offset array at the location specified by the key. The ***3babld*** is supported in both maxi and 3B20S computer environments. The existing OMDB key assignment tools are copied and modified to assign and maintain acknowledgement keys. The ACKDB key assignment tools prevent duplicate keys from being assigned but will not check for duplicate text. Developers are able to examine all messages contained in the ACKDB using the ***dmpackdb*** command. However, ***dmpackdb*** will not be able to identify the subsystem which owns a given message and thus, not be able to indicate where a constant is #defined. The ***3babld*** is responsible for flagging multiple uses of a key as an error.

New libCFT and libminCFT Functions

5.87 ***ackdb_init( ):*** This function is called by the CFTSHL, the DLGSHL, or DAP to read the database from disk into memory, if it has not already been done. The supervisor calls ***alocseg( ), addseg( ), setmap( ),*** and ***unbikseg( )*** which is used to initialize the segment. Synchronization will be handled inside this function. The return code from ***alocseg( )*** will determine the state of the ACKDB and whether it must be read from disk.

5.88 ***query ackdb (key1, key2, key3, key4,*** and ***buffer):*** This function will accept four keys and a buffer as arguments, and place the retrieved acknowledgement into the buffer. One key is for the two character code, and the other three, which may be NULL if desired, are for the optional explanatory

text. The DAP, CFTSHL, and DLFSHL will use this function directly because they perform their own I/O. The database will be accessed in the following manner. Let ACKDBASE be the base address of the database. Then for a given key, the text for that key can be found by looking at the offset entry for that key and adding it to ACKBASE + (the size for the offset array).

5.89 ***get_ackdb( ):*** The ***get_ackdb*** function will inspect the process's pcb to see if the ACKDB is already part of its address space. If not, it will make the acknowledgement database part of the calling process's address space, giving the segment read-only permission. It will return 0 if successful, −1 if not.

5.90 ***rmv_ackdb( ):*** This function will remove the ACKDB from a process's address space.

5.91 ***ackudb( ), ackldb( ), acklpdb( ),*** and ***ackip2db( ):*** These functions will act as fronts for their counterparts, ***acku( ), ackl( ), acklp( ),*** and ***acklp2( ).*** When a command process calls one of these new functions, it will pass it a key for the two character ack and up to three keys for the explanatory text. If less than four keys are desired, the last argument must be 0. Then ***get_ackdb( )*** will be called to attach to the database. Next, ***query_ackdb( )*** will be called, being passed the key for the two character code and the keys for the text, if any. The acknowledgement database function will receive the concatenated acknowledgement associated with the keys and call its corresponding acknowledgement function [***acku( ),*** etc.], passing this function the retreived acknowledgement. Errors involving an invalid key will be handled with an error indication of "NA - INVALID KEY", and the process will be allowed to continue. In the case of ***ackudb( ), rmv_ackdb( )*** will be called to remove the ACKDB from the process's address space. Acknowledgement trapping will still be done.

5.92 ***trapacksdb( ):*** This function will trap and/or translate acknowledgements for the ***libCFT ack*** functions in support of improved input message acknowledgements time for the ACKDB feature. ***Trapacksdb( )*** will be called from ***ackudb( ),*** etc., instead of ***trapacks( )*** when the UNIX Craft Shell has already done the acknowledgement for an input message.

5.93 ***inechodb( ):*** When a command process calls this function it must pass it four keys (including 0) and the output message class. This function passes the original input line and the specified acknowledgement to the output message spooler for the input message echoing function. This function will call ***query_ackdb( )*** to retrieve the acknowledgement portion of the input message and then will concatenate both before passing it to the spooler.

Craft Shell and Dialogue Changes

5.94 The Craft Shell and Dialogue Shell currently retrieve an acknowledgement string, either IP, PF, or OK, from the IMCATLG. The Shell will test this string and choose the proper key to use in ***query_ackdb( ).***

Performance

5.95 Since no searching or hashing is required, the vast majority of the additional time it will take to write an acknowledgement is consumed by adding the database to each calling process's address space. This is estimated to take less than 14 milliseconds.

ACKDB Field Update Procedures

5.96 A new command, CLR:ACKDB, makes sure the database exists on disk, and then clears the segment name, using the ***clrname( )*** supervisor call. This makes sure that any new processes must use the new version and the ACKDB must be reinitialized for them. If a process is currently attached to the database, it will use the old copy of the segment. Note that once a process does an unlocking acknowledgement, the ACKDB will be removed from its address space.

5.97 The apply phase can be done as follows. Do the file replacement for the ACKDB, execute CLR:ACKDB, and on the 103 page poke the Shell Restart and then the DAP Restart.

5.98 To back out of the changes, execute the field update backout command, execute CLR:ACKDB, and kill the shells and DAP again via the 103 page.◆

6. GLOSSARY OF TERMS AND ACRONYMS

GLOSSARY OF TERMS

6.01 A glossary of terms and acronyms is provided to aid in the understanding of this document.

Audit —Validity checks that are performed to assure the proper operation of the operating system and the authenticity of its data structures.

Baud —A unit of signaling speed expressed in the number of bits per second.

Daemon —In software, a process that controls information or other processes with unusual effectiveness.

Delimiter —A type of token used to separate other tokens or define fields within a message. For example, the terminal message token ":" separates the verb token from the first id token.

Firmware —Software instructions stored in read-only memory.

Histogram —A graphical representation of parameters showing distribution, deviation, failure limits, and sample size by means of rectangles whose widths represent class intervals and whose heights represent corresponding frequencies.

Identifier (id) —A type of token used to represent an object or specific hardware unit. For example, the terminal message token "CU" represents a control unit.

Maintenance Terminal —The terminal(s) used for the exchange of maintenance information/commands between the maintenance person and the computer.

Off-line —A CC is off-line if it is not in control of system configuration and execution although it may be active (executing diagnostics).

On-line —A CU is on-line if it is actively executing. More specifically for a 3B20D computer, the on-line CU is that which is in active control of system configuration and execution.

Parse —To resolve data into a collection of predefined tokens based on syntactical relationships. For example, a sentence can be parsed into words. Parsing is an operation in the execution of terminal messages.

Poke —The activation of a virtual key that is a portion of a virtual panel displayed on a maintenance terminal. A poke is performed via cursor position, light pen, or menu mode keyboard input.

Process —The basic executable entity in DMERT or UNIX RTR.

Queue —A waiting list usually consisting of processes or tasks waiting for processor time to execute.

Random-Access Memory—Memory that can be written into and read from. Any element can be accessed with equal ease.

Read-Only Memory —Memory that cannot be written into. The contents of read-only memory can be read, but never destroyed via overwrite.

Receive-Only Printer —A terminal that is designed for output only. The maintenance person cannot originate input from a receive-only printer. This type of terminal may be used to maintain a printed record of the PDS/MML type I/O communication associated with a maintenance terminal.

Sequenced-Mode Audits —A group of audits that are performed in a particular sequence in order to successfully complete tests and maximize their efficiency.

Time-Out —A system action based upon the absence of an expected event during a prescribed time interval.

Token —A defined character or sequence of characters that represent a particular action, object, or a type of delimiter. For the 3B20D computer, most tokens are either verbs, identifiers, or delimiters.

Verb —A type of token that represents an action. For example, the terminal message token "DGN" represents the request to diagnose.

ACRONYMS/ABBREVIATIONS

6.02 The following acronyms are used within this section.

ACRONYMS	WORDS
ACKDB	Acknowledgement Database
ACP	Alarm Control Process
AIM	Application Integrity Monitor
ASCII	American Standard Code for Information Interchange
ASW	All-Seems-Well
AUDMGR	Audit Manager
CC	Central Control
CFTSHL	Craft Shell
CLREAI	Clear Emergency Action Interface
CMON	Craft Interface Integrity Monitor
CRON	Clock Daemon
CSOP	Coordinator of the Spooler Output Processor
CU	Control Unit
DAP	Display Administration Process
DFC	Disk File Controller
DMERT	Duplex Multienvironment Real-Time
DTIM	Disable Sanity Timer
DUI	Direct User Interface
DUIC	Direct User Interface Controller
EAEN	Emergency Action Enabled
EAI	Emergency Action Interface
EAIMRF	EAI Maintenance Request Functions
ECD	Equipment Configuration Database
EFTBL	Enumeration File Table
EIH	Error Interrupt Handler
ESS	Electronic Switching System
FBDP	Force Boot Device Primary
FBDS	Force Boot Device Secondary
FOFL	Forced Off-Line
FONL	Forced On-Line
FSMON	File System Monitor
ID	Identification
IM	Input Manual
I/O	Input/Output
IOP	Input/Output Processor
LED	Light Emitting Diode
LLA	Low Level Access
MHD	Moving Head Disk
MIRA	Maintenance Input Request Administrator
MML	Man Machine Language
MRF	Maintenance Reset Function
MT	Magnetic Tape
MTC	Magnetic Tape Controller
MTTYPC	Maintenance Teletypewriter Peripheral Controller
OMDB	Output Message Database
OPQAUD	Operating Queue Audit
OPQUEUE	Operating Queue
OST	Operating System Trap

ACRONYMS	WORDS
PADL	Page Descriptor Language
PDF	Page Descriptor Files
PDS	Program Documentation Standards
PDSHL	Program Documentation Standards Shell
PMS	Plant Measurement System
PONL	Processor On-Line
PRM	Processor Recovery Message
RAM	Random-Access Memory
ROM	Read-Only Memory
ROS	Request Out-of-Service
RST	Restored To Service
RTS	Real-Time Status
SCC	Switching Control Center
SCCS	Switching Control Center System
SC/SD	Scanner/Signal Distributor
SDL	Synchronous Data Links
SDLC	Synchronous Data Link Controller
SG	System Generation
SIM	System Integrity Monitor
SIOF	System Integrity Output Formatter
SIP	Spooler Input Processor
SOP	Spooler Output Processor
TTY	Teletypewriter
UART	Universal Asynchronous Receiver Transmitter
UCB	Unit Control Block
ULARP	UNIX Level Automatic Restart Process