AD-A076 501 INTERNATIONAL COMPUTING CO BETHESDA MD F/G 9/2
VESSEL TRAFFIC SERVICES PROCESSING/DISPLAY SUBSYSTEM DETAILED S--ETC(U)
JUL 79 D A COHN , F T MICKEY DOT-CG-81-78-1833
UNCLASSIFIED USCG-D-66-79 NL

REPORT NO. CG-D-66-79

LEVEL

VESSEL TRAFFIC SERVICES
PROCESSING/DISPLAY SUBSYSTEM
DETAILED SOFTWARE DESIGN
OPERATING SYSTEM

DAVID A. COHN
FRANK T. MICKEY
International Computing
4330 East-West Highway
Bethesda, Maryland 20014

U.S. Coast Guard Research and Development Center
Avery Point Groton Connecticut 06340

AD A076501

DDC
RECEIVED
NOV 7 1979
E

JULY 1979

FINAL REPORT

THIS DOCUMENT IS BEST QUALITY PRACTICABLE.
THE COPY FURNISHED TO DDC CONTAINED A
SIGNIFICANT NUMBER OF PAGES WHICH DO NOT
REPRODUCE LEGIBLY.

DDC FILE COPY

Document is available to the U.S. public through
the National Technical Information Service
Springfield, Virginia 22161

79 11 07 075

PREPARED FOR

U.S. DEPARTMENT OF TRANSPORTATION
UNITED STATES COAST GUARD
OFFICE OF RESEARCH AND DEVELOPMENT
WASHINGTON D.C. 20590

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
CG-D-66-79		

4. Title and Subtitle
VESSEL TRAFFIC SERVICES PROCESSING/DISPLAY SUBSYSTEM DETAILED SOFTWARE DESIGN OPERATING SYSTEM

5. Report Date
July 1979

6. Performing Organization Code

7. Author(s)

A. Cohn T. Mickey

8. Performing Organization Report No.

9. Performing Organization Name and Address

International Computing Company
4330 East-West Highway
Bethesda, Maryland 20014

10. Work Unit No. (TRAIS)

11. Contract or Grant No.
DOT-CG-81-78-1833

12. Sponsoring Agency Name and Address
U.S. Department of Transportation
United States Coast Guard
Office of Research and Development
Washington, D.C. 20590

13. Type of Report and Period Covered
Final Report
May - July 1979

14. Sponsoring Agency Code

15. Supplementary Notes The contract under which this report was submitted was under the technical supervision of the Coast Guard Research and Development Center, Groton, Connecticut, 06340. R&DC 22-79.

16. Abstract

This report provides the detailed design of an operating system which can support the VTS application in a multicomputer, high reliability environment. Since computer hardware has not been selected, the design represents a generalized operating system which can be implemented on a wide variety of minicomputer systems which the Coast Guard may wish to select for the VTS Processing/Display Subsystem. The discussion here includes an overview of the Executive, a detailed description, as well as structure charts, Program Design Language specifications and a description of the system data structures used by the Executive. Basic considerations for the design and implementation of fault tolerant software, with specific application to VTS, are also presented. In conclusion constraints on hardware selection and assumptions that were made concerning manufacturer-provided development software, in regard to this VTS/OS design, are cited.

17. Key Words
Operating System
Executive
Event Management
Task Management
Inter-Task Communication
Error Reporting & Control
Fault Tolerance

18. Distribution Statement
Document is available to the U.S. public through the National Technical Information Service, Springfield, Virginia 22161

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	360	

Form DOT F 1700.7 (8-72) Reproduction of form and completed page is authorized

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
2	OVERVIEW OF VTS/OS DESIGN DOCUMENTATION	2-1
3	THE VTS/OS EXECUTIVE	3-1
3.1	Design Philosophy	3-2
3.2	Overview	3-4
3.3	Detailed Description	3-7
3.3.1	The Executive	3-7
3.3.2	Executive Level Service Routines	3-10
3.3.3	Data Structures	3-17
3.4	Executive PDL Specifications	3-29
3.4.1	Executive	3-29
3.4.2	Callcomplete	3-38
3.4.3	Eventcomplete	3-40
4	USER'S MANUAL	4-1
4.1	Event Management	4-3
4.1.1	Events	4-3
4.1.2	Time-Interval Waits	4-4
4.1.3	Event-terminated Waits	4-6
4.2	Program and Task Management	4-9
4.2.1	Overview	4-9
4.2.2	Program Management Calls	4-10
4.2.3	Task Management Calls	4-12
4.3	Memory Management	4-14
4.3.1	Task Memory Structure	4-15
4.3.2	Overlay Control	4-16
4.3.3	Buffer Control	4-18

TABLE OF CONTENTS (CONT.)

Section		Page
4.4	Inter-Task Communications	4-24
4.4.1	Sendmessage	4-25
4.4.2	Requestmessage	4-27
4.4.3	Transfermessage	4-28
4.4.4	Sendanswer	4-29
4.5	File Management	4-30
4.5.1	Disc File Structures	4-30
4.5.2	File Access Methods	4-31
4.5.3	File Maintenance Procedures	4-31
4.5.4	Input/Output Operations	4-35
4.5.5	Data Transfer Operations	4-37
4.6	Error Reporting and Control	4-44
4.6.1	FAILUREDETECTED	4-44
5	VTS/OS DETAILED DESCRIPTION	5-1
5.1	Event Management	5-2
5.1.1	Events	5-2
5.1.2	Time-Interval Units	5-7
5.1.3	Event-Terminated Waits	5-13
5.2	Program and Task Management	5-19
5.2.1	Program Management Calls	5-19
5.2.2	Task Management Calls	5-26
5.3	Memory Management	5-36
5.3.1	System Memory Control	5-36
5.3.2	Overlay Control	5-37
5.3.3	Buffer Control	5-39
5.4	Inter-Task Communication	5-48
5.4.1	Router	5-51
5.4.2	Dispatcher	5-56
5.4.3	Bus Interface Driver Operation	5-62

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist Avail and/or special
A

TABLE OF CONTENETS (CONT.)

Section		Page
5.5	File Mangement and Input/Output Operations	5-63
5.5.1	File Management	5-63
5.5.2	Input/Output Operations	5-85
5.6	Error Reporting and Control	5-106
5.6.1	Data Structures for Error Control	5-106
5.6.2	Procedure Declarations	5-107
5.6.3	MERCUPDATE	5-111
6	PROGRAM DESIGN LANGUAGE SPECIFICATIONS FOR VTS/OS	6-1
6.1	Event Control	6-5
6.1.1	Time-Interval Waits	6-5
6.1.2	Event-Terminated Waits	6-10
6.2	Program and Task Management	6-13
6.2.1	Program Management Calls	6-13
6.2.2	Task Management Calls	6-23
6.3	Memory Management	6-37
6.3.1	Overylay Control	6-37
6.3.2	Buffer Control	6-39
6.4	Inter-Task Communications	6-48
6.4.1	Router Functions	6-48
6.4.2	Dispatcher	6-55
6.5	File Management and Input/Output Functions	6-60
6.5.1	File Management	6-60
6.5.2	Input/Output Operations	6-80
6.6	Recovery and Reconfiguration	6-99
6.6.1	Data Structures and Variable Declarations	6-99
6.6.2	Procedure Declarations	6-100

TABLE OF CONTENTS (CONT.)

Section		Page
7	FAULT TOLERANCE	7-1
7.1	Envirionment	7-1
7.2	General Approach for Achieving Fault-Tolerance	7-4
7.2.1	Redundancy	7-9
7.2.2	Error Handling	7-10
7.2.3	Recovery	7-10
7.3	Hardware Fault Tolerance	7-11
7.3.1	Memory	7-12
7.3.2	Main Processors	7-16
7.3.3	Discs	7-17
7.3.4	Display Processors	7-18
7.4	Software Fault Tolerance	7-20
7.5	Error Detection and Reporting	7-22
7.5.1	Process Failure	7-23
7.5.2	Main Processor Failure	7-24
7.5.3	Position/Collision Processor Failure	7-24
7.5.4	Display Processor Failure	7-25
7.5.5	Bus Failure	7-26
7.5.6	Data Base Failure	7-26
7.6	Error Recovery	7-27
7.6.1	Process Failure	7-28
7.6.2	Main Processor Failure	7-28
7.6.3	Position/Collision Processor Failure	7-29
7.6.4	Display Processor Failure	7-29
7.6.5	Bus Failure	7-29
7.6.6	Data Base Failure	7-30
7.6.7	Other Hardware Failures	7-30
7.7	Reconfiguration	7-31
7.7.1	Current Configuration Information	7-31
8	RECOMMENDATIONS AND CONCULSIONS	8-1

1 INTRODUCTION

This report is the fourth in a series prepared by International Computing Company (ICC) under Contract No. DOT-CG-31-78-1833 for the United States Coast Guard.

The first report* described the initial design study which focused on alternative architectures. The second report** carried the initial design to a greater level of detail, discussed critical design issues and presented basic hardware specifications.

The third report*** presented detailed software specifications and an intermediate level design for the application software. This fourth report is a companion volume dealing with the design of an operating system for the Vessel Traffic Services (VTS) Processing/Display Subsystem.

This fourth report provides a detailed design of an operating system which can support the VTS application in a multicomputer, high reliability environment. The design has been prepared to the level of detail which is appropriate at this point in the total system design. However, since computer hardware has not been selected, it has been necessary to limit the detail provided for device drivers and other hardware specific areas.

The design, therefore, represents a generalized operating system which will be implemented for a wide variety of minicomputer systems which the U. S. Coast Guard may wish to select for the VTS Processing/Display Subsystem.

---

* Henson, C.C., Cleaver, R.A., Kaisler, S.H., "Preliminary Design Study for VTS Processing/Display Subsystem," June, 1978.

** Henson, C.C., Mickey, F.T., Graham, R.S., McIntosh, B. A., "VTS Processing/Display Subsystem Design," January, 1979.

*** Henson, C.C., Graham, R.S., McIntosh, B.A., "VTS Processing/Display Subsystem Software Requirements and Design," July, 1979.

2 OVERVIEW OF VTS/OS DESIGN DOCUMENTATION

The design of an operating system involves the translation of a set of software requirements into a set of program modules, tables and data bases which, when implemented, will perform the required functions. The design process is essentially the selection of techniques, structures, algorithms, etc., from nearly infinite possibilities.

Design is a complex process which is not yet completely understood. Great strides have been made, however, in developing concepts, techniques and tools which can assist the designer in doing his job. These techniques allow the designer to approach his task in an orderly fashion.

In addition to providing some assistance to the designer, recently developed techniques make what may well be a more significant contribution by providing techniques for communicating a design to others so that they can understand clearly what the designer intended. This understanding makes it possible for others to evaluate, critique, and, if need be, alter the design.

In developing and describing the design of the operating system for the VTS Processing/Display Subsystem, we have used a number of techniques and concepts which should be noted before the design is presented.

The design presented here is based upon the preliminary specifications presented in Section 7 of the Phase II report* to the United States Coast Guard. The design presented in that earlier report has been modified in order to provide a more consistent structure and provide the user with a simpler interface to operating system functions.

---

*Henson, C.C., Mickey, F.T., Graham, R.S., McIntosh, B.A., "VTS Processing/Display Subsystem Design," January, 1979.

Each process is designed in this report as a hierarchy using top-down design techniques,** as top-down design leads to well structured, modular software.

Several techniques have been used during the design of the Vessel Traffic Service Operating System (VTS/OS). Initially, ordinary language was used to describe elements of the operating system. Where necessary or useful, the Composite Design techniques of Myers** were used to clarify some of the sequences of transformations of system data structures. Myers' structure charts were also used, although it was found that due to the complexity of the interactions between various operating system routines, these structure charts proved to be less helpful than was initially expected, and seemed in general to be less applicable to operating system design than to application program design. Finally, Program Design Language (PDL) specifications have been used extensively to describe the design.

PDL is a kind of structured English. Many different PDL's have been suggested in the literature; however, the PDL used in this report is based on the PASCAL language structures. PDL is a program-like description of an algorithm, but is often simplified, less detailed, and more readily understood than would be the actual program. The major differences between the PDL used in this report and pure PASCAL are described in Section 6.

Inasmuch as users have no direct interface with the Executive itself, it has been discussed separately in Section 3. This discussion includes a general overview of the Exec, a detailed description, as well as structure charts, Program Design Language specifications, and a description of the system data structures used by the Exec.

**Myers, G.J., "Reliable Software Through Composite Design," Petrocelli/Charter, New York, 1975.

Within this report, three levels of description have been provided for the remainder of the operating system. Section 4, which describes the user's interface with the operating system, is designed to be extracted for use as the nucleus of a User's Manual when the implementation-dependent details of the operating system are known. Section 5, which presents detailed descriptions and structure charts of the VTS/OS routines, is designed for use by those programmers seeking a deeper understanding of the algorithms used by these routines, but who are not interested in the actual code. Section 6, which presents the Program Design Language specifications of these routines, is designed for use both as a guide to the details of the algorithms during the implementation and for subsequent maintenance and modification of the VTS/OS. These PDL specifications are, of necessity, incomplete. Those very low-level routines which interact directly with the hardware, such as small service routines and device drivers, are discussed from a functional point of view, but are obviously impossible to specify at this time.

Concluding the VTS/OS discussion, Section 7 addresses basic considerations for the design and implementation of fault tolerant software, with specific application to VTS. This is presented both as an aid to implementation of VTS/OS and as a guide to allow programmers implementing the VTS applications to integrate their procedures and structures with the fault tolerance approach adopted by VTS/OS.

3 THE VTS/OS EXECUTIVE

This chapter describes the VTS/OS Executive, which is the heart of the entire operating system. First, the philosophy behind the design chosen for the VTS/OS Executive is discussed. This is followed by a general overview of the structure of the Executive, together with the relationship between the Executive, system tasks, and application tasks. A detailed description of the various operating system data structures is then presented. Finally, Program Design Language specifications are given for the Executive.

3.1 DESIGN PHILOSOPHY

The VTS/OS processors will be special purpose, dedicated systems. The software will operate in what is known as a "friendly" environment, since all programs and data structures will have been defined at system generation time and thoroughly tested before the system goes on-line.

In contrast, a general purpose system must provide elaborate safeguards to protect itself against willful or innocent invasion by users. It must also protect one user from another, and if real-time response is required, complex priority and preemption techniques must be employed. Such requirements lead to extremely complex operating systems with unavoidably high overhead.

Elaborate protection mechanisms, complex preemptive priority techniques, and other necessary parts of general purpose systems are neither necessary nor desirable in VTS/OS.

For VTS/OS, ICC has used a non-preemptive scheduling strategy known as a Volitional Executive. The Volitional Executive gives control of the CPU to a task which retains control until it gives up control of its own volition. An executing task is not preempted by a higher priority task. Another task can gain control of the CPU only when the task which is currently executing releases the CPU.

The volitional approach allows processor scheduling to be accomplished in a straightforward manner. A Ready queue is maintained,

comprised of all tasks which are ready for execution. When the CPU is free, the Executive removes the first task from the Ready queue and gives control of the CPU to that task. When that task gives up control, either by using a PAUSE, WAITANY, or WAITALL call (q.v. in Chapter 4), or by requesting any system service with an event flag of SUSPEND (see Chapter 4), the next task on the Ready queue is given control.

If priorities are unnecessary, the Ready queue can be organized as a simple first-in-first-out queue. If priority must be given to some tasks, the Ready queue can be ordered on a priority basis. It must be emphasized, however, that a priority-ordered Ready queue does not imply preemption of the currently running task.

3.2 OVERVIEW

The VTS Operating System provides system and user tasks with controlled access to all system resources. Tasks which are ready for execution are placed on the Ready queue. The task at the head of the Ready queue receives control of the processor, and returns control to the Executive only when it issues a system service call. The information on the requested service and parameters for that request are placed in an Event Control Block (ECB). This ECB will remain associated with this service call for the life of the call.

Based on the function code or class of service (e.g., memory management, I/O, etc.), the Executive calls the appropriate distributor routine. This routine may first perform some preliminary procedures common to all requests in the specified class. It will then call the appropriate service routine for the specific call (e.g., LOADOVERLY, READSQ, etc.) which is identified by the subfunction code. When a synchronous service has been performed, or an asynchronous service has been initiated, the distributor returns to the Executive, and the task at the head of the Ready queue is given control of the processor. A task may remain at the head of the Ready queue while several services are performed by, or initiated by, the operating system.

Within VTS/OS, applications tasks and system tasks compete for system resources in exactly the same way. They will use the same mechanisms for interacting with each other and for interacting with the Executive for scheduling purposes, and will have the same set of calls (as described in Chapter 4) available.

The major distinction between the two kinds of tasks is that system tasks will always be mapped into the system table area, which contains all of the system tables and data structures. Applications tasks will not be mapped into this area while executing applications code. As a result, during compilation, applications tasks will only require access to one global system

dictionary, which will contain those system-wide variables and constants needed by every task. System tasks, however, will require access both to this dictionary and to a second dictionary defining system-level variables, constants, and data structures.

An additional difference, although not a significant one, is that there will be a library of procedures available to all system tasks for performing fairly primitive but frequently needed operations (e.g., link a node onto a queue, remove a node from a queue, etc.). This serves primarily to promote uniformity of technique and structure among the system tasks, which may or may not be desirable for applications tasks. Should it prove desirable, defining a similar library for applications tasks appears to pose no particular problems.

Due to the existence of a separate memory map for each task, a set of small routines, termed "user interface stubs" in this report, will be required. These routines are part of the operating system, but are so heavily dependent on the specific structure of the computer selected and the PASCAL implementation used by that computer that they are described only in terms of their functions. The user interface stubs reside in each task at a specified location and perform a common set of operations for each operating system call. For example, if a user task issues a "SCHEDULE (EFLAG, ..., STATUS)," call, an external reference to the name "SCHEDULE" is created in the user task. This external reference is resolved when the user task is linked, becoming a call to the user interface stub for the SCHEDULE system call. The user interface stub provides for the transfer of CPU control to the operating system, and is responsible for:

1. modifying the task map so that the task is mapped into both the system table area and that portion of the task's address space containing the parameters for the system service call,

2. allocating a free Event Control Block and setting up the fields that are not call-specific,

3. moving the call parameters into the appropriate fields of the ECB (note: the stub may also do some preliminary error checking of the parameters as discussed below),

4. linking the ECB into the ECB-queue for the Executive, and

5. saving the task's context and returning control of the CPU to the Executive.

To keep these stubs as small as possible, since they will use some of each task's address space, a minimum of intelligence should be placed in each. However, it may be reasonable to perform some minimal error-checking at this level before going through the entire cycle described above. Primarily, since almost every system service call uses an event flag as a parameter, and since the number of valid values for this parameter is small (see Sections 4.1.1 and 5.1.1), it may be reasonable at this point for each stub to call a simple routine which checks to ensure that the event flag specified is both valid and not currently assigned. In the Program Design Language specifications for the various VTS/OS routines given in Chapter 6, the assumption has been made that this check is indeed performed by the stubs, and that the completion code ERR-FLAG is returned to the calling task directly by the stub if this error check is not passed.

3.3 DETAILED DESCRIPTION

3.3.1 The Executive

The system Executive for VTS/OS consists of three major nested loops. The outermost of these three loops is a simple infinite loop, which never terminates. Thus, the Executive is never KILLed. The next loop, referred to as the task loop, is driven by the Ready queue. This queue consists of the TCBs of all tasks which are ready to execute once they are given control of the CPU. During normal processing, there will be periods when no tasks are on the Ready queue. This will be the case when all tasks are awaiting interrupt-driven events, such as device input and output completion. When this condition occurs, the task WHILE-loop of the Executive will terminate, and system diagnostic or test routines may be called. In this manner, self-diagnostic routines will not be performed during peak load situations.

The Executive uses several subordinate routines which are shown in Figure 3-1 and described briefly here. Given a task awaiting processing on the Ready queue, upon entry to the task loop, the Exec first checks to see whether or not LERCUPDATE and MERCUPDATE must be called. These two routines, which are described in detail in Section 5.6, essentially monitor the current health of the local processor and the system. These routines are called on a cyclic basis after every "maxcycle" entries into the task loop. If an error is detected by LERCUPDATE during processing, the error is reported and verified by the LERC. If a LERC or a main processor fails to send an "I AM OPERATIONAL" message to a MERC, MERCUPDATE disconnects that processor or a MERC reconfigures the system.

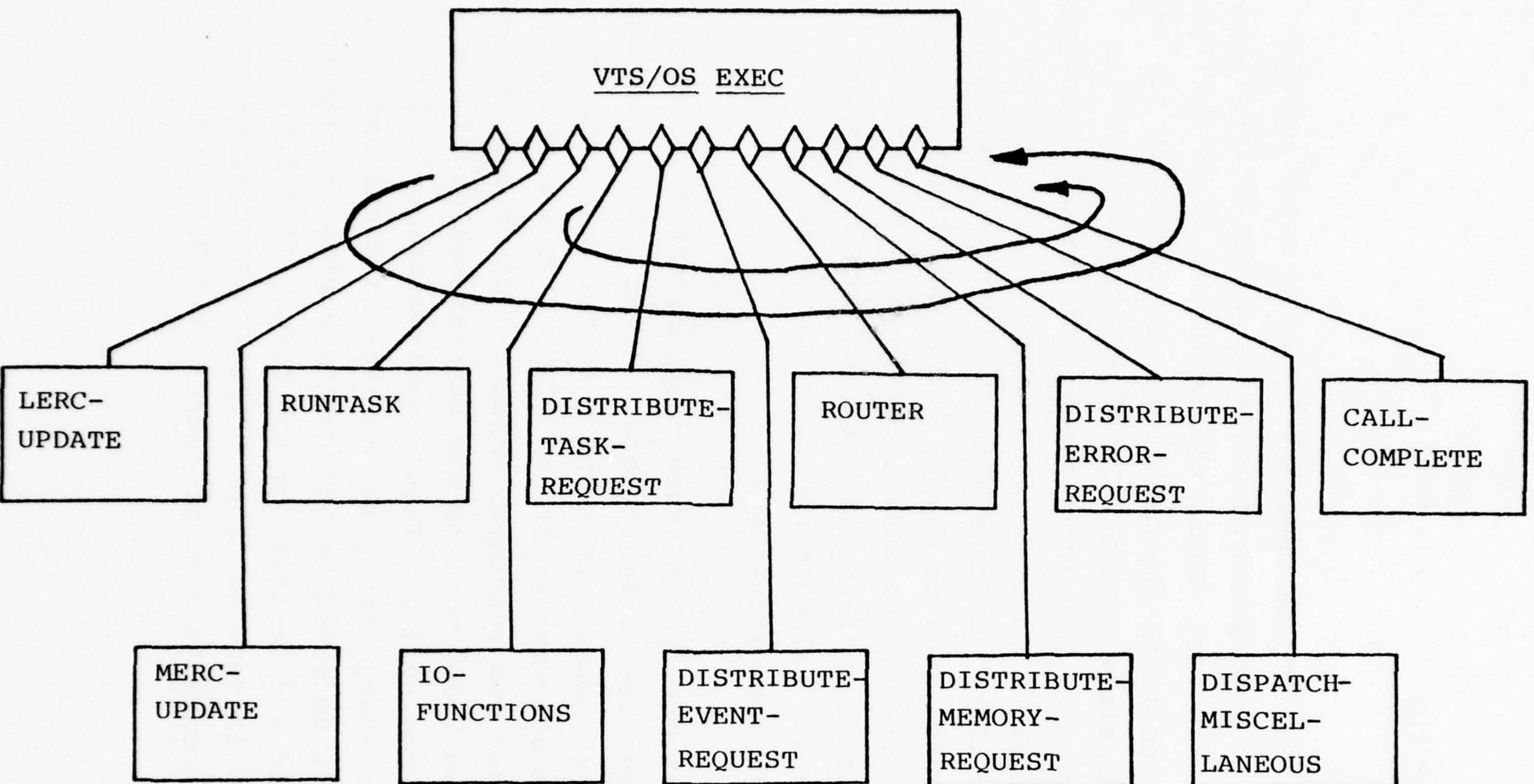


Figure 3-1. VTS/OS Executive Structure

After calling these two routines, if necessary, the Exec calls routine RUNTASK, passing as a parameter the address of the top Task Control Block on the Ready queue. The internal operations of this routine cannot be specified in any detail at this time since they are entirely hardware dependent. Routine RUNTASK does perform whatever context switching is necessary to pass control of the CPU to the task specified. The task then assumes control of the CPU and executes until it makes a system service request. At this time, control of the CPU returns to the Exec, which now enters the innermost loop, the ECB loop.

The vehicle used for signalling all task requests to the Executive is the Event Control Block (ECB), which is described in Section 3.3.3.4. A fixed header is present in each ECB, containing general linkage information and function parameters describing the nature of the service requested by the task. The Exec removes the top ECB from the active ECB queue and uses the function code to pass the ECB to a distributor routine (i.e., IOFUNCTIONS, DISTRIBUTETASKREQUEST, DISTRIBUTEEVENTREQUEST, ROUTER, DISTRIBUTEMEMORYREQUEST, DISTRIBUTEERRORREQUEST, and DISTRIBUTEMISCELLANEOUS) for a logically associated set of functions which will identify the appropriate routine to service the request made by the task by examining the subfunction parameter within the ECB.

Upon return of control to the Executive from the called distributor, the contents of the ECB may have been altered by the system function, in which case, the Executive will call the appropriate distributor to perform the newly-specified function. If the ECB was not altered by the system function, it will not be on the active ECB queue, and the Executive will call the CALLCOMPLETE routine to update the status of the call in the task's Task Control Block. The Executive will then return to the top of the task loop. If the previous task did not specify a WAIT operation, a PAUSE operation, or any operation with an event flag value of 'SUSPEND', that task will still be at the head of the Ready queue, and will be given control by the Executive.

3.3.2 Executive Level Service Routines

Some functions must be performed by many or all of the functional processing routines called by the Executive. Two major classes of service must be provided at this level: queue management for the synchronous and asynchronous portions of system calls, to allow passing of ECB's for processing; and event completion processing, to update the system data structures to reflect the completion of a requested service.

3.3.2.1 Queue Management

The routines cited here are provided for manipulation of system queues. Since they represent functions which must be used frequently by all VTS/OS system calls, they may be implemented in assembly language to optimize VTS/OS response time and efficiency. These routines are described only in terms of their capabilities. PDL descriptions are provided for most of the routines, but give only a typical implementation of the algorithm.

The standard form of a VTS/OS queue is shown in Figure 3-2. There is a Queue Control Block, containing information on the definition and properties of the queue. This information includes a QHEAD which has pointers to the first and last nodes of the queue. Each node in the queue is linked to its successor and predecessor nodes by forward and backward links, referred to as FLINK and BLINK. The forward link of the node at the tail of the queue is linked to NIL, as is the backward link of the node at the head of the queue.

The two main queue manipulation routines are LINK and DELINK. LINK is used to add a node to a specified queue. The specified node will be added at the end of the queue. DELINK is used to remove a node from a specified queue. The specified node may be removed from anywhere in the queue. A third routine, ENQBEFORE, may be used to add a new node at a specified position in a queue.

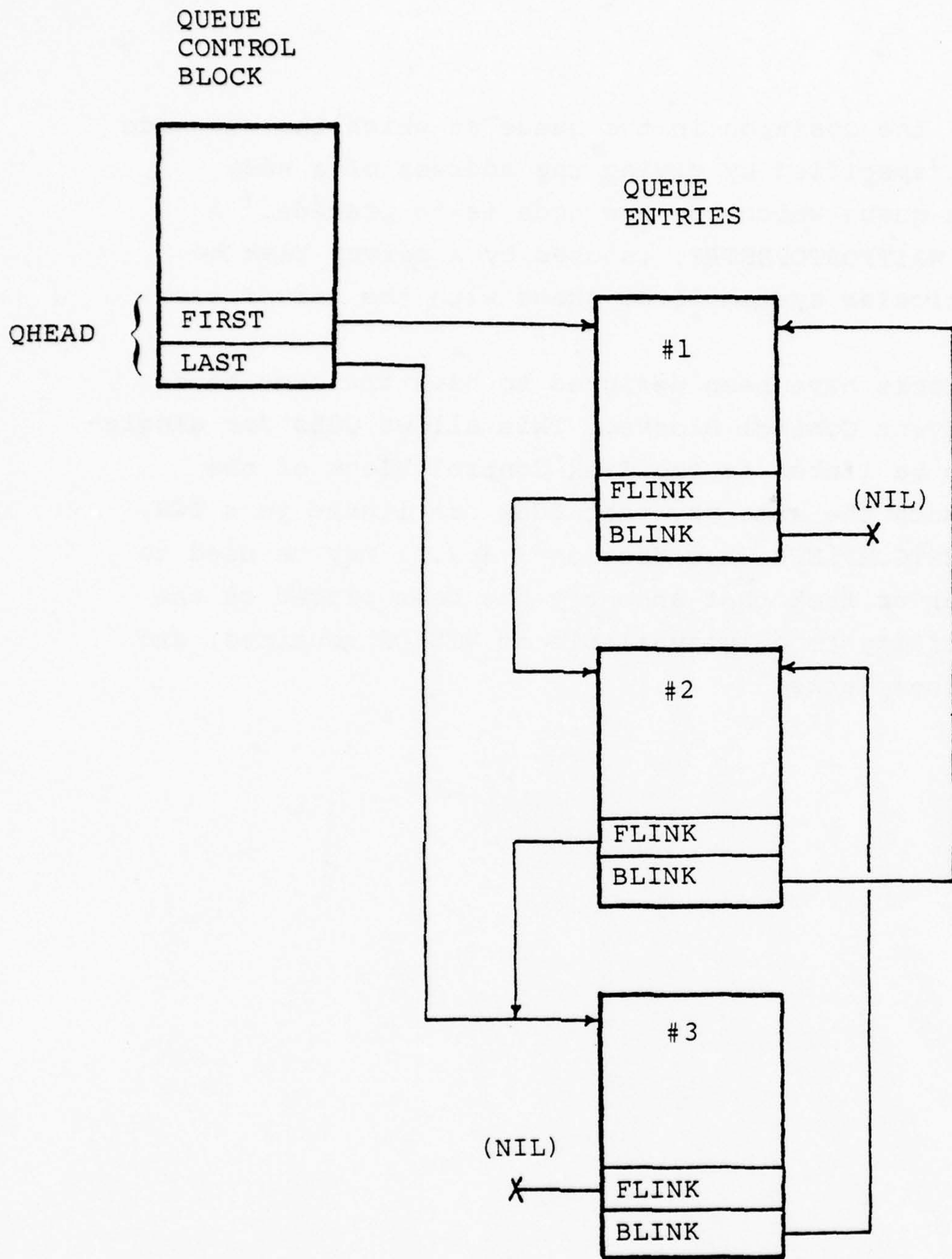


Figure 3-2 VTS/OS Queue Structure

In this routine, the position in the queue at which the new node will be added is specified by giving the address of a node currently in the queue which the new node is to precede. A fourth routine, WAITFORTOPENTRY, is used by a server task to associate a particular system-level queue with the server task.

Queue Control Blocks have been designed to have the same header information as Event Control Blocks. This allows QCBs for single-server queues to be linked to the Task Control Block of the server task in much the same way that ECBs are linked to a TCB. Furthermore, EVENTCOMPLETE (see Section 3.3.2.2) may be used to signal to the server task that an entry has been placed on the queue. This facility is only available to VTS/OS routines, and not to applications tasks.

```
PROCEDURE LINK( NODE, QHEAD, QLINK )

/*  INPUT:   NODE - ADDR OF A NODE TO BE PLACED ON Q
             QHEAD - ADDR OF 2-WORD QHEAD
             QLINK - SPECIFIER OF 2-WORD LINK FIELD
    OUTPUT:  NODE WILL BE ADDED TO END OF SPECIFIED
             QUEUE */


       BEGIN
          IF QHEAD^.LAST <> NIL
             THEN BEGIN
                OLDLAST := QHEAD^.LAST;
                OLDLAST^.QLINK^.FLINK := NODE;
                NODE^.QLINK^.BLINK := OLDLAST;
                NODE^.QLINK^.FLINK := NIL;
                QHEAD^.LAST := NODE;
             END
             ELSE BEGIN
                QHEAD^.FIRST := NODE;
                QHEAD^.LAST := NODE;
                NODE^.QLINK^.FLINK := NIL;
                NODE^.QLINK^.BLINK := NIL;
             END;
       END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE DELINK ( NODE, QHEAD, QLINK )

/*  INPUT:   NODE - ADDR OF A NODE TO BE REMOVED FROM Q
             QHEAD - ADDR OF 2-WORD QHEAD
             QLINK - SPECIFIER OF 2-WORD LINK FIELD
    OUTPUT:  NODE WILL BE REMOVED FROM SPECIFIED QUEUE

    NOTE:    1)  OPERATION NOT DEFINED IF "NODE" IS NIL
             2) THIS CAN BE USED TO ACCESS THE
             VARIOUS FREE-NODE QUEUES BY:

             IF QHEAD^.FIRST <> NIL
                THEN DELINK(QHEAD^.FIRST, QHEAD,
                   QHEAD^.FIRST^.QLINK)
                ELSE (NONE AVAIL) ERR-NONEAVAILABLE;

             OR ANY CODE THAT IS SIMILAR. */


       BEGIN
          IF NODE^.QLINK^.FLINK <> NIL /* IS NOT LAST */
             THEN BEGIN /* TAKE CARE OF SUCCESSOR */
                SUCC := NODE^.QLINK^.FLINK;
                SUCC^.QLINK^.BLINK := NODE^.QLINK^.BLINK;
             END
             ELSE /* IS LAST -- FIX QHEAD */
```

```
         QHEAD^.LAST := NODE^.QLINK^.BLINK;
   IF NODE^.QLINK^.BLINK <> NIL /* IS NOT FIRST */
      THEN BEGIN /* TAKE CARE OF PREDECESSOR */
         PRED := NODE^.QLINK^.BLINK;
         PRED^.QLINK^.FLINK := NODE^.QLINK^.FLINK;
      END
      ELSE /* IS FIRST -- FIX QHEAD */
         QHEAD^.FIRST := NODE^.QLINK^.FLINK;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDQ

```
PROCEDURE WAITFORQOPENTRY ( EVENTFLAG, QHEAD, STATUS )

/*  FUNCTION:   THIS PROCEDURE IS USED BY A SERVER TASK
                TO ASSOCIATE ITSELF WITH A PARTICULAR SINGLE-
                SERVER QUEUE

    INPUT:      EVENTFLAG - FLAG BY WHICH SERVER TASK SHALL
                BE NOTIFIED WHEN AN ENTRY IS PLACED ON THE QUEUE
                QHEAD - SPECIFIES THE QUEUE OF INTEREST - QUEUE
                MUST HAVE A DEFINED QUEUE CONTROL BLOCK
    OUTPUT:     STATUS - RETURNED TO CALLING TASK - ONE OF
                NO-ERROR
                ERR-QCLAIMED -- ANOTHER TASK IS SERVING THIS QUEUE

    NOTE:       HOW THIS WILL BE PERFORMED IS NOT AT THIS
                TIME COMPLETELY SPECIFIED.  IT IS PARTIALLY
                DEPENDENT UPON THE PASCAL IMPLEMENTATION
                USED, AND MAY WIND UP BEING DONE IN MACHINE
                LANGUAGE FOR EFFICIENCY.  HENCE, NO CODE IS
                GIVEN HERE.  THE SERVING TASK WILL BE NOTIFIED
                BY THE TASK PLACING AN ENTRY ON THE QUEUE, WHO
                WILL USE THE LINK ROUTINE DEFINED ABOVE TO ADD
                THE NEW ENTRY, AND WILL THEN CALL EVENTCOMPLETE,
                PASSING THE QCB AS THE PARAMETER.  THE SERVING TASK
                MAY THEN USE DELINK TO REMOVE THE TOP NODE. */

PROCEDURE ENQBEFORE ( QHEAD, NEWNODE, OLDNODE, QLINK )

/*  FUNCTION:   THIS PROCEDURE MAY BE USED TO INSERT A NODE ON A QUEUE

    INPUT:      QHEAD - ADDRESS OF QHEAD
                NEWNODE - NODE TO ADD TO Q -- WILL GO BEFORE:
                OLDNODE - IF NOT NIL, NEWNODE WILL PRECEDE IT --
                   IF NIL, NEWNODE WILL BE LAST NODE ON Q
                QLINK - LINK FIELD IN NODE
    OUTPUT:     NEWNODE IS LINKED TO QUEUE */

  BEGIN
    IF OLDNODE = NIL /* THEN NEWNODE GOES ON END */
      THEN LINK(NEWNODE, QHEAD, QLINK) /* SO USE LINK */
      ELSE BEGIN /* NEWNODE GOES IN MIDDLE */
         NEWNODE^.QLINK^.FLINK := OLDNODE;
         NEWNODE^.QLINK^.BLINK := OLDNODE^.QLINK^.BLINK;
         OLDNODE^.QLINK^.BLINK := NEWNODE;
         IF NEWNODE^.QLINK^.BLINK = NIL /* IS A NEW FIRST */
            THEN QHEAD^.FIRST := NEWNODE;
      END;
  END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

3.3.2.2 Event Completion

There are two subroutines for processing event completion. The first, CALLCOMPLETE, accepts as input the ECB passed to the Executive for task request processing, and updates the status of that request. The second, EVENTCOMPLETE, is used to signal a task that one of its service requests has been completed. If the event which has completed allows that task to be activated, EVENTCOMPLETE will place the task's TCB on the Ready queue. EVENTCOMPLETE is called by the asynchronous portions of the system service requests when their processing is finished.

CALLCOMPLETE will return to the user any errors detected in processing the requested task without allocating a new Event Control Block to that task. If the event flag specified is one of EFLAG1..EFLAG15, the user task will remain at the head of the Ready queue; if the event flag is SUSPEND, the user task will be suspended if the operation is not complete, or will be placed at the tail of the Ready queue if the operation is complete or an error has occurred.

3.3.3. DATA STRUCTURES

THE FOLLOWING SECTION DESCRIBES DATA TYPES USED IN VTS/OS TO CONTROL TASKS, EVENTS, I/O OPERATIONS, AND OTHER SYSTEM ACTIVITIES REQUIRING SYNCHRONIZATION BY THE OPERATING SYSTEM. WHEREVER POSSIBLE, DATA STRUCTURES HAVE BEEN TIGHTLY LINKED TO EACH OTHER IN ORDER TO AID IN THE DETECTION OF SYSTEM FAILURES AND TO AID IN THE RECOVERY FROM THESE FAILURES.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

3.3.3.1 TYPE DECLARATIONS

THE FOLLOWING TYPE DECLARATIONS ARE USED IN ALL SYSTEM DATA STRUCTURES. ADDITIONAL TYPES MAY BE NECESSARY OR CONVENIENT.

```
    TYPE DISCADDR= RECORD
                       SURFACE,TRACK,SECTOR: INTEGER
                   END;

    TYPE DATE= RECORD
                   MONTH: 1..12;
                   DAY:   1..31;
                   YEAR:  INTEGER
               END;

    TYPE TIME=  RECORD
                   HOURS: 0..23;
                   MINS:  0..59;
                   SECS:  0..59
                END;

    TYPE STATUSTYPE = RECORD
                          ERROR: INTEGER;
                          VALUE1: INTEGER;
                          VALUE2: INTEGER
                      END;
    TYPE QHEAD = RECORD
               FIRST: POINTER; /* FIRST NODE OF Q */
               LAST: POINTER; /* AND LAST NODE */
         END;

    TYPE QLINK = RECORD
               FLINK: POINTER; /* NEXT NODE OF Q */
               BLINK: POINTER; /* AND PREVIOUS NODE */
         END;

    TYPE FILENAME = RECORD
                  DEVICE: PACKED ARRAY [1..4] OF CHAR;
                  NAME: PACKED ARRAY [1..10] OF CHAR;
                  EXTENSION: PACKED ARRAY [1..2] OF CHAR;
         END;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

3.3.3.2 IMAGE CONTROL BLOCK

THE IMAGE CONTROL BLOCK (ICB) IS USED TO MAINTAIN THE BASIC INVENTORY OF EXISTING PROGRAMS IN EACH PROCESSOR. WHEN A PROGRAM IS INSTALLED IN A PROCESSOR, AN ICB, CONTAINING ALL FIXED INFORMATION ABOUT THE PROGRAM, IS CREATED. ICB'S ARE DESTROYED BY THE 'REMOVE' CALL.

ALL ICB'S ARE ON ONE OF TWO QUEUES. THOSE IN USE ARE ON A QUEUE CALLED THE 'ICBQUEUE', WHILE THOSE NOT IN USE ARE ON A QUEUE CALLED THE 'ICBAVAILQ'.

```
ICB=  RECORD
          ICBQLINK: QLINK; /* ICB IN-USE AND AVAIL Q'S */
          NODETYPE: 1..MAXNODETYPES; /* IDENTIFIES ICB NODE */
          TASKNAME: PACKED ARRAY [1..N] OF CHAR;
          USECOUNT: INTEGER;
          PROCESSNAME: INTEGER; /* USED IN IPC */
          OCBQHD: QHEAD; /* OCB'S OF PROGRAM */
          REMOVEECBPTR: POINTER; /* ECB OF REMOVE IN PROGRESS */
          MAPSFORROOT:  INTEGER; /* THESE FOUR FIELDS CONTAIN */
          MAPSFOROLAY: INTEGER; /* THE NUMBER OF MAP REGISTERS */
          MAPSFORVAR: INTEGER;  /* REQUIRED FOR THE ROOT CODE, */
          MAPSFORBUFF: INTEGER; /* OVERLAYS, VARIABLES, & BUFFERS */
          TASKMAP: ARRAY [1..MAXMAPREGISTERS] OF INTEGER; /*ROOT*/
          TCBQHD: QHEAD; /*  TCB'S OF THIS PROGRAM*/
          INITIALCONTEXT: ARRAY [1..CONTEXTSIZE] OF INTEGER;
      END;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

3.3.3.3 TASK CONTROL BLOCK

THE TASK CONTROL BLOCK (TCB) IS USED TO MAINTAIN THE INVENTORY OF ACTIVE TASKS IN EACH PROCESSOR. WHEN A TASK IS SCHEDULED, A TCB, CONTAINING ALL THE VARIABLE INFORMATION FOR THE TASK, IS CREATED. THIS TCB EXISTS UNTIL THE TASK IS KILLED, EITHER BY ITSELF OR BY ANOTHER TASK.

ALL TCB'S ARE ON ONE OF THREE QUEUES, AS FOLLOWS:

TCBAVAILQ - ALL TCB'S NOT IN USE ARE ON THIS Q

RDYQHD - TCB'S OF TASKS THAT ARE READY TO USE THE CPU, SHOULD THEY GET IT

SUSPENDQHD - TCB'S OF TASKS THAT ARE CURRENTLY SUSPENDED OR WAITING FOR EVENTS TO OCCUR.

```
TCB= RECORD
        TCBQLINK: QLINK; /* AVAIL, RDY, SUSP, & OTHER Q'S */
        NODETYPE: 1..MAXNODETYPES; /* IDENTIFIES TCB NODE */
        ICBPTR: POINTER;
        CLONEQLINK: QLINK; /* TASKS OFF SAME ICB */
        BROTHERQLINK: QLINK; /* BROTHER TCB-Q */
        CHILDQHD: QHEAD; /* CHILD TCB-Q HEAD */
        PARENT: POINTER;
        POSTECB: POINTER;
        EFLAGSACTUAL: INTEGER;
        EFLAGSPOSSIBLE: INTEGER;
        WAITFLAGS: INTEGER;
        WAITTYPE: INTEGER; /* WAITALL, WAITANY, SUSPEND, NIL */
        OCCURREDEVENTPTR: POINTER; /* FOR WAITALL,WAITANY */
        EFLAGPTR: ARRAY [1..16] OF POINTER;
        KILLECBPTR: POINTER; /* ECB ADDR IF TASK BEING KILLED*/
        TASKID: INTEGER;
        BCBQHD: QHEAD; /* BUFFER CONTROL BLOCK-Q HEAD*/
        LASTBUFFERID: INTEGER;
        LASTSYSTEMCALL: RECORD
                           FUNCTION: 1..MAXFUNCCODE;
                           SUBFUNCTION: INTEGER;
                           STATUS: STATUSTYPE;
                        END;
        TASKCONTEXT: ARRAY [1..CONTEXTSIZE] OF INTEGER;
        TASKMAP: ARRAY [1..MAXMAPREGISTERS] OF INTEGER
     END;
```

3.3.3.4 EVENT CONTROL BLOCK

THE EVENT CONTROL BLOCK (ECB) IS USED TO MAINTAIN THE STATUS OF OUTSTANDING SERVICE REQUESTS. AN ECB IS CREATED WHEN A TASK FIRST ISSUES A SERVICE REQUEST, AND IS NOT DESTROYED UNTIL THE REQUEST HAS TERMINATED AND THE TASK HAS PERFORMED A GETSTATUS REQUEST TO OBTAIN THE STATUS OF THE INITIAL REQUEST.

```
ECB=  RECORD
          ECBQLINK: QLINK;/* AVAIL, IN-USE, ETC Q'S */
          NODETYPE: 1..MAXNODETYPES; /* IDENTIFIES ECB NODE */
          STATUS: STATUSTYPE;
          TCBPTR: POINTER;
          EFLAG: INDEPENDENT, EFLAG1..EFLAG15, SUSPEND, O;
          FUNCTION: 1..MAXFUNCCODE;
          SUBFUNC: INTEGER;
          STATUSPTR: POINTER; /* INTO CALLER'S ADDRESS SPACE */
          STORAGE: ARRAY [1..N] OF INTEGER; /*PARAMETERS AND TEMPS*/
          /* NOTE PARAMETERS ARE OFTEN REFERENCED AS ECB^.PARAM1,
             ECB^.PARAM2, ETC. */
      END;
```

3.3.3.5 BUFFER CONTROL BLOCK

BUFFER CONTROL BLOCKS ARE USED TO CONTROL THE ALLOCATION OF SYSTEM MEMORY TO RUNNING TASKS. A CHAIN OF BCB'S HANGS OFF OF EACH TCB, INDICATING WHICH PAGES OF MEMORY HAVE BEEN ASSIGNED TO THIS TASK. THE BCB WITH BUFFERID OF 0 IS FOR THE TASK'S VARIABLE SPACE. ALL OTHERS ARE FOR MEMORY THAT THE TASK HAS EITHER REQUESTED DIRECTLY WITH A GETBUFFER CALL, OR INDIRECTLY, WITH AN EQUATEBUFFER CALL.

BCB'S THAT ARE IN USE HANG OFF THE BCBQHD ENTRY IN EACH TCB. THOSE THAT ARE FREE ARE ON THE BCBAVAILQ.

```
BCB=  RECORD
          BCBQLINK: QLINK; /* AVAIL AND EACH-TASK Q'S */
          NODETYPE: 1..MAXNODETYPES; /* IDENTIFIES BCB NODE */
          BUFFERID: INTEGER; /*IF 0, IS VARIABLE SPACE
               ALLOCATION, AND NOT UNDER TASK CONTROL*/
          USECOUNT: INTEGER;
          IOINPROGRESS: INTEGER; /* NUMBER OF XFERS */
          SHARE: BOOLEAN;
          RELEASEECBPOINTER: POINTER;    /*ECB OF RELEASE CALL*/
          OWNERBCBPOINTER: POINTER; /* THIS ALSO SERVES
             AS AN INDICATOR OF WHETHER OR NOT THIS TASK 'OWNS'
             THE MEMORY.  IF IT DOES, THIS FIELD IS NIL; IF NOT
             THIS FIELD POINTS TO THE OWNER'S BCB. */
          OWNERTCBPOINTER: POINTER; /*ALWAYS TCB OF OWNER */
          NUMBEROFMAPREGS: INTEGER;      /*PAGES IN BUFFER*/
          BEGINNINGREG: INTEGER;         /*LOGICAL ADDRESS START*/
          MAPREGS: ARRAY [1..MAXMAPREGISTERS] OF INTEGER;
      END;
```

NOTE:

IF OWNERBCBPTR IS NOT NIL, THEN USECOUNT AND SHARE ARE IRRELEVANT. FURTHER, IOINPROGRESS IS A COUNT ONLY OF I/O XFERS BY THIS TASK. IF OWNERBCBPTR IS TRUE, THEN IOINPROGRESS IS A COUNT OF XFERS IN PROGRESS FOR THIS BUFFER BY ALL TASKS ACCESSING THIS BUFFER. THE NUMBER OF SUCH TASKS IS IN USECOUNT.

3.3.3.6 FILE DESCRIPTOR BLOCK

```
/*THE FILE DESCRIPTOR BLOCK IS USED BY THE I/O ROUTINES
  TO CONTAIN INFORMATION ON THE PROGRESS OF I/O REQUESTS
  AND TO MAKE ACTION REQUESTS TO DEVICE DRIVERS.  FDB'S
  ARE LINKED TO THE DDB FOR THE DEVICE INVOLVED, AND THE
  FDB POINTER IS PLACED IN THE CHANNEL TABLE FOR THE
  OPENED CHANNEL INVOLVING THAT FILE OR DEVICE.*/

FDB= RECORD
   FDBQLINK: QLINK;        /*LINK FOR DDB FDB QUEUE*/
   DDBPOINTER: POINTER;     /*POINTER TO DDB*/
   CHANNEL: 1..MAXCHANNEL;     /*CHANNEL FOR THIS FDB*/
   ECBPTR: POINTER;        /*USER REQUEST ECB*/
   /*INFORMATION CONTAINED IN THE DISC DIRECTORY*/
   DISCDIRECTORYENTRY = RECORD
      FILENAME: FILENAMETYPE;
      ATTRIBUTES= RECORD
                  READPROTECT: BOOLEAN;
                  WRITEPROTECT: BOOLEAN;
                  PROTECTED: BOOLEAN;
                  DIRECTORY: BOOLEAN;
                  BLOCKED: BOOLEAN;
                  SEGMENTED: BOOLEAN;
                  CONTIGUOUS: BOOLEAN;
                  SEQUENTIAL: BOOLEAN;
                  AVAILABLE: BOOLEAN
                  END;
      STARTADDR: DISCADDR;
      FILESIZE: INTEGER;
      ENDADDR: DISCADDR;
      DATASEGMENTSIZE: INTEGER;
      CRDATE: DATE;
      CRTIME: TIME;
      END;
   USERMEMORYSPEC = RECORD     /*OBTAINED AT TIME OF CALL*/
                    BUFADDR: POINTER;
                    BUFFSIZE: INTEGER;
                    NUMBEROFMAPREGISTERS: INTEGER;
                    MAPREGISTERS: ARRAY [1..MAXMAPREGISTERS] OF
                       INTEGER;
                    END;
   SEQPOSITION = RECORD   /*POSITION FOR SEQUENTIAL I/O*/
                 BLOCK: DISCADDR;       /*CURRENT DISC BLOCK*/
                 WORD: INTEGER       /*WORD WITHIN BLOCK*/
                 END;
   WORDSTRANSFERRED: INTEGER;
   BUFFER: POINTER;  /*TO BUFFER BCB*/
   CURRENTSPBBUFFER: POINTER;
   DEVICEDRIVERREQUEST: INTEGER;
   CURRENTDISCSECTOR: DISCADDR;   /*USED BY DRIVER*/
   WORDSWRITTEN: INTEGER;         /*FOR SPLIT READ/WRITE*/
   DRIVERRETURNQCB: POINTER;      /*FOR NOTIFICATION OF DRIVER
                                    REQUEST COMPLETION*/
   END;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

3.3.3.7 DEVICE DESCRIPTOR BLOCK

```
/*THE DEVICE DESCRIPTOR BLOCK CONTAINS THE ATTRIBUTES
  AND PARAMETERS DEFINING USER ACCESS TO AND DEVICE
  DRIVER OPERATIONS FOR THE DESIGNATED DEVICE.  ARRAY
  OPERATION CONTAINS NIL FOR ALL OPERATIONS NOT
  ALLOWED FOR THIS DEVICE.*/

DDB=  RECORD
      DEVICENAME: PACKED ARRAY [1..4] OF CHAR;
      DDBQLINK: QLINK;
      FDBQHEAD: QCB;  /*QUEUE OF OPEN FDB'S*/
      ATTRIBUTES= RECORD
                  READPROTECT: BOOLEAN;
                  WRITEPROTECT: BOOLEAN;
                  PROTECTED: BOOLEAN;
                  DIRECTORY: BOOLEAN;
                  BLOCKED: BOOLEAN;
                  SEGMENTED: BOOLEAN;
                  CONTIGUOUS: BOOLEAN;
                  SEQUENTIAL: BOOLEAN;
                  AVAILABLE: BOOLEAN
                  END;
      OPERATION: ARRAY [1..MAXIOSUBFUNC] OF
                     CREA, DEL, REN, GETATR,
                     SETATR, OPN, CLS, RDR,
                     WRR, RDSQ, WRSQ, RDLN,
                     WRLN, IORST;
      BLOCKSIZE: INTEGER;
      ADDRESS: INTEGER;
      TIMEOUT: INTEGER;
      RETRYCOUNT: 1..10
      END;
```

3.3.3.8 SEGMENT POINTER BLOCK

```
/*THE SEGMENT POINTER BLOCK IS A DISC-BASED DATA
  STRUCTURE CONTAINING POINTERS TO THE DATA
  SEGMENTS OF SEGMENTED FILES. */

SPB=  RECORD
          LASTSPB: DISCADDR;
          NEXTSPB: DISCADDR;
          DATASEGPTR: ARRAY [1..MAXDSP] OF DISCADDR
      END;
```

3.3.3.9 OVERLAY CONTROL BLOCK

OVERLAY CONTROL BLOCKS ARE USED TO CONTROL THE ALLOCATION OF SYSTEM MEMORY TO PROGRAM OVERLAY CODE. EACH OCB SPECIFIES, BY THE CONTENTS OF ITS MAP REGISTERS, THE PAGES OF PHYSICAL MEMORY CONTAINING ONE OVERLAY. SINCE NOT ALL PROGRAMS USE OVERLAYS, OCB'S ARE NOT USED TO CONTROL THE ALLOCATION OF SYSTEM MEMORY TO PROGRAM ROOT CODE. THIS MEMORY IS CONTROLLED THROUGH THE MAPREGISTERS FIELD IN THE IMAGE CONTROL BLOCK.

FREE OCB'S ARE ON THE OCBAVAILQ. THOSE THAT ARE IN USE ARE ON THE OCB-QUEUES THAT HANG OFF OF THE VARIOUS ICB'S, WITH A QUEUE HEADER IN EACH ICB BY THE NAME OF 'OCBQHD'.

```
OCB= RECORD

        OCBQLINK: QLINK; /* AVAIL AND EACH-PROGRAM Q'S */
        NODETYPE: 1..MAXNODETYPES; /* IDENTIFIES OCB NODE */
        ICBPTR: POINTER; /* POINTER TO ICB */
        OVERLAYID: INTEGER; /* UNIQUE IDENTIFIER OF OVERLAY */
        ICBPTR: POINTER; /* ICB OF THIS OCB */
        BEGINNINGREG: INTEGER; /* FIRST MAP REGISTER FOR THIS
                              OVERLAY */
        NUMBEROFMAPREGS: INTEGER; /* NUMBER OF REGS */
        MAPREGS: ARRAY [1..MAXMAPREGISTERS] OF INTEGER;
   END;
```

3.3.3.10 QUEUE HEADS AND QUEUE CONTROL BLOCKS

THIS SECTION CONTAINS A COMPLETE LISTING OF ALL SYSTEM-LEVEL QUEUES, I.E., THOSE QUEUES WHOSE QHEADS ARE SYSTEM-LEVEL VARIABLES. THOSE QUEUES WHOSE QHEADS ARE IMBEDDED IN RECORDS(E.G., THE OCBQHD WHICH IS IN EACH ICB) ARE NOT LISTED HERE.

```
ICBAVAILQHD -- QHEAD FOR AVAILABLE ICB'S
TCBAVAILQHD -- QHEAD FOR AVAILABLE TCB'S
OCBAVAILQHD -- QHEAD FOR AVAILABLE OCB'S
BCBAVAILQHD -- QHEAD FOR AVAILABLE BCB'S
ECBAVAILQHD -- QHEAD FOR AVAILABLE ECB'S
ICBQHD      -- QHEAD FOR ICB'S IN USE
RDYQHD      -- QHEAD FOR TCB'S WHOSE TASKS ARE READY TO USE
               THE CPU.  (OBVIOUSLY, THESE TCB'S ARE IN USE.)
SUSPENDQHD  -- QHEAD FOR TCB'S WHOSE TASKS ARE NOT READY TO
               USE THE CPU.  (OBVIOUSLY ALSO IN USE)
TIMEOUTQHD  -- QHEAD OF ECB'S WAITING FOR TIMEOUTS; ORDERED, WITH
               INTER-ECB INTERVAL IN SECONDS IN ECB^.PARAM1
REALTIMECLOCKQCB -- QCB FOR ECB'S QUEUED BY THE REAL-TIME
               CLOCK HANDLER TO THE TIMEOUTASYNC ROUTINE.
               THERE WILL USUALLY BE ONLY ONE ECB ON THIS QUEUE,
               SINCE THE HANDLER WILL QUEUE ONE PER SECOND
INSTALLQCB  -- QCB FOR ECB'S QUEUED UP BY THE SYNCHRONOUS
               INSTALL ROUTINE FOR THE ASYNCHRONOUS INSTALL
ECBQHD      -- QHEAD FOR ECB'S QUEUED UP BY THE USER INTERFACE
               STUBS FOR THE EXEC
```

3.3.3.11 QUEUE CONTROL BLOCK

THIS BLOCK IS USED TO CONTROL SINGLE-SERVER SYSTEM-LEVEL QUEUES. IT HAS ALMOST THE SAME FORMAT AS THE EVENT CONTROL BLOCK, AND CAN THEREFORE BE PASSED TO EVENTCOMPLETE TO SIGNAL THE SERVER TASK THAT A NODE HAS BEEN PLACED ON THE QUEUE. THIS IS USUALLY DONE BY THE SYNCHRONOUS PORTION OF A SYSTEM REQUEST PROCESSOR TO ACTIVATE THE ASYNCHRONOUS PORTION.

```
QCB = RECORD
   HEAD:  QHEAD; /* FIRST & LAST NODES */
   NODETYPE:  1..MAXNODETYPES; /* IDENTIFIES QCB NODE */
   STATUS:  STATUSTYPE; /* FOR COMPATIBILITY */
   TCBPTR:  POINTER; /* TO TCB OF SERVER TASK */
   EFLAG:  EFLAG1..EFLAG15, SUSPEND, NIL;
   MAXNODES: INTEGER; /* EVER ON QUEUE */
   MINNODES: INTEGER; /* EVER ON QUEUE */
   STATUSPTR:  POINTER; /* TO STATUS OF SERVER TASK */
   STORAGE:  ARRAY [1..N] OF INTEGER; /* WORKING STORAGE */
END;
```

3.4 EXECUTIVE PDL SPECIFICATIONS

3.4.1 EXECUTIVE

```
PROGRAM EXEC;

        /*THE EXEC PERFORMS SERVICES, BUT HAS NO INPUTS OR OUTPUTS
          OTHER THAN EVENT CONTROL BLOCKS ASSOCIATED WITH USER
          TASKS*/

BEGIN
    CYCLE := 0;
    WHILE TRUE      /*INFINITE LOOP*/
      DO BEGIN
        WHILE READYQHEAD <> NIL  /*TASK LOOP - ANY TASKS READY?*/
          DO BEGIN
            CYCLE := CYCLE + 1;
            IF CYCLE = MAXCYCLE
              THEN BEGIN
                LERCUPDATE;
                MERCUPDATE;
                CYCLE := 0;
              END;
            ACTIVETASK := READYQHEAD;
            RUNTASK (ACTIVETASK); /*TRANSFER CONTROL TO TASK*/
            WHILE ECBQHEAD <> NIL    /*PROCESS REQUEST ECB */
              DO BEGIN
              ACTIVEECB := ECBQHD^.FIRST;
              DELINK(ACTIVEECB,ECBQHD,ACTIVEECB^.ECBQLINK);
                CASE ACTIVEECB^.FUNCTION OF
                  IO:    IOFUNCTIONS (ACTIVEECB);
                  TASK:  DISTRIBUTETASKREQUEST (ACTIVEECB);
                  EVT:   DISTRIBUTEEVENTREQUEST (ACTIVEECB);
                  ERR:   DISTRIBUTEERRORREQUEST (ACTIVEECB);
                  ITC:   ROUTER (ACTIVEECB);
                  MEM:   DISTRIBUTEMEMORYREQUEST (ACTIVEECB);
                  MISC:  DISPATCHMISCELLANEOUS (ACTIVEECB)
                END;       /*END OF CASE STATEMENT*/
              END;       /*END OF ECB LOOP*/
            /*PROCESS EVENT FLAG AND SUSPEND TASK IF NECESSARY*/
            CALLCOMPLETE (ACTIVEECB)
          END;       /*END OF TASK LOOP*/
      /*POTENTIAL SELF-DIAGNOSTIC ROUTINE*/
    END;       /*END OF INFINITE LOOP*/
END.
```

```
PROCEDURE RUNTASK ( ACTIVETASK )

/*  INPUT:    ACTIVETASK - TCB OF TASK TO BE ALLOWED TO
              EXECUTE NEXT
    OUTPUT:   NONE -- PERFORMS TRANSFER OF CONTROL
    NOTE:     IT IS NOT POSSIBLE TO SPECIFY THIS ROUTINE AT THIS
              TIME AT ANY LEVEL OF DETAIL, AS IT IS ENTIRELY DEP-
              ENDENT ON THE SPECIFIC HARDWARE USED.  IT IS ALSO
              LIKELY THAT THIS ROUTINE WILL HAVE TO BE WRITTEN IN
              THE ASSEMBLY LANGUAGE OF THE SPECIFIC HARDWARE. */
```

```
PROCEDURE IOFUNCTIONS (ECBPTR);

/*INPUT:   POINTER TO CURRENT EVENT CONTROL BLOCK,
           PASSED BY THE EXEC.
  OUTPUT:  NONE.  CALLS PROPER SYNCHRONOUS PROGRAM
           FOR THE REQUESTED SYSTEM FUNCTION--
           IS THE I/O DISTRIBUTOR */

BEGIN
FINDDEVICE (ECBPTR, DDBPOINTER);
IF     /*WAS THE DEVICE FOUND?*/
   DDBPOINTER <> NIL
THEN      /*YES, DO FUNCTION; IF NOT, ERROR IS RETURNED BY EXEC*/
   IF     /*CHECK FOR INVALID FUNCTION*/
      DDBPOINTER^.OPERATION [ECBPTR^.SUBFUNCTION] <> NIL
   THEN      /*VALID FUNCTION, PERFORM SUBFUNCTION*/
      BEGIN
      CASE DDBPOINTER^.OPERATION [ECBPTR^.SUBFUNCTION] OF
         CREA:      CREATESYNC (ECBPTR);
         DEL:       DELETESYNC (ECBPTR);
         REN:       RENAMESYNC (ECBPTR);
         GETATR:    GETATTRSYNC (ECBPTR);
         SETATR:    SETATTRSYNC (ECBPTR);
         OPN:       OPEN (ECBPTR);
         CLS:       CLOSE (ECBPTR);
         IORST:     IORESET (ECBPTR);
         RDR,
         WRR,
         RDSQ,
         WRSQ,
         RDLN,
         WRLN:      READWRITE (ECBPTR);
         END;       /*END OF CASE*/
      END
   ELSE      /*SEND INVALID FUNCTION ERROR TO TASK*/
      ECBPTR^.STATUS.ERROR := ERINVALIDFUNCTION
ELSE      /*INVALID DEVICE*/
   ECBPTR^.STATUS.ERROR := ERDEV;
/*RETURN TO EXEC WITH FUNCTION STATUS*/
END.
```

```
PROCEDURE FINDDEVICE (ECBPTR, DDBPOINTER);

/*FINDDEVICE LOOKS FOR AN EXPLICIT REFERENCE TO A DEVICE, AND
  RETURNS A POINTER TO THE DEVICE DESCRIPTOR BLOCK IF THE
  DEVICE NAME MATCHES A VALID DEVICE SPECIFIER.  IF NO DEVICE
  IS EXPLICITLY REFERENCED, THE SYSTEM DISK DEVICE IS ASSUMED
  BY DEFAULT, AND ITS POINTER IS RETURNED TO THE CALLER.*/

BEGIN
IF     /*IS THIS A CHANNEL REQUEST?*/
   ECBPTR^.SUBFUNC = CREATE
      OR ECBPTR^.SUBFUNC = DELETE
      OR ECBPTR^.SUBFUNC = RENAME
      OR ECBPTR^.SUBFUNC = OPEN
THEN       /*CHANNEL REQUEST, FIND EXPLICIT OR IMPLICIT DEVICE*/
   BEGIN
   IF
      ECBPTR^.FILENAME.DEVICE = NIL
   THEN      /*THE DEFAULT NAME MUST BE SUBSTITUTED*/
      ECBPTR^.FILENAME.DEVICE := "DKO:";
   DDBPOINTER := DDBQHEAD^.FIRST;
   /*LOOK FOR A DEVICE NAME MATCH*/
   WHILE     /*LOOP UNTIL NAME FOUND OR LIST EXHAUSTED*/
      DDBPOINTER <> NIL
         AND DDBPOINTER^.DEVICENAME <> ECBPTR^.FILENAME
   DO
      DDBPOINTER := DDBPOINTER^.DDBQLINK^.FLINK;
   END
ELSE      /*CHANNEL-BASED REQUEST, GET DEVICE FROM FDB*/
   DDBPOINTER := CHANNELTABLE [ECBPTR^.CHANNEL].FDBPTR^.DDBPOINTER;
END.
```

```
PROCEDURE DISTRIBUTETASKREQUEST ( ACTIVEECB )
/*  INPUT:    POINTER TO CURRENT ECB
    OUTPUT:   NONE - ACTIVATES PROPER SYNCHRONOUS
              ROUTINE -- IS TASK MANAGEMENT DISTRIBUTOR */

BEGIN
       CASE ECBPTR^.SUBFUNC OF
           INSTALL:   INSTALLSYNCH ( ACTIVEECB );
           REMOVE:    REMOVE ( ACTIVEECB );
           SCHEDULE:  SCHEDULE ( ACTIVEECB );
           POST:      POST ( ACTIVEECB );
           KILL:      KILL ( ACTIVEECB );
       END;
END.
```

```
PROCEDURE DISTRIBUTEEVENTREQUEST ( ACTIVEECB )
/*  INPUT:    ACTIVEECB, WHICH HAS CURRENT REQUEST
    OUTPUT:   NONE - ACTIVATES PROPER ROUTINE --
              IS THE EVENT MANAGEMENT DISTRIBUTOR */

BEGIN
      CASE ACTIVEECB^.SUBFUNC OF
          SETDELAY:        SETDELAY(ACTIVEECB);
          REMOVEDELAY:     REMOVEDELAY(ACTIVEECB);
          WAITANY:         WAITANY(ACTIVEECB);
          WAITALL:         WAITALL(ACTIVEECB);
          PAUSE:           PAUSE(ACTIVEECB);
          GETSTATUS:       GETSTATUS(ACTIVEECB);
      END;
END.
```

```
PROCEDURE DISTRIBUTEMEMORYREQUEST ( ECBPTR )
/*  INPUT:    POINTER TO CURRENT ECB
    OUTPUT:   NONE - ACTIVATES PROPER SYNCHRONOUS
              ROUTINE -- IS MEMORY MANAGEMENT DISTRIBUTOR */

BEGIN
      CASE ECBPTR^.SUBFUNC OF
         LOADOLAY:  LOADOVERLAY (ECBPTR);
         RLSEOLAY:  RELEASEOVERLAY (ECBPTR);
         GETBUFFR:  GETBUFFER (ECBPTR);
         EQUATBFR:  EQUATEBUFFER (ECBPTR);
         MAPBUFFR:  MAPTOBUFFER (ECBPTR);
         RLSEBUFR:  RELEASEBUFFER (ECBPTR);
      END;
END.
```

```
PROCEDURE DISTRIBUTEERRORREQUEST (ECBPTR);

/*INPUT:   POINTER TO ACTIVE ECB
  OUTPUT:  NONE.  CALLS APPROPRIATE ERROR LOGGING ROUTINE
           IS THE ERROR CONTROL DISTRIBUTOR */

BEGIN
CASE ECBPTR^.SUBFUNC OF
   FAILDETECT: FAILUREDETECTED (ECBPTR);
   END;
END.
```

```
PROCEDURE ROUTER (ECBPTR);

/*THE ROUTER DISPATCHES ITC-RELATED USER FUNCTION CALLS
  TO THE APPROPRIATE SYNCHRONOUS SERVICE ROUTINES.
  THE ROUTER ROUTINES WILL PASS REQUEST ECB'S
  TO THE DISPATCHER AS NECESSARY.  THIS IS THE ITC DISTRIBUTOR.  */

BEGIN
CASE ECBPTR^.SUBFUNC OF
   SENDMS:      SENDMESSAGE (ECBPTR);
   REQUESTMS:   REQUESTMESSAGE (ECBPTR);
   TRANSFERMS:  TRANSFERMESSAGE (ECBPTR);
   SENDANS:     SENDANSWER (ECBPTR);
   END;
END.
```

3.4.2 CALLCOMPLETE

```
PROCEDURE CALLCOMPLETE (ECBPTR)

   /*AN EVENT CONTROL BLOCK IS PASSED TO THIS ROUTINE
     SPECIFYING THE TCB AND EVENT NUMBER OF THE USER
     REQUEST.  IF AN EVENT FLAG OF SUSPEND WAS
     SPECIFIED, THE TASK IS REMOVED FROM THE HEAD OF
     THE READY QUEUE*/

   BEGIN
      /*INITIALIZE LOCAL VARIABLES*/
      TCBPOINTER := ECBPTR^.TCBPTR;
      EVENTFLAG := ECBPTR^.EFLAG;
      EFMASK := MASKS [EVENTFLAG];
      /*SIGNAL THAT EVENT IS EXPECTED*/
      TCBPOINTER^.EFLAGSPOSSIBLE :=
         TCBPOINTER^.EFLAGSPOSSIBLE OR EFMASK;
      IF     /*CHECK FOR ERROR DETECTED*/
         ECBPTR^.STATUS^.ERROR < 0
      THEN BEGIN  /*RETURN ERROR AS STATUS PARAMETER OF CALL*/
         ECBPTR^.STATUSPTR^.STATUS := ECBPTR^.STATUS;
         IF      /*CHECK FOR SYSTEM LEVEL ERROR*/
            /*ERROR IS IN SYSTEM CLASS*/
         THEN
            /*NOTIFY LERC*/;
      END /* RETURNING ERROR  */
      ELSE BEGIN      /*NO ERROR*/
         IF      /*CHECK WHICH EVENT FLAG WAS SPECIFIED*/
            EVENTFLAG = SUSPEND
         THEN      /*SET TCB TO REFLECT THE IMPLIED 'WAIT'*/
            BEGIN
            TCBPOINTER^.WAITFLAGS := EFMASK;
            TCBPOINTER^.WAITTYPE := SUSPEND;
            /*  REMOVE TCB FROM RDYQHD FOR IMPLIED WAIT */
            DELINK (TCB, RDYQHD, TCB^.TCBQLINK);
            IF      /*IF NOT COMPLETE, SUSPEND TASK*/
               ECBPTR^.STATUS^.ERROR = REQUEST-IN-PROGRESS
            THEN   /*PUT TCB ON SUSPENDQ*/
               LINK (TCB, SUSPENDQHD, TCB^.TCBQLINK)
            ELSE BEGIN /* PUT AT END OF READY-Q */
               LINK (TCB, RDYQHD, TCB^.TCBQLINK);
               ECBPTR^.STATUSPTR^.STATUS := ECBPTR^.STATUS;
            END;
         END
         ELSE BEGIN /* IS NOT SUSPEND, SO RETURN STATUS FIRST */
            ECBPTR^.STATUSPTR^.STATUS := ECBPTR^.STATUS;
            /* NOW, IF EFLAG IS ZERO, THE CALL PROCESSING
               IS DONE, SO WE CAN REUSE THIS ECB.  OTHER-
               WISE, WE MUST GET A NEW ECB */
            IF EVENTFLAG <> 0 /* GET A NEW ONE */
            THEN BEGIN /* SAVING THIS ONE AND GET A NEW ONE  */
               TCBPOINTER^.EFLAGPTR (EVENTFLAG) :=
               ECBPTR;
```

```
            GETECB (NEWECBPTR);   /*WAITS IF NONE AVAILABLE*/
         END /* SAVING CURRENT ONE */
         ELSE  /* JUST REUSE THIS ONE */
            NEWECBPTR := ECBPTR;
         TCBPOINTER^.EFLAGPTR (SUSPEND) := NEWECBPTR;
      END;
   END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE GETECB (VAR NEWECBPTR);

/*THIS PROCEDURE IS CALLED BY CALLCOMPLETE AFTER THE SYSTEM
  REQUEST MADE BY THE USER HAS BEEN INITIATED.  A NEW ECB MUST
  BE RETURNED TO THE USER TASK BEFORE ANOTHER SYSTEM REQUEST
  MAY BE ISSUED.  IF A FREE ECB IS NOT AVAILABLE, THE USER
  TASK WILL BE PLACED AT THE TAIL OF THE READY QUEUE, AND
  THE ATTEMPT TO ALLOCATE THE NEW ECB WILL BE MADE WHEN THIS
  USER TASK AGAIN BECOMES READY*/

BEGIN
/*SEE IF AN ECB IS AVAILABLE*/
WHILE
   ECBAVAILQHD = NIL
DO
   BEGIN
   /*NO FREE ECB IS AVAILABLE.  PLACE THE TCB OF THIS
     TASK AT THE TAIL OF THE READY QUEUE AND PLACE
     THE TASK CONTEXT ON THE SYSTEM STACK SO THAT
     THE TASK WILL RESUME EXECUTION HERE WHEN IT AGAIN
     BECOMES ACTIVE.  CONTROL MUST THEN BE TRANSFERRED
     TO THE BEGINNING OF THE TASK LOOP IN THE
     EXECUTIVE, SO THAT OTHER TASKS MAY PERFORM
     THEIR PROCESSING.  WHEN THIS TASK REACHES THE
     HEAD OF THE READY QUEUE, IT WILL AGAIN CHECK
     FOR A FREE ECB, AND WILL PROCEED IF ONE IS
     AVAILABLE*/
   /*THIS ROUTINE MAY BE CODED IN ASSEMBLY LANGUAGE*/
   END
/*REMOVE FREE ECB FROM QUEUE AND RETURN TO CALLCOMPLETE*/
NEWECBPTR := ECBAVAILQHD;
ECBAVAILQHD := ECBAVAILQHD^.FLINK;
END.
```

3.4.3 EVENTCOMPLETE

```
PROCEDURE EVENTCOMPLETE (ECB)

/*AN EVENT CONTROL OR QUEUE CONTROL BLOCK IS PASSED TO THIS
  ROUTINE, SPECIFYING THE TCB AND EVENT NUMBER FOR PROCESSING.
  IF THE TASK IS SUSPENDED, THE CONDITIONS FOR ACTIVATING THAT
  TASK ARE EXAMINED, AND THE TASK IS MADE READY IF THOSE
  CONDITIONS ARE SATISFIED.*/

   BEGIN
      TCBPOINTER := ECB^.TCBPTR;
      EVENTMASK := MASKS(ECB^.EVENTFLAG);
      IF TCBPOINTER^.EFLAGSPOSSIBLE AND EVENTMASK <> 0
         THEN BEGIN
            TCBPOINTER^.EFLAGSACTUAL :=
               TCBPOINTER^.EFLAGSACTUAL OR EVENTMASK;
            CONDITIONSATISFIED := FALSE;
            IF TCBPOINTER^.WAITTYPE = WAITANY
               THEN FLAGSWAITEDFOR := EVENTMASK
               ELSE FLAGSWAITEDFOR := TCBPOINTER^.WAITFLAGS;
            IF TCBPOINTER^.WAITFLAGS AND EVENTMASK =
               FLAGSWAITEDFOR THEN CONDITIONSATISFIED := TRUE;
            IF CONDITIONSATISFIED
               AND (TCBPOINTER^.KILLECBPTR = NIL)
               THEN BEGIN
                  TCBPOINTER^.EFLAGSACTUAL :=
                     TCBPOINTER^.EFLAGSACTUAL AND
                     NOT FLAGSWAITEDFOR;
                  TCBPOINTER^.EFLAGSPOSSIBLE :=
                     TCBPOINTER^.EFLAGSPOSSIBLE AND
                     NOT FLAGSWAITEDFOR;
                  IF TCBPOINTER^.WAITTYPE := SUSPEND
                     THEN ECB^.STATUSPTR^.STATUS := ECB^.STATUS
                     ELSE TCBPOINTER^.OCCURREDEVENTPTR^.EVENTS:=
                        FLAGSWAITEDFOR;
                  /* PUT TCB ON READY-Q */
                  TCBPOINTER^.WAITFLAGS := 0;
                  TCBPOINTER^.WAITTYPE := NIL;
                  LINK(TCBPOINTER, RDYQHD, TCBPOINTER^.
                     TCBQLINK);
               END
               ELSE IF TCBPOINTER^.KILLECBPTR <> NIL
                  THEN DISMEMBER (TCBPOINTER);
         END;
   END.
```

4 USER's MANUAL

This chapter contains the VTS/OS User's Manual. It describes the various operating system services which are available to the applications programmer, together with the calls which are used to request these services. Those tasks which are not directly callable by applications are included in Sections 5 and 6. The system services are described here only from a functional viewpoint, with no explanations given of the internal workings of the routines which provide the services. For each call, a description is given of the use of the call, as well as the various input and output parameters. The possible completion codes for each call are also listed, although the various codes are given only as symbolic values (e.g., NO-ERROR if no error was encountered, ERR-NMR if the system resources required to satisfy the request were unavailable, etc.). It is anticipated that during the implementation of VTS/OS, the actual values would be made available in a global symbol dictionary, thereby obviating the need for application programs to code in specific values.

Most of the system service requests can be processed either synchronously or asynchronously, at the requesting task's option. If a task requests asynchronous service, it may receive two "completion" notifications. The first, which is always returned synchronously, is an indication of whether a preliminary check on the call parameters and format revealed any error. If no error was detected, the initial notification is a code of NO-ERROR, and the task will receive, at some later time, an asynchronous notification of service completion. This notification will indicate whether or not the call has been processed successfully. If an error is detected in the preliminary check, no further processing of the request is done by VTS/OS. If the task requests synchronous service, it will receive only one completion notification.

If the notification code is NO-ERROR, the call has been processed successfully. If the code is not NO-ERROR, either an error was detected in the parameters or call format during preliminary processing, or else errors of some kind occurred during the subsequent processing of the request.

Several asynchronous routines (e.g., TIMEASYNC, CLOSEASYNC, IORESETASYNC) are infinite loops which never terminate and have no strict calling sequence. They are awakened by Event Control Blocks, which are enqueued on input, and subsequently signal that the requested asynchronous event has completed or an error has been detected.

4.1 EVENT MANAGEMENT

In VTS/OS, tasks may voluntarily give up control to other tasks. They may also control the internal sequencing of operations by delaying themselves for a period of time or by waiting for the completion of one or more events. This section discusses the service requests that provide for the sequencing of operations within a task.

4.1.1 Events

VTS/OS makes extensive use of events for the purposes of task sequencing and synchronization. In VTS/OS, an event is defined as the setting of an event flag belonging to a task. This can occur in several ways. For example, most of the system facilities provided to tasks by VTS/OS can cause events to occur at their completion. In addition, a task that requests a service from a child task can specify that the completion of that service shall cause an event to occur. An event is considered to be capable of occurring if a call has been issued that specifies that the event shall occur upon completion of the request processing, and completion has not yet occurred.

Each task in VTS/OS has 15 event flags associated with it, numbered from EFLAG1 to EFLAG15. In general, for the EVENTFLAG parameter of a call any of several values may be specified. Specifically permissible values are discussed with each call. A value in the range EFLAG1 to EFLAG15 indicates that the specified event shall occur at the completion of the request processing. A value in this range does not cause the task to relinquish the CPU. All of the numbered event flags are interchangeable with each other; no prior association exists between a specific flag and a service request. A value of "SUSPEND" (a system-defined global constant) indicates that the task making the call wishes to give up the use of the CPU (i.e., suspend itself) until the event has occurred.

In this case, VTS/OS will select an unused event flag and associate it with the call, transparent to the task. Other mechanisms for relinquishing the CPU are discussed below.

Each event flag may also have three words of information about the completion of the request associated with it. In general, these three words consist of one word of status information, followed by two words of request-specific data. The status word will have the value "REQUEST IN PROGRESS" until the request is completed. The two data words are in general meaningless until the request has completed. A mechanism for accessing this information is described below in 4.1.3.3. If "SUSPEND" is used as the value for the EVENTFLAG parameter, the three words of information will be returned to the user in the STATUS parameter. The STATUS parameter must always be an array of at least three words.

4.1.2 Time-Interval Waits

4.1.2.1 SETDELAY (EVENTFLAG, SECONDS-TO-WAIT, STATUS)

This call allows a task to set a delay timer for the number of seconds specified in SECONDS-TO-WAIT. The task may specify either a value in the range EFLAG1 to EFLAG15 or the value "SUSPEND" for the EVENTFLAG parameter. In the former case, the task will not relinquish the CPU; in the latter case, the task will be suspended until the interval specified in SECONDS-TO-WAIT has elapsed. The task may not specify zero as the EVENTFLAG. SECONDS-TO-WAIT must be in the range 1 to MAXSEC, a system-defined global constant. At the return to the calling

task, the STATUS parameter will contain a code indicating either that the call has been accepted or the reason for the rejection. The possible values for the STATUS parameter are listed below.

NO-ERROR -- successful call

ERR-TIME -- number of seconds either less than 1 or greater than MAXSEC

ERR-EFLAG -- illegal EVENTFLAG specification

ERR-DELETED -- the time-out was cancelled by a subsequent REMOVEDELAY call

4.1.2.2 REMOVEDELAY (EVENTFLAG, STATUS)

This call allows a task to cancel an outstanding SETDELAY request. Cancellation of the SETDELAY call does not automatically free the event flag associated with that call; rather, the REMOVEDELAY call forces an immediate completion of the event associated with the SETDELAY. The EVENTFLAG parameter in the REMOVEDELAY call is used to identify the SETDELAY call and must, therefore, be the flag specified in the original SETDELAY call. STATUS is used to return a success or failure code to the calling task. It will contain one of the following values:

NO-ERROR -- no error encountered by REMOVEDELAY

ERR-EFLAG -- illegal EVENTFLAG specification

ERR-WRONGFLAG -- flag is not associated with a SETDELAY call

ERR-TOOLATE -- the interval specified in the original SETDELAY call has already elapsed

Although this call forces immediate completion of the SETDELAY call, it does not free the event flag. The task must still use GETSTATUS (q.v., Section 4.1.3.3) to free the flag. If the STATUS returned for REMOVEDELAY is NO-ERROR, the STATUS returned for GETSTATUS (which is actually the completion STATUS of the SETDELAY call) will be ERR-DELETED, indicating that the SETDELAY call was

successfully forced to an early completion. If the STATUS returned for REMOVEDELAY is ERR-TOOLATE, the STATUS returned for GETSTATUS will be NO-ERROR, indicating that the SETDELAY call ran to a normal completion.

4.1.2.3 PAUSE

This call allows a task to relinquish the CPU for a short time to give other tasks access to the CPU. The task will simply be placed at the end of the scheduling queue. It will remain eligible to use the CPU, as there is no requirement that any event occur before the task uses the CPU again. However, the task will not regain control of the CPU until it has reached the head of the Ready queue, which will take an unpredictable amount of time. This call should be used to release the CPU during long computational sequences that might otherwise monopolize the CPU.

4.1.3 Event-terminated Waits

4.1.3.1 WAITANY (SET-OF-EVENTS, OCCURRED-EVENT)

This call allows a task to relinquish the CPU until any of the events specified in SET-OF-EVENTS, that is capable of occurring, has occurred. Events specified in SET-OF-EVENTS that are not capable of occurring will not be waited for. If none of the events in SET-OF-EVENTS can occur, no wait will be performed. At the return, all events which have actually occurred will be indicated in OCCURRED-EVENT. If no wait has been performed, OCCURRED-EVENT will be zero. If more than one of the events specified in SET-OF-EVENTS has already occurred at the time that the WAITANY call is made, OCCURRED-EVENT will contain the bit index of the lowest-number event to have occurred.

SET-OF-EVENTS and OCCURRED-EVENT both use bit-position indices to indicate events. The low-order bit corresponds to event flag 1;

the next-to-high-order bit corresponds to event flag 15. Even if a request is satisfied immediately, either because no event can occur or because one or more have already occurred, the task will still be placed at the end of the Ready queue, and will not regain control of the CPU until it has reached the head of the Ready queue.

4.1.3.2 WAITALL (SET-OF-EVENTS, OCCURRED-EVENTS)
This call allows a task to relinquish the CPU until all of the events specified in SET-OF-EVENTS that are capable of occurring have occurred. Events specified in SET-OF-EVENTS that are not capable of occurring will not be waited for. If none of the events in SET-OF-EVENTS can occur, no wait will be performed.

At the return, all events that have actually occurred will be indicated in OCCURRED-EVENTS. If no wait has been performed, OCCURRED-EVENTS will be zero.

SET-OF-EVENTS and OCCURRED-EVENTS both use bit-position indices to indicate events. The low-order bit corresponds to event flag 1; the next-to-high-order bit corresponds to event flag 15. Even if a request is satisfied immediately, either because no event can occur or because all that are necessary have already occurred, the task will still be placed at the end of the Ready queue, and will not regain control of the CPU until it has reached the head of the Ready queue.

4.1.3.3 GETSTATUS (EVENTFLAG, STATUS)
This call allows a task to examine the three words of information associated with an event flag. A value in the range EFLAG1 to EFLAG15 must be specified for the EVENTFLAG. The corresponding three words of status and data will be placed in the three words of STATUS. The four possible conditions for the EVENTFLAG are discussed below.

a. The EVENTFLAG is not in the range EFLAG1 to EFLAG15. In this case, a value of "ERR-EFLAG" will be returned in the first word of STATUS, and the second and third words will be zeroed.

b. The EVENTFLAG is not associated with any service call whatsoever. In this case, a value of "FLAG-NOT-IN-USE" will be placed in the first word of STATUS, and the second and third words will be zeroed.

c. The EVENTFLAG is associated with a service call which has not yet completed. In this case, a value of "REQUEST-IN-PROGRESS" will be returned in the first word of STATUS, and the second and third words will contain either call-specific information or zero.

d. The EVENTFLAG is associated with a service call which has completed. In this case, a completion code will be returned in the first word of STATUS, and call-specific information will be returned in the second and third words. In addition, the association between this EVENTFLAG and the service call will be destroyed so that the EVENTFLAG will be considered free and unassociated, as described in 4.1.3.3.a above.

4.2 PROGRAM AND TASK MANAGEMENT

4.2.1 Overview

Within VTS/OS, a distinction is made between a program and a task. A program is a fixed, possibly re-entrant, body of code which exists in an executable form. It is controlled by an Image Control Block. A task is an invocation of a program, consisting of the code, together with any dynamically allocated system resources, such as buffers and any data. It is controlled by a Task Control Block. If a program is re-entrant, several tasks can execute the same program. In this case, a Task Control Block exists for each task.

VTS/OS provides a set of system service requests which enable tasks to control the execution of other tasks. Under VTS/OS, a task can cause another program to be brought into the same processor, to execute as a task, to stop executing, and to be removed from the processor. The invoked task may also be able to communicate brief results back to the invoking task by a mechanism other than normal interprocessor communication, as described in Section 4.5. Every task has a complete set of capabilities so that Task A can invoke Task B which, as part of its execution, can bring programs C and D into the processor, cause them to execute as Tasks C and D, receive partial results from them, use these results to formulate a final result to pass back to Task A, and then remove programs C and D from the processor.

Within VTS/OS, tasks are organized in a hierarchical structure. No task may have more than one parent, although a parent may have many children. All of the children must reside in the same processor as the parent. There is no such structure for the corresponding programs.

Among tasks that are lineally related (i.e., for a given task, its children and parent, grandchildren and grandparent, etc.), capabilities and constraints arise that are not relevant for non-related tasks. For example, the POST operation allows a task to transmit three words of information back to its parent. This capability is available only to the child in a parent-child relationship. On the other hand, the KILL operation, which unequivocally stops a specified task, also stops each and every lineal descendant of that task.

4.2.2 Program Management Calls

4.2.2.1 INSTALL (EVENTFLAG, NAME, STATUS)

This call allows a task to have a program brought into the processor in which the calling task resides. The program whose ASCII name is given in NAME will be located and loaded (if necessary, down-line loaded) into the calling task's processor. An Image Control Block will then be created for the newly loaded program. The file containing the program must have the same name as the program itself.

When the request processing is completed, completion processing will occur as specified by the value in EVENTFLAG. EVENTFLAG may be either a value in the range EFLAG1 to EFLAG15, in which case the specified event will occur at completion, or the value SUSPEND, in which case the calling task will be suspended until completion.

At the return to the calling task, the STATUS parameter will contain a notification of either success or an error in the call or processing of the call. The possible values for STATUS are given below.

NO-ERROR -- no error encountered

ERR-TAI -- program by this name is already installed

ERR-EFLAG -- illegal EVENTFLAG specified

ERR-NMR -- no more room in memory for this program

ERR-IO -- some kind of I/O error occurred during the processing of this request; program has not been installed. In this case, the second word of the STATUS variable will contain the actual I/O error code. Note that if a program name is given improperly, it will be impossible to open the file, and this error will result.

This call does not create or imply any parent-child relationship between the task making the call and the program name specified in the call.

4.2.2.2 REMOVE (EVENTFLAG, NAME, STATUS)

This call allows a task to have a program removed from the processor. It performs a complementary function to the INSTALL call (q.v. 5.2.1.1). A program may only be REMOVEd after every running copy of it has been KILLed. Once the REMOVE call has been issued, an INSTALL must be performed before a task using that program can be SCHEDULEd. Unlike a KILL, a REMOVE call affects only the program whose ASCII name is in NAME, and not any of its descendants.

The EVENTFLAG parameter is used to determine which event shall occur when the REMOVE processing is complete. A value in the range EFLAG1 to EFLAG15 indicates that the specified event shall occur when the REMOVE processing is complete. In this case, the calling task may use the GETSTATUS call to obtain information about the completion. A value of "SUSPEND" indicates that the calling task is to be suspended until the REMOVE processing is complete.

The STATUS parameter is used to return information to the calling task about the success or failure of the REMOVE operation. The possible returned values for STATUS are the following:

NO-ERROR	-- no error was encountered
ERR-FLAG	-- illegal EVENTFLAG was specified
ERR-TNI	-- the program specified is not currently installed
ERR-ALIVE	-- not all running tasks of this program have been KILLed
ERR-SUICIDE	-- a task cannot have its own program REMOVEd

4.2.3 Task Management Calls

4.2.3.1 SCHEDULE (EVENTFLAG, NAME, TASKID, STATUS)

This call may be used by a task to create and execute a task for the program whose ASCII name is in NAME. The invoking task may make the invoked task its child or it may make it totally independent. The invoked task will be placed in the Ready queue, and will be able to begin execution when it is given control of the CPU. The invoked task's program must already be resident in the processor, as the result of either system initialization procedures or an earlier INSTALL call.

The EVENTFLAG parameter may have several values. A value in the range EFLAG1 to EFLAG15 indicates that the invoked task is to be a child of the calling task. The specified event will occur when the child performs a POST operation, after which the parent may use GETSTATUS to obtain the three words of information passed by the child. A value of "SUSPEND" used as the EVENTFLAG also indicates that the involved task is to be a child; in this case, the parent's suspension will be lifted when the child performs a POST, which will place three words of information directly in the STATUS parameter. The final permissible value for EVENTFLAG is "INDEPENDENT"; in this case, the invoked task is independent of the invoking task.

If no error has occurred, TASKID will contain a unique identifier of the invoked task at the return to the calling task. This TASKID will be used in the KILL call. If an error has occurred, TASKID will be zero.

At the return to the calling task, if a value in the range EFLAG1 to EFLAG15 or "INDEPENDENT" was specified for the EVENTFLAG, the STATUS parameter will contain a code indicating whether the system accepted the call or rejected it for syntactical reasons. If a value of "SUSPEND" was specified for the EVENTFLAG, the STATUS parameter will contain either notification of a syntactical error or the information resulting from the child's POST operation.

The possible values that may be returned in STATUS are listed below.

NO-ERROR -- no error encountered
ERR-TNI -- task specified has not been installed
ERR-EFLAG -- illegal EVENTFLAG specified
ERR-NMR -- not enough room in memory to run this task
ERR-CHILD -- child died before POSTing
ERR-PERMIT -- program is flagged for removal

4.2.3.2 POST (DATAWORDS, STATUS)

This call allows a child task to communicate three words of information back to its parent. The content and significance of the three words must be mutually agreed upon by the child and parent tasks. The three words of information will be taken from the child's DATAWORDS parameter and passed to the parent as the STATUS associated with the EVENTFLAG specified in the parent's SCHEDULE call. Since the parent obtains this information as the STATUS of its SCHEDULE call, care should be taken to avoid placing in the first word of the STATUS any value that is also one of the possible error codes resulting from the SCHEDULE call. A child may only POST to its parent once.

At the return to the calling task, STATUS will contain a value indicating either success or the reason for failure. The possible values for STATUS are listed below.

NO-ERROR	--	no error encountered
ERR-INDEP	--	task has no parent
ERR-AUU	--	task has already POSTed to its parent once

4.3.2.3 KILL (EVENTFLAG, TASKID, STATUS)

This call is used by a task to stop either itself or a first-generation child. If a task is stopping itself, it sets TASKID to zero. If it is stopping a child, it must give the child's task identifier in TASKID.

If a task is stopping itself, the system will not return to it if the call is successful, so that the other two parameters will be unused. If a task is stopping its child, it can use EVENTFLAG to specify which event shall occur when the KILL processing is complete. A value in the range EFLAG1 to EFLAG15 indicates that the specified event shall occur when the KILL processing is complete. In this case, the task may use the GETSTATUS call to obtain information about the completion. A value of "SUSPEND" indicates that the calling task is to be suspended until the KILL processing is complete.

The STATUS parameter is used to return information to the calling task about the success or failure of the KILL operation. The possible returned values for STATUS are the following:

NO-ERROR	--	no error was encountered
ERR-EFLAG	--	illegal EVENTFLAG specified (note that even a task killing itself can get this error return)
ERR-NYC	--	TASKID is neither zero nor the task identifier of a first-generation child of the calling task

Stopping a task will automatically stop every lineal descendant of that task.

4.3 MEMORY MANAGEMENT

In the absence of detailed hardware specifications, the area of memory allocation and memory management cannot be discussed with any great detail or precision. While it is known that the VTS/OS processors will all be mapped processors, the details of the mapping mechanism have not yet been determined. One important aspect of the mechanism that is as yet unknown is the granularity of the mechanism, i.e., how many map registers it takes to span the logical address space. For example, in an architecture in which 8 map registers span the entire 32K address space, the granularity is 4K. Under other architectures, as many as 32 and 64 registers may be used to span the same space, providing a granularity of 1K to .5K. Thus, with finer granularity, more flexibility can be provided in VTS/OS, although a tradeoff is made in terms of increased overhead. The remainder of this discussion will assume that sufficient granularity exists in the mapping mechanism to support the features and capabilities described. When a particular feature or capability depends on this specific assumption, a discussion of the various alternatives will be provided.

All memory in the VTS/OS system can be divided into three parts: system memory, fixed task images, and variable task data buffers. Within VTS/OS, an extensive set of primitives is provided to the application tasks which enables them to perform overlaying of code, to request variable data buffers, to switch back and forth between several buffers without losing control of any of them, and, within family lines, to share buffers between tasks.

4.3.1 Task Memory Structure

This section will discuss the overall memory structure of an applications task. The structure is analyzed by examining the allocation of memory map registers to the various portions of a task.

Each applications task can be divided into four portions, not all of which need appear in every task. Since a block of map registers will be permanently assigned to each portion of the task image, a finer granularity for the mapping hardware provides a definite payoff in terms of the increased flexibility in sizing each of the portions of the task.

The first portion of the task contains the main program code, or root. This portion will be write-protected, if the mapping hardware will support this feature. This portion will not be under the explicit control of the applications task for purposes of mapping into it and out of it; rather, the applications task will be mapped into this portion when the task is first SCHEDULEd, and will not be mapped out until the task is KILLed. This portion will also contain routines to interface with VTS/OS.

The second portion of the task image is the overlay space. Obviously, if a task uses no overlays, no map registers will be allocated to this portion. This area, like the root, will be write-protected if possible. However, this portion will be under the explicit control of the applications task. The calls that are used to manage this space are discussed in Section 4.3.2. The number of map registers reserved for the overlay space will be sufficient for the longest overlay (if only one level of overlaying is permitted) or for the longest branch in the overlay tree (if multi-level overlaying is permitted). This portion of the task address space will always be reserved for overlays, even when the task is not mapped to any overlays.

The third portion of the address space is for storage of variables. This portion will not be write-protected. It will also not be under the explicit control of the applications task. As with the root portion, the applications task will be mapped into this portion when the task is first SCHEDULEd, and will not be mapped out until the task is KILLed. This portion will contain the variable storage for the root as well as for every overlay. Whether or not this is feasible is a function both of the granularity of the mapping hardware and of the facilities provided by the compiler and linker used. With sufficiently fine granularity, it may not be necessary to combine all the variable spaces (i.e., that for the root and those for the overlays) into a single variable space. On the other hand, the compiler and linker may do this as part of their normal operation. This combining can always be accomplished in a rather brute force manner by declaring all variables used in every overlay as GLOBAL variables in the root program.

The fourth portion of the task's address space is buffer space. If a task does not use any buffers, the map registers that are left over will go unused. The difference between buffer space and overlay space is that VTS/OS will be able to determine whether or not a task requires any registers to map into its overlay space by examining the information produced by the compiler and linker, while it will not be able to do so for buffer space. This space, which will not be write-protected, will be under the explicit control of the applications task. The calls that are used to manage this space are discussed in Section 4.3.3.

4.3.2 Overlay Control

As VTS/OS will be required to operate on processors without direct access to discs, the overlaying process will necessarily have to consist of mapping a task to an area of physical memory which

already contains the required overlay. All overlays which are associated with a particular program will have to be installed in memory when the program is initially installed. The VTS/OS INSTALL command processing will ensure that this condition is met, without the user having to install each overlay separately. Although overlays may be re-entrant, as are the main programs to which they belong, they may not be shared among programs. If two or more programs wish to use the same overlay, each program must have a separate copy of the overlay. The creation of the linkage between a main program and its various overlays will be the function of the compiler and linker.

4.3.2.1 LOADOVERLAY (OVERLAYID, STATUS)

This call allows a task to request that a particular overlay be mapped into the task's address space. The call itself must not be in a portion of the task's address space that will be occupied by the overlay. However, if another overlay occupies that space, no indication of this condition will be given. The parameters of this call are given below.

OVERLAYID -- the numeric identification of the overlay to be mapped in

STATUS -- returned code indicating success or failure of the call -- possible values are:
NO-ERROR -- no error encountered
ERR-OLAYID -- specified overlay does not exist

4.3.2.2 RELEASEOVERLAY (OVERLAYID, STATUS)

This call allows a task to release an overlay (i.e., to free a portion of its logical address space) without mapping to another

overlay. However, it is not necessary to call RELEASEOVERLAY before calling LOADOVERLAY to map another overlay. The parameters for this call are given below.

OVERLAYID -- the identifier of the overlay to be "mapped out"

STATUS -- a returned code indicating success or failure of the call -- possible values are:
NO-ERROR -- no error encountered
ERR-MAP -- task is not currently mapped to the specified overlay
ERR-OLAYID -- specified overlay does not exist

4.3.3 Buffer Control

VTS/OS provides an extensive set of memory allocation capabilities to applications tasks which includes the ability to request a buffer of any size, starting at a user-specified logical address in the user's address space, to have several buffers under the user's control which share this address space, to switch back and forth between buffers, to return buffers, and, within family lines, to share buffers.

Unless otherwise specifically requested by tasks, no two tasks will have access to the same physical block of buffer space. However, this protection feature may not be implementable at a reasonable cost if it should turn out that the typically requested buffer size is significantly smaller than the granularity provided by the mapping mechanism. In this case, it may be necessary for two or more tasks to share a buffer block, with each task being told by VTS/OS which portion of the buffer block it may use. Under these circumstances, protection would not be enforceable.

4.3.3.1 GETBUFFER (LOGADDR, NWORDS, SHAREMODE, BUFFERID, STATUS)
This call allows a task to request a buffer. All processing for this call will be performed synchronously, so that there is no need for an EVENTFLAG. The parameters of the call are described below.

LOGADDR -- the logical address in the user's address space at which this buffer is to begin -- must be beyond the end of the main program code, and must be aligned according to the granularity of the mapping mechanism.

NWORDS -- the number of words of buffer space requested -- must be at least one, and cannot be so large as to extend beyond the end of the logical address space when mapped in beginning at LOGADDR

SHAREMODE -- if PRIVATE, no other task may use this buffer -- if SHARABLE, calling task allows lineal descendants to map this buffer -- (Note: see EQUATEBUFFER for a description of the mechanism which allows the calling task to control access to this buffer)

BUFFERID -- a value returned to the calling task which uniquely identifies this buffer

STATUS -- a code returned to the calling task indicating success or reason for failure -- possible values are:

NO-ERROR -- no error encountered

ERR-ADDR -- logical address specified is illegal

ERR-NWORDS -- number of words specified is illegal

ERR-NMR -- buffer space is not available

4.3.3.2 EQUATEBUFFER (PARENTBUFFERID, PARENTTASKID, MYBUFFERID, STATUS)

This call may be used by a task to gain access to a buffer originally allocated to one of its ancestors. The child must then use the MAPTOBUFFER call to map itself into the buffer. The child must obtain the task ID and original buffer ID from the ancestor in order for this call to succeed. VTS/OS does not provide a specific mechanism to enable a task to obtain this information from its ancestor; thus, each "family" of tasks may use whatever mechanism is most appropriate. The parameters for this call are described below.

PARENTBUFFERID -- the buffer ID returned to the original "owner" of the buffer in response to a GETBUFFER call

PARENTTASKID -- the task ID of the original "owner" of the buffer

MYBUFFERID -- the buffer ID returned to the calling task by VTS/OS, and used by this task as the BUFFERID parameter for subsequent RELEASEBUFFER and MAPTOBUFFER calls

STATUS -- a code indicating success or a reason for failure -- possible values are:

NO-ERROR -- no error encountered
ERR-NMR -- system resources not available
ERR-BUFFID -- value given in PARENTBUFFERID is not a buffer which is assigned to the task in PARENTTASKID
ERR-PARENT -- task specified in PARENTTASKID is not an ancestor of the calling task
ERR-PERMIT -- buffer specified in PARENT-BUFFERID is either not sharable, or is flagged for release

This mechanism does not prevent a child, e.g., Task B, from obtaining an original buffer ID and task ID from an ancestor, e.g., Task A, and then passing on the original buffer ID and task ID to its descendants, without Task A being informed.

4.3.3.3 MAPTOBUFFER (BUFFERID, STATUS)

This call allows a task to map itself to a buffer that has been previously allocated to it, or to which it has gained access from a lineal ancestor. This call must be used before a buffer can be addressed. The parameters of the call are described below.

BUFFERID -- the identifier of a buffer which was returned in a previous GETBUFFER or EQUATEBUFFER call

STATUS -- a code returned to the calling task indicating success or reason for failure -- possible values are:
NO-ERROR -- no error encountered
ERR-BUFFID -- buffer ID not assigned to this task
ERR-PERMIT -- buffer is flagged for release (Note: only the buffer owner can receive this error code)

4.3.3.4 RELEASEBUFFER (EVENTFLAG, BUFFERID, STATUS)

This call allows a task to release a buffer when it is through using that buffer. While the actual mechanics of the release depend upon whether the buffer is private or sharable, whether the calling task is the owner or not, and whether more than one task has access to the buffer at the time of the call, the net effect of the call is the same. That is, if successiul, the calling task can no longer access the buffer.

If the buffer is a private buffer, it will simply be returned. If the buffer is a sharable buffer and the calling task is not the owner, the buffer will appear to that task to have been returned. If the buffer is sharable and the calling task is the owner, the buffer will be flagged for release and will actually be released when all descendants have also released it. Until the buffer is released, no additional descendant tasks will be allowed to EQUATEBUFFER to that buffer again, even if it has not yet been returned. However, descendant tasks that have already executed an EQUATEBUFFER call will still be able to MAP to the buffer.

The parameters for this call are given below.

BUFFERID -- the identifier of a buffer which was returned in a previous GETBUFFER or EQUATEBUFFER call

EVENTFLAG -- the event flag to be used to indicate completion -- - if calling task is the owner:
if EVENTFLAG is in the range EFLAG1 to EFLAG15, EVENT FLAG will be set when the buffer is returned to the free pool; if EVENTFLAG is SUSPEND, the calling

task will be suspended until the buffer is returned to the free pool.

- if calling task is not the owner: processing of this call is synchronous, but EVENTFLAG must still be EFLAG1 to EFLAG15 or SUSPEND

STATUS -- a code returned to the caller indicating success or a reason for failure -- possible values are:

NO-ERROR -- no error encountered

ERR-BUFFID -- Buffer ID is not assigned to this task

ERR-EFLAG -- illegal event flag specified

ERR-IOIP -- I/O transfer to/from this buffer still in progress

4.4 INTER-TASK COMMUNICATIONS

The system calls provided for Inter-Task Communications (ITC) differ from those described in the Phase II report; however, the differences are largely in terms of the parameters passed by and events signalled to the user tasks, and do not greatly alter the communications process. As a result, ITC system calls are more consistent with other calls the user may issue to the operating system.

From the viewpoint of the sending task, the SENDMESSAGE call is issued with the addresses and lengths of the message and answer buffers, a timeout parameter (the maximum time for completion of the conversation), the name of the destination task, and an event flag of SUSPEND or EFLAG1..EFLAG15. If the SUSPEND event flag is used, control will be returned to the sending task when the answer has been received from the receiving task and is available in the answer buffer. If one of EFLAG1..EFLAG15 is used, the sending task may perform additional processing and then wait on the event flag specified in the SENDMESSAGE call. The answer will be available when the sender's wait condition involving his event flag is satisfied. In either case, STATUS.ERROR contains NOERROR if the conversation proceeded normally, or an error code indicating the nature of any error detected during the conversation. Return from the wait condition associated with the SENDMESSAGE call terminates the conversation.

From the viewpoint of the receiving task, a REQUESTMESSAGE call must have been issued to associate the receiving task's "IN" queue with an event flag before messages may be received by the task. Messages arriving before the REQUESTMESSAGE call will be retained, but the receiving task does not have access to them until the call is issued. Upon returning from a REQUESTMESSAGE call with an event flag of SUSPEND or a wait involving one of EFLAG1..EFLAG15 specified in the REQUESTMESSAGE call, a GETSTATUS

call will return to the receiving task the message ID of the first message on its message input queue. The receiving task then issues the TRANSFERMESSAGE call to move the message to his message buffer, regardless of the size of the message and regardless of whether the sending task is local or is in another processor. If the message needs to be transmitted from another processor, TRANSFER-MESSAGE will request the transmission, and the message will be transferred without involving either the sending or receiving tasks. After the TRANSFERMESSAGE request is complete, the receiving task has access to the message data and may formulate an answer. The receiving task uses the SENDANSWER call to transmit his answer to the sending task; the completion status of the SENDANSWER call indicates only that the answer was sent, and not that it was received. At this point, the receiving task has completed the conversation.

The allocation of buffer space for messages and answers is not discussed in the above treatment of ITC, since the tasks involved may allocate buffers in advance or at the time of the conversation. However, the normal considerations for deadlock prevention apply to any allocation of buffers while the tasks are running. The only restriction imposed by the ITC functions is that any memory space used for messages or answers must be mapped into the logical address space of the task issuing an ITC system call at the time of that call.

4.4.1 SENDMESSAGE

The SENDMESSAGE call initiates the transfer of a message from one task to another, regardless of which processors contain the two tasks. The destination task is referenced by name, and its location is determined transparent to the sending task. Buffer specifications for the message and answer buffers contain a buffer ID, a starting address for the allocated area, and the length of that area in words. The timeout parameter is provided to allow the sending task to indicate the maximum amount of time the requested

operation should take if the system is operating properly; if the answer has not been received within this time, processing of the conversation is terminated and an error code of ERTIMEOUT is returned to the sending task.

Calling sequence -

```
   SENDMESSAGE (EVENTFLAG, PROCESSNAME, MBUFFSPEC, ABUFFSPEC,
               TIMEOUT, STATUS);
```

Parameters -

```
      EVENTFLAG:  EFLAG1..EFLAG15, SUSPEND
      PROCESSNAME:  PROCNAMETYPE;  /*name of destination task*/
      MBUFFSPEC=  record  /*descriptor for message buffer*/
                     BUFFERID: 1..MAXBUFFERID;
                     ADDRESS:  integer;
                     LENGTH:  integer
                  end;
      ABUFFSPEC=  record  /*descriptor for answer buffer*/
                     BUFFERID:  1..MAXBUFFERID;
                     ADDRESS:  integer;
                     LENGTH:  integer
                  end;

      TIMEOUT:    integer;  /*maximum time allowed for completion*/
      STATUS:     STATUSTYPE;
                  /*ERRORS:
                     ERNOPROCESS      process not found
                     ERBUFFSPEC       invalid buffer specification
                     ERTIMEOUT        operation not completed
                                      within specified interval
                     ERBUSERROR       transmission retry count
                                      exceeded or other bus error*/
```

4.4.2 REQUESTMESSAGE

The REQUESTMESSAGE call is used by the receiving task to initiate a conversation. When a message has arrived for the task, the MESSAGEID parameter will contain the message ID which will be used by the receiving task for all operations in this conversation. If the user task does not have buffer space allocated for the message and answer, the space must now be allocated. MSLENGTH and ANSLENGTH contain the size required for the message and the maximum answer length.

Calling sequence -

```
    REQUESTMESSAGE (EVENTFLAG, MESSAGEID, MSLENGTH
                    ANSLENGTH, STATUS)
```

Parameters -

```
    EVENTFLAG:    EFLAG1..EFLAG15, SUSPEND
    MESSAGEID:    record
                  MESSAGENUMBER: integer;
                  PROCESSOR ID: integer
                  end;
    MSLENGTH:     integer;
    ANSLENGTH:    integer;
    STATUS:       STATUSTYPE;
                  /*ERRORS:
                     ERNOIPC            task not defined to receive
                                        messages*/
```

4.4.3 TRANSFERMESSAGE

TRANSFERMESSAGE is called by the receiving task when it wishes to obtain access to the message data of a particular conversation. If the sending task is in another processor, TRANSFERMESSAGE will accomplish the transmission and moving of message data so that the receiving task may have access to it. If the sender is in the receiver's processor, the TRANSFERMESSAGE call will move the message from the message buffer of the sending task to that specified by the receiving task.

Calling sequence -

```
   TRANSFERMESSAGE (EVENTFLAG, MESSAGEID, MBUFFSPEC, ABUFFSPEC,
   STATUS);
```

Parameters -

```
   EVENTFLAG:  EFLAG1..EFLAG15, SUSPEND
   MESSAGEID:  record
               MESSAGENUMBER:  integer;  /*VALUE1 from GETSTATUS*/
               PROCESSORNUMBER:  integer  /*VALUE2 from GETSTATUS*/
               end;

   MBUFFSPEC:  record
               BUFFERID:  integer;
               BUFFAD:  MEMORYADDRESS;
               BUFFSIZE:  integer
               end;

   STATUS:  STATUSTYPE;
               /*ERROR:
                  ERMESSAGEID   Invalid message ID*/
```

4.4.4 SENDANSWER

The receiving task issues a SENDANSWER call to terminate the conversation and return acknowledgement or data to the sending task. If the sending and receiving tasks are in the same processor, the answer is moved to the sender's answer buffer when the call is issued, and event completion is signalled on the event flag specified in the sender's SENDMESSAGE call. Otherwise, SENDANSWER will supervise transmission of the answer over the bus. As noted above, the status returned by SENDANSWER indicates that the answer was properly sent, without any connotation of acknowledgement on the part of the task that originated the conversation with the SENDMESSAGE call.

Calling sequence -

```
   SENDANSWER (EVENTFLAG, MESSAGEID, ABUFFSPEC, STATUS);
```

Parameters -

```
   EVENTFLAG:  EFLAG1..EFLAG15, SUSPEND;
   MESSAGEID:  record  /*From GETSTATUS call, VALUE1 and VALUE2*/
               MESSAGENUMBER:  integer;
               PROCESSORNUMBER:  integer
               end;
   ABUFFSPEC:  Record
               BUFFERID:  integer;
               BUFFAD:  MEMORYADDRESS;
               BUFFSIZE:  integer
               end;
   STATUS:  STATUSTYPE;
            /*Errors:
               ERMESSAGEID   Invalid message ID*/
```

4.5 FILE MANAGEMENT

4.5.1 Disc File Structures

File structures are provided by the VTS Operating System to give users a logical structure for storing and retrieving data. Two types of file structures are provided by VTS/OS, both of which may be used for either sequential or random data access, with the exception that sequential write operations cannot be performed on contiguous files.

4.5.1.1 Segmented Files

Segmented files consist of a set of Segment Pointer Blocks containing information on the location of data segments composing the file. Segment Pointer Blocks are linked together, with a two-word entry indicating the location of each data segment in the file. Additional Segment Pointer Blocks and data segments may be allocated, allowing extension of the file as needed. The format of the segmented file is described in detail in Section 5.5.1.

4.5.1.2 Contiguous Files

Contiguous files consist of a set of sequentially allocated disc sectors, allowing random access to the file without requiring that a Segment Pointer Block be read to determine the physical data location. The size of the contiguous file must be provided at file creation time, and cannot be extended once the file has been created. The format of the contiguous file is described in detail in Section 5.5.1. It is anticipated that the majority of files in VTS applications will be contiguous, since access to contiguous files is faster and the size of most applications files will be known.

4.5.2 File Access Methods

Random and sequential access methods are supported by VTS/OS. Either access method may be used on segmented or contiguous files. Buffering is provided by the operating system for all sequential operations, and for those random operations which do not affect an integral number of disc sectors.

4.5.2.1 Random Access

The user may specify data transfer of any size for random access, using the READ and WRITE calls. Records are designated by a file offset and record length, where the file offset is the number of words from the beginning of the file and the record length is the number of words to be read or written.

4.5.2.2 Sequential Access

For sequential access, the operating system maintains a file sequence pointer which indicates the number of words from the beginning of the file at which the next data transfer will begin. READSQ, WRITESQ, READLN and WRITELN system calls will read and write a user-specified number of words from the current location of the file sequence pointer, and will update the pointer to reflect the completed read or write.

4.5.3 File Maintenance Procedures

4.5.3.1 CREATE

The CREATE procedure allows the addition of files on system devices with directories. For contiguous files, the indicated number of sequential device blocks are allocated when the file is created. For segmented files, one Segment Pointer Block is allocated, but no data segments are allocated at creation time.

Calling Sequence

CREATE (EVENTFLAG, FILENAME, FILETYPE, SIZE, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
FILENAME	ASCII file name
FILETYPE	SEGMENTED or CONTIGUOUS
SIZE	if FILETYPE = CONTIGUOUS, file length in words if FILETYPE = SEGMENTED, data segment length in words
STATUS	error code; NOERRROR ERFILENAME invalid file name ERFILETYPE invalid file type if EVENTFLAG = SUSPEND, the following errors are possible: ERNODEV no such device or not a directory device ERFILEEXISTS file already exists ERCONTIG insufficient contiguous space ERSPACE insufficient space on device ERDIR directory error

4.5.3.2 DELETE

The DELETE procedure removes the specified file name from the directory for the specified device directory. Any space allocated to that file is released and, where possible, combined with adjacent free areas. This will aid in future allocation of contiguous files.

Calling Sequence

DELETE (EVENTFLAG, FILENAME, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND	
FILENAME	ASCII file name	
STATUS	error code:	
	NOERROR	
	ERFILENAME	invalid file name
	or, if EVENTFLAG = SUSPEND:	
	ERNODEV	no such device or not a directory device
	ERNOFILE	file does not exist
	ERPROTECT	file protected
	ERFILEOPEN	file still open
	ERDIR	directory error

4.5.3.3 RENAME

The RENAME procedure allows the user to change the name of an existing file. Files which are protected or are open cannot be renamed.

Calling Sequence

RENAME (EVENTFLAG, OLDFILENAME, NEWFILENAME, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
OLDFILENAME	ASCII name of existing file to be renamed
NEWFILENAME	ASCII name to replace OLDFILENAME
STATUS	error code:

NOERROR	
ERFILENAME	invalid file name
EREFLAG	invalid event flag
ERNODEV	no such device or not a directory device

or, if EVENTFLAG = SUSPEND:

ERNOFILE	file OLDFILENAME does not exist
ERFILEEXISTS	file NEWFILENAME already exists
ERPROTECT	file is protected
ERFILEOPEN	file still open
ERDIR	directory error

4.5.3.4 SETATTRIBUTES

The SETATTRIBUTES procedure allows the user to set the protection attributes of a file. The specified attributes are set for that file, regardless of whether or not they were already set. Attributes may not be set for non-directory devices.

Calling Sequence

SETATTRIBUTES (EVENTFLAG, FILENAME, ATTRIBUTES, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
FILENAME	ASCII file name
ATTRIBUTES	any combination of PROTECT, READPROTECT, WRITEPROTECT
STATUS	error code:

NOERROR	
ERFILENAME	invalid file name
EREFLAG	invalid event flag
ERNODEV	no such device or not a directory device

if EVENTFLAG = SUSPEND:

ERNOFILE	file does not exist
ERATTR	invalid attribute specification
ERFILEOPEN	file still open
ERDIR	directory error

4.5.3.5 GETATTRIBUTES

The GETATTRIBUTES procedure allows the user to access the protection attributes of a file. The specified attributes are transferred to the user's program space. GETATTRIBUTES has an implied SUSPEND event flag, if the file is on disc.

Calling Sequence

GETATTRIBUTES (FILENAME, ATTRIBUTES, STATUS)

Parameters

FILENAME	ASCII file name
ATTRIBUTES	any combination of PROTECT, READPROTECT, WRITEPROTECT
STATUS	error code:

NOERROR	
ERFILENAME	invalid file name
ERNODEV	no such device
ERNOFILE	file does not exist
ERDIR	directory error

4.5.4 Input/Output Operations

All input and output operations within VTS/OS are performed on a channel basis. Files and non-file devices (e.g., printers) are associated with a channel number for the duration of a task's I/O to that file or device.

4.5.4.1 OPEN

The OPEN procedure assigns a logical path between the user task and a file or device. The channel number provided in the OPEN call is used for all later I/O requests involving that file or device.

Calling Sequence

OPEN (EVENTFLAG, FILENAME, CHANNEL, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND	
FILENAME	ASCII file name	
CHANNEL	Channel number, returned by the OPEN call	
STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERFILENAME	invalid file name
	ERCHAN	invalid channel number
	or, if EVENTFLAG = SUSPEND:	
	ERNOFILE	file does not exist
	ERDEV	no such device or, if a named file was specified, not a directory device
	ERDIR	directory error

4.5.4.2 CLOSE

The CLOSE procedure terminates the logical equivalence between a file or device and the user-assigned channel number. Since CLOSE does not act asynchronously, no event flag is specified.

Calling Seqtence

CHANNEL	channel number	
STATUS	error code:	
	NOERROR	
	ERCHANNEL	invalid channel number
	ERNOTOPEN file not open	

4.5.4.3 IORESET

The IORESET procedure allows the user to close all open channels assigned to a particular task without explicitly referencing each channel number.

Calling Sequence

IORESET (EVENTFLAG, TASKID, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
TASKID	ID of task whose channels are to be closed
STATUS	error code:
	NOERROR
	ERTASKID — invalid task ID

4.5.5 Data Transfer Operations

All data transfer operations are performed to or from buffers allocated by the user task with GETBUFFER. This restriction is placed on I/O transfers due to the implementation-dependent nature of local storage allocation for PASCAL programs. If the allocation of storage is sequential and compatible with allocated buffers, then I/O transfers from variable space may be allowed.

4.5.5.1 READR

The READR procedure provides random access to both segmented and contiguous files. The system provides buffering for those READR request which are not aligned with disc sector boundaries.

Calling Sequence

READR (EVENTFLAG, CHANNEL, ↑BUFFER, WORDS, FILEADDRESS, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number
BUFFER	pointer to data buffer
WORDS	word count of transfer

FILEADDRESS	offset in words from beginning of file	
STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERBUFFER	invalid memory address
	ERCHANNEL	invalid channel number
	or, if EVENTFLAG = SUSPEND:	
	ERNOTOPEN	file not open
	ERFILEADDR	file address outside of file
	ERREADPROTECT	file read protected
	ERNORANDOM	not a random access device
	EREOF	end of file during read - partial read may have resulted
	ERDEV	device error detected during read
	ERDIR	for segmented files, a directory error in the Segment Pointer Block occurred

4.5.5.2 WRITER

The WRITER procedure provides random access to both segmented and contiguous files. If the file address and word count for the WRITER call reflect the physical block size of the device involved, the transfer will be performed from the user's buffer area.

Calling Sequence

WRITER (EVENTFLAG, CHANNEL, ↑BUFFER, WORDS, FILEADDRESS, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number
↑BUFFER	pointer to data buffer
WORDS	word count of transfer
FILEADDRESS	offset in words from beginning of file

STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERBUFFER	invalid memory address
	ERCHANNEL	invalid channel number
	or, if EVENTFLAG = SUSPEND:	
	ERNOTOPEN	file not open
	ERFILEADDR	file address outside of file
	ERWRITEPROTECT	file read protected
	ERNOSPACE	no space available on this device - partial write may have resulted
	ERDIR	for segmented files, a directory error in the segment pointer block occurred
	ERNORANDOM	not a random access device

4.5.5.3 READSQ

The READSQ procedure provides sequential access to both segmented and contiguous files. The sequence pointers are maintained by VTS/OS and cannot be accessed or altered by user tasks. Sequence pointers are set to the first word in the file when the file is opened.

Calling Sequence

READSQ (EVENTFLAG, CHANNEL, ↑BUFFER, WORDS, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number
↑BUFFER	pointer to data buffer
WORDS	word count

STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERBUFFER	invalid memory address
	or, if EVENTFLAG = SUSPEND:	
	ERNOTOPEN	file not open
	ERREADPROTECT	file read protected
	EREOF	end of file encountered during read - partial read may have resulted
	ERDEV	device error detected during read
	ERDIR	for segmented files, a directory error in the segment pointer block occurred

4.5.5.4 WRITESQ

The WRITESQ procedure provides sequential access to both segmented and contiguous files. The sequence pointers are maintained by VTS/OS and cannot be accessed or altered by user tasks. WRITESQ may not be used for contiguous files. The sequence pointer for sequential write operations on segmented files begins at the end of the current file, and extends the length of the file.

<u>Calling Sequence</u>

WRITESQ (EVENTFLAG, CHANNEL, ↑BUFFER, WORDS, STATUS)

<u>Parameters</u>

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number
↑BUFFER	pointer to data buffer
WORDS	word count of transfer

STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERCHANNEL	invalid channel number
	ERBUFFER	invalid memory address
	or, if EVENTFLAG = SUSPEND:	
	ERNOTOPEN	file not open
	ERWRITEPROTECT	file write protected
	ERDEV	device error detected during write
	ERDIR	for segmented files, a directory error in the segment pointer block occurred
	ERNOSEQUENTIAL	WRITESQ not allowed for contiguous files

4.5.5.5 READLN

The READLN procedure provides sequential access to both segmented and contiguous files. The sequence pointers are maintained as described under READSQ. READLN terminates the read upon finding a carriage return or a null (zero) in the data stream. The carriage return or null is transferred to the user data area.

Calling Sequence

READLN (EVENTFLAG, CHANNEL, ↑BUFFER, MAXWORDS, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number

↑BUFFER	pointer to data buffer
MAXWORDS	maximum word count of transfer
STATUS	error code:

NOERROR	
EREFLAG	invalid event flag
ERCHANNEL	invalid memory address

or, if EVENTFLAG = SUSPEND:

ERNOTOPEN	file not open
ERREADPROTECT	file read protected
EREOF	end of file encountered during read - partial read may have resulted
ERDEV	device error detected during read
ERDIR	for segmented files, a directory error in the segment pointer block occurred

STATUS.VALUE1 contains the number of words read.

4.5.5.6 WRITELN

The WRITELN procedure provides sequential access to segmented files. The sequence pointers are maintained as described under WRITESQ. WRITELN terminates the write upon finding a carriage return or a null in the data stream. The carriage return or null is written to a specified device or file.

Calling Sequence

WRITELN (EVENTFLAG, CHANNEL, ↑BUFFER, WORDS, STATUS)

Parameters

EVENTFLAG	EFLAG1..EFLAG15 or SUSPEND
CHANNEL	channel number
↑BUFFER	pointer to data buffer
WORDS	word count of transfer

STATUS	error code:	
	NOERROR	
	EREFLAG	invalid event flag
	ERCHANNEL	invalid channel number
	ERBUFFER	invalid memory address
	or, if EVENTFLAG = SUSPEND:	
	ERNOTOPEN	file not open
	ERWRITEPROTECT	file write protected
	ERDEV	device error detected during write
	ERDIR	for segmented files, a directory error in the segment pointer block occurred

4.6 ERROR REPORTING AND CONTROL

The error reporting and recovery mechanisms provided in VTS/OS rely heavily on participation of the applications tasks in detecting and verifying the occurrence of errors. This approach is taken not to place the burden of error control on the applications programs, but to provide flexibility and context sensitivity in the detection of failures. Many error conditions are possible which, under certain circumstances, do not represent failures of the system, and may even be expected by the applications task attempting the function causing the error.

Since the applications programmer is in the best position to determine the significance of error conditions, he is allowed to detect and evaluate them. VTS/OS provides the mechanism for reporting errors to a central location, and for attempting system level recovery on the basis of reported errors.

4.6.1 FAILUREDETECTED

FAILUREDETECTED allows applications programs to report unexpected errors to the Local Error Report Center (LERC) for the processor in which the task is running. Some error types may be acted upon immediately in attempting recovery; others are logged and examined over time to see if they recur, or whether they are transient.

AD-A076 501 INTERNATIONAL COMPUTING CO BETHESDA MD F/G 9/2
VESSEL TRAFFIC SERVICES PROCESSING/DISPLAY SUBSYSTEM DETAILED S--ETC(U)
JUL 79 D A COHN , F T MICKEY DOT-CG-81-78-1833
UNCLASSIFIED USCG-D-66-79 NL
2 OF 4
AD A
076501

Calling sequence -

FAILUREDETECTED (SENDERID, DESTNAME, ERRORCODE, FAILURENUMBER);

Parameters -

SENDERID -	Task id of reporting task
DESTNAME -	NIL or PROCESSNAME of the other party for ITC errors
ERRORCODE -	BADDATAPRODUCED, BADDATARECEIVED, MSGNOT-ANSWERED, UNABLETOINSTALL, UNABLE TO SCHEDULE;
FAILURENUMBER -	caller-assigned ID for this failure

5 VTS/OS DETAILED DESCRIPTION

This chapter contains the detailed description of all VTS/OS routines other than the Executive itself. Although the chapter is organized along the same lines as Chapter 4, the User's Manual, the emphasis here is on the internal workings of the routines which provide services to the users, rather than on the services themselves. In addition, many routines with which users have no direct interface are described in this chapter.

For each routine, the description given includes a brief explanation of the use of the routine, the preliminary processing required of the user interface stub (if any) that is specific to each routine, and then a detailed description of the algorithm used by the routine. Where relevant, a discussion is presented of the alternatives and considerations which led to the selection of a particular design. Finally, wherever necessary, a description is given of those system data structures with which the routine interacts, and of the nature of the interaction.

5.1 EVENT MANAGEMENT

5.1.1 Events

In VTS/OS, each task has fifteen numbered event flags, numbered from 1 (or EFLAG1) to 15 (or EFLAG15). There is in addition a sixteenth event flag, SUSPEND, which is unnumbered.

Figures 5-1A, 5-1B, and 5-1C depict the status for a single event flag other than SUSPEND, i.e., EFLAG1..EFLAG15, while Figure 5-1D depicts the status for the SUSPEND event flag.

The state of each of the event flags for a task is recorded in three entries in the Task Control Block: EFLAGSPOSSIBLE, EFLAGSACTUAL, and the array EFLAGPTR. EFLAGSPOSSIBLE AND EFLAGSACTUAL are both single-word entries, each using a bit-position index for each event flag. The low-order bit in each word corresponds to event flag 1; the next-to-high order bit corresponds to event flag 15; the high-order bit corresponds to the SUSPEND event flag. EFLAGPTR is a 16-word array, with a single entry for each event flag. The ordering is similar to that used in EFLAGSPOSSIBLE and EFLAGSACTUAL. That is, the first entry is for event flag 1; the last entry is for the SUSPEND event flag.

When a system service call is made using an event flag specifically as the EVENTFLAG subfield of the Event Control Block (note: REMOVEDELAY and GETSTATUS, in particular, have an event flag as a parameter, but it goes in the PARAM1 subfield of the ECB; hence, this discussion does not apply to them), the CALLCOMPLETE routine will set the appropriate bit in EFLAGSPOSSIBLE. This bit will not be cleared until after the system service request has completed. At that time, it will be cleared by either a GETSTATUS request for that event flag, or if it is used in satisfying either a WAITANY call or a WAITALL call.

When the system service request completes, the appropriate bit in EFLAGSACTUAL will be set. This bit will be cleared either by a

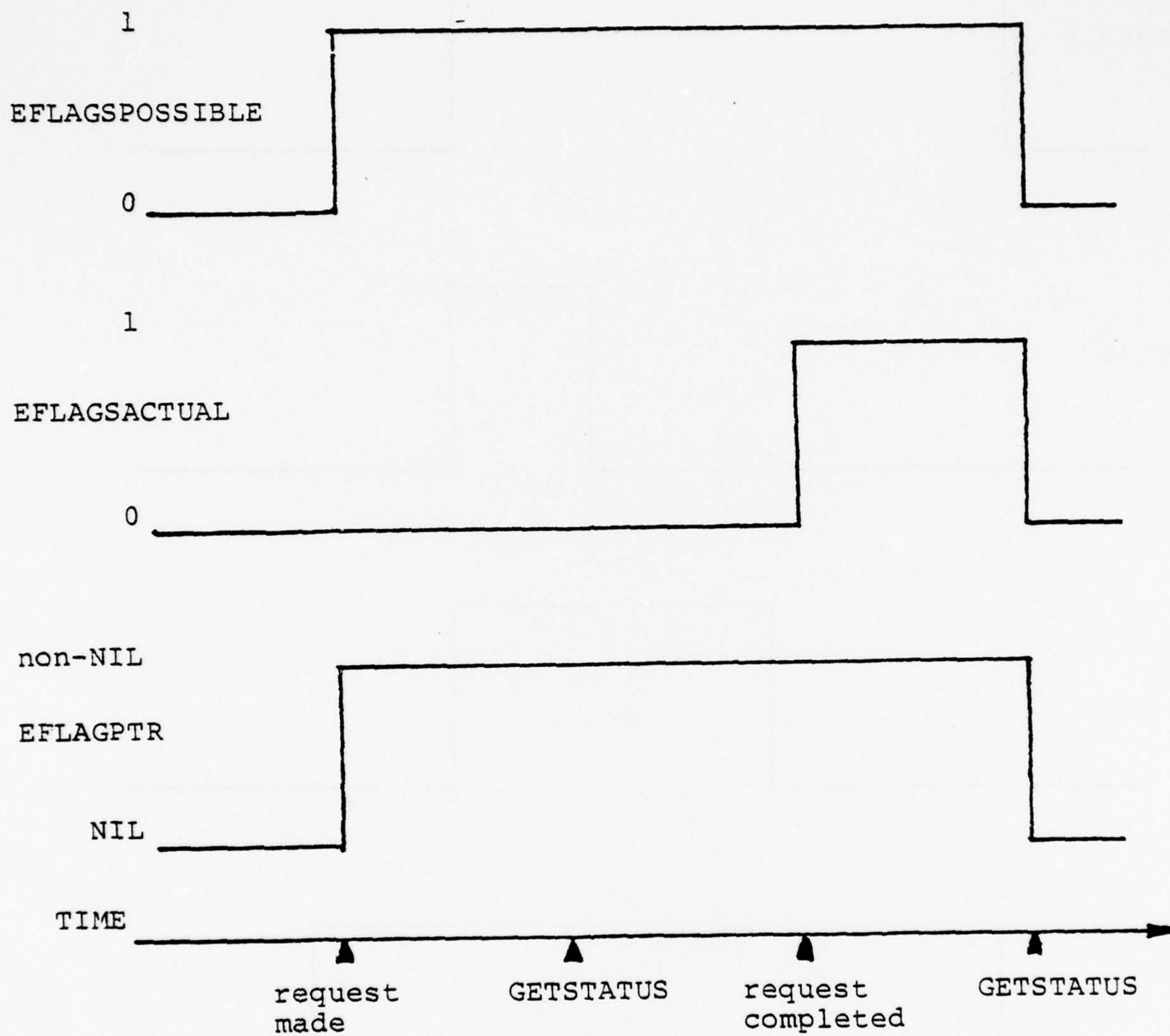


Figure 5-1A. Status for a Single Event Flag other than SUSPEND

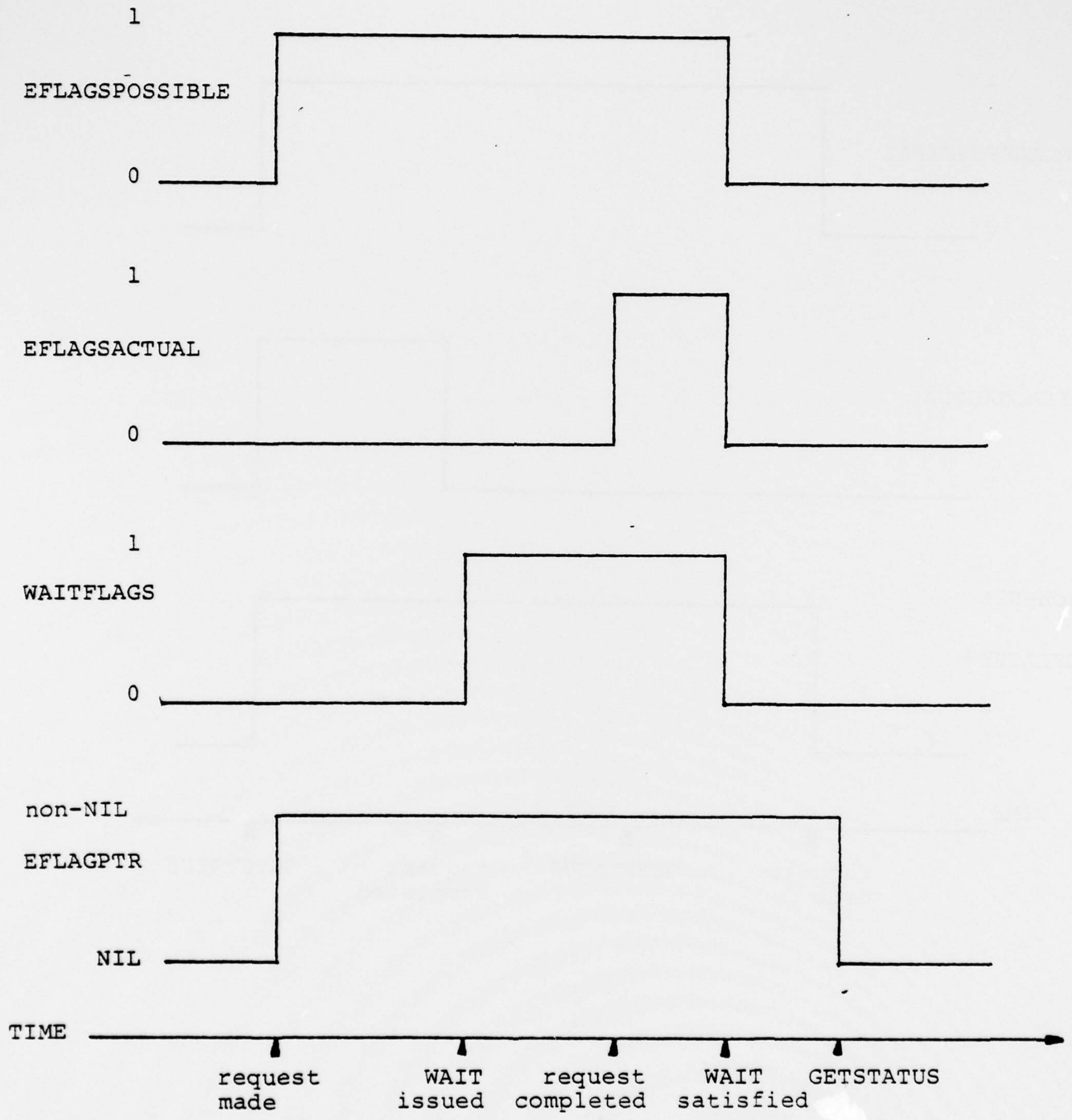


Figure 5-1B. Status for a Single Event Flag other than SUSPEND

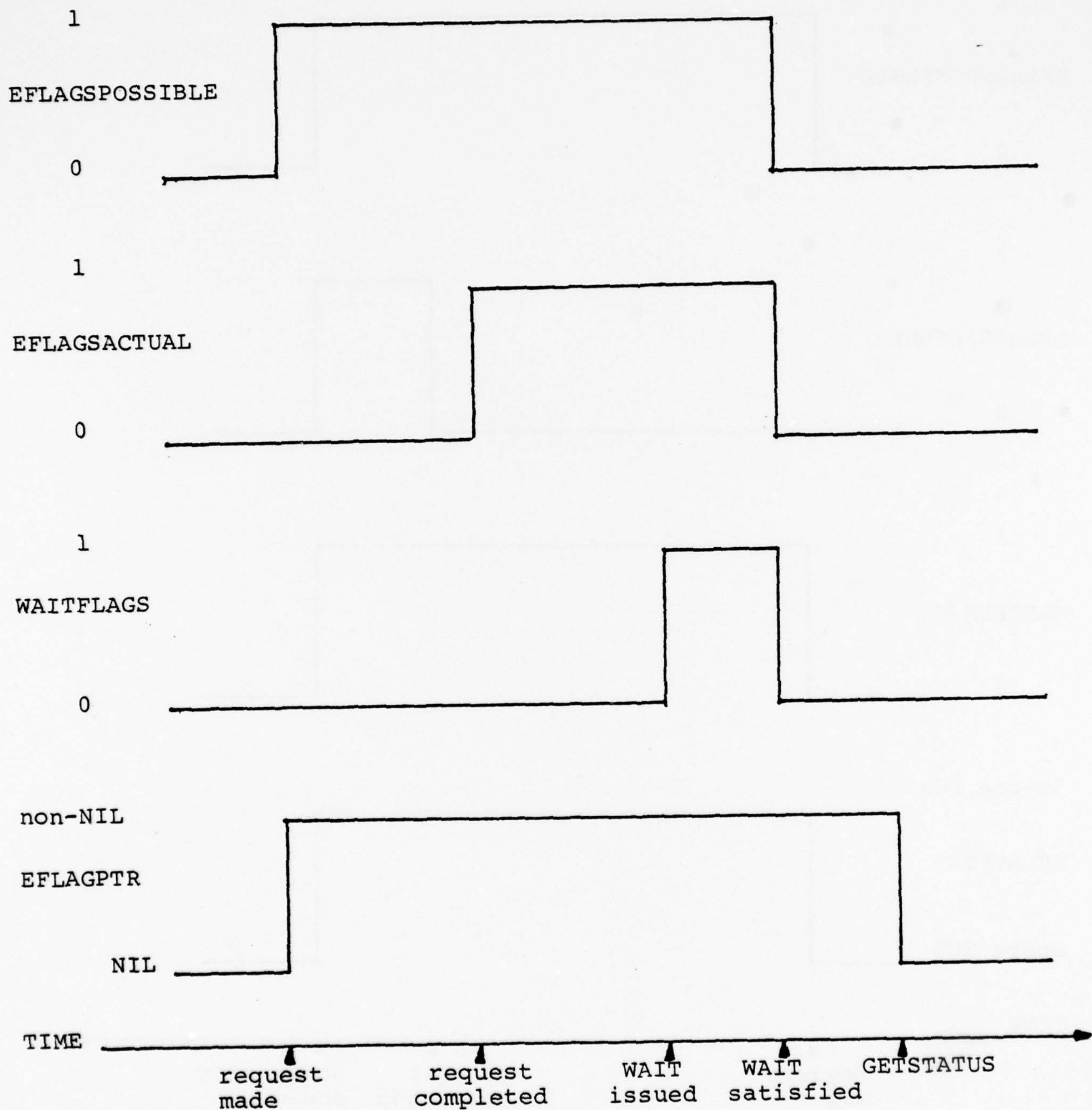


Figure 5-1C. Status for a single event flag other than SUSPEND

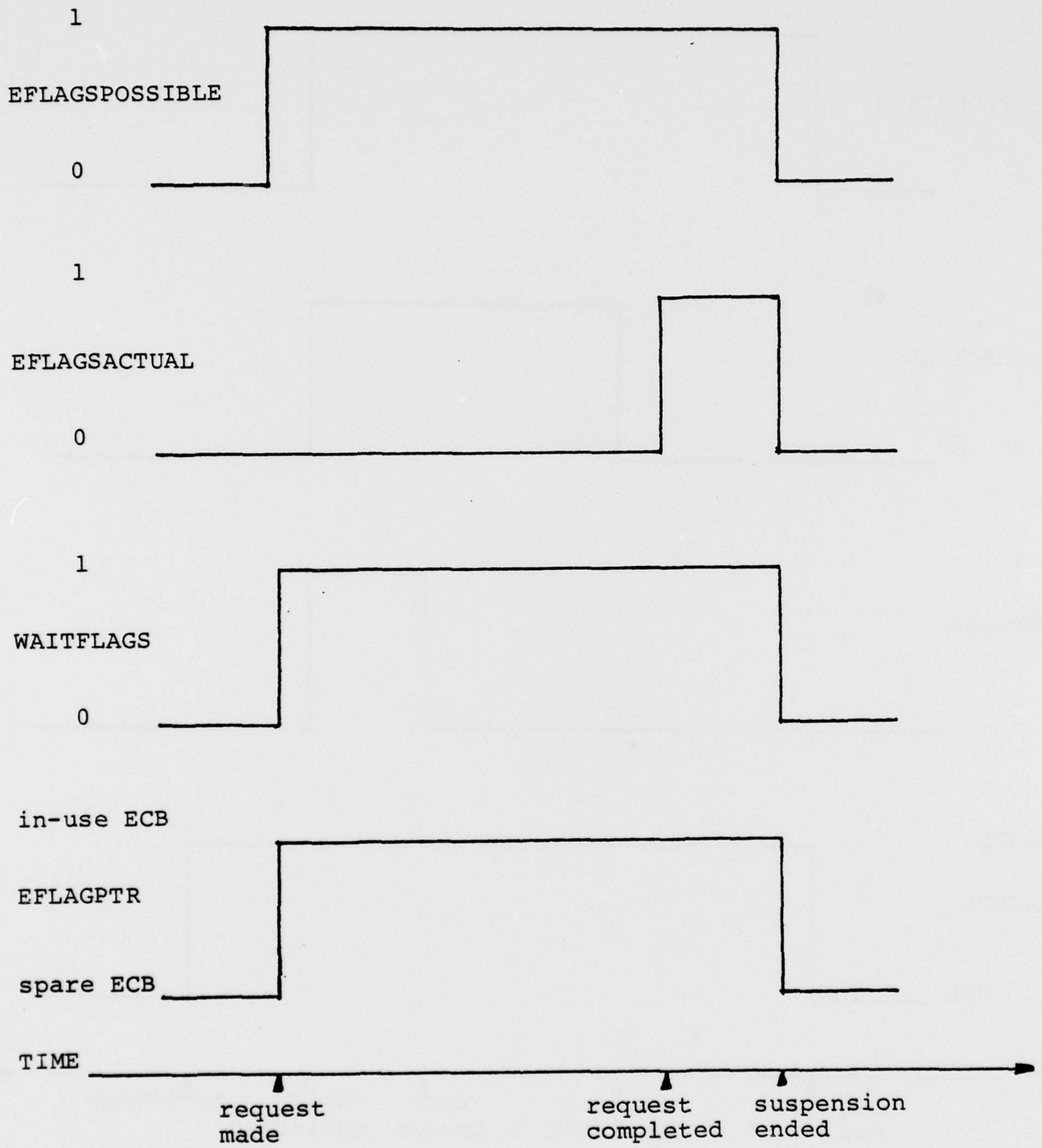


Figure 5-1D. Status for the SUSPEND event flag

GETSTATUS request for that event flag, or if that event flag is used in satisfying either a WAITANY call or a WAITALL call.

When the system service call is originally made, the CALLCOMPLETE routine will place the address of the ECB for the call in the appropriate entry of the EFLAGPTR array. This entry will not be cleared (actually, set to NIL) until a GETSTATUS call is made that specifies that event flag after the associated system service request has completed.

If a task is not suspended specifically by issuing a system service request with SUSPEND as the event flag, it always has a "spare" ECB pre-allocated to it. The address of the ECB is in the SUSPEND entry of the EFLAGPTR array. This does not indicate that the SUSPEND event flag is in use. The SUSPEND event flag is in use only while the task is actually on the Suspend queue with a WAITTYPE of SUSPEND.

In discussing event flags, one other entry in the Task Control Block, WAITFLAGS, must be examined. The WAITFLAGS entry is used while the task is actually on the Suspend queue and waiting for one or more events. If the task is SUSPENDed, WAITFLAGS will have the bit-position index for the SUSPEND event flag. If the task is waiting as the result of either a WAITANY or a WAITALL call, bit-position indices of the event flags that may be used to satisfy that call will be kept in WAITFLAGS.

5.1.2 Time-Interval Units

5.1.2.1 SETDELAY

GENERAL

This routine is used by a task to set a delay timer, whose expiration will cause a specified event to occur. This may be used by

a task for watch-dogging various critical events, such as child completion, or for periodic rescheduling of itself. Most system service calls will have a built-in watchdog timer, so that tasks need not explicitly time system service calls.

USER INTERFACE STUB

The user interface stub will set up the following fields in the Event Control Block:

EVENTFLAG - user's EVENTFLAG parameter
STATUSPTR - address of user's STATUS variable
PARAM1 - number of seconds, from user's call

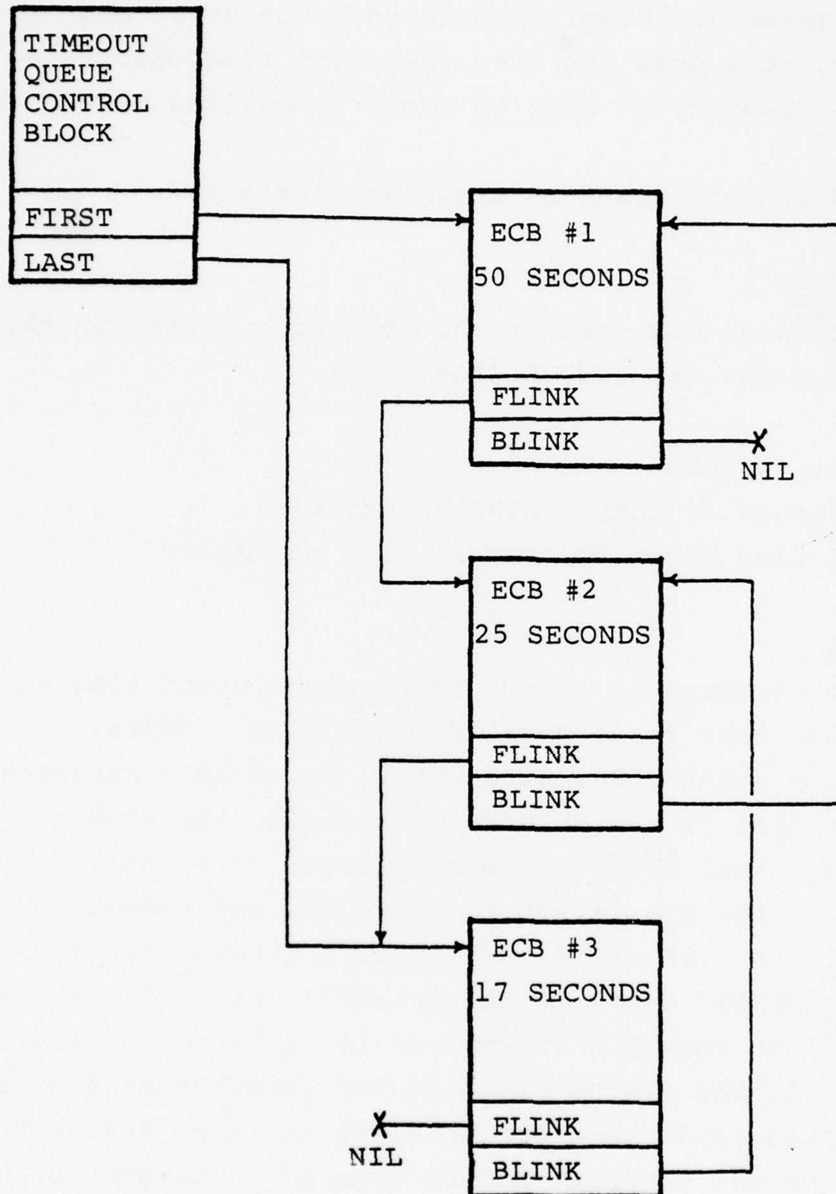
SPECIFIC PROCESSING

This routine first checks the PARAM1 of the ECB to determine whether or not the number of seconds for which the user wishes to delay is valid (i.e., positive, but less than or equal to MAXSEC). If the value is not legal, the STATUS in the ECB is set to ERR-TIME, and SETDELAY exits. If the value is legal, the routine traverses the chain of timeout ECB's, each of which contains in its PARAM1 field the number of seconds that must elapse after the completion of the previous timeout, summing these inter-arrival times into a running total of the number of seconds to elapse until completion of each ECB. If this total becomes greater than the PARAM1 field of the SETDELAY ECB, it links the SETDELAY ECB into the Timeout queue before the last entry scanned, and then adjusts the PARAM1 fields of the SETDELAY ECB and the ECB after it. If it reaches the end of the queue without this total exceeding PARAM1 of the SETDELAY ECB, it places the SETDELAY ECB at the end of the queue and adjusts its PARAM1 field accordingly. It then sets the STATUS in the ECB to NO-ERROR and exits.

5.1.2.2 REMOVEDELAY

GENERAL

This routine may be used by a task to cancel an outstanding timeout request, if that request has not yet completed. (See Figure 5-2 for the timeout list and a deletion example). The original time-out is cancelled by



ECB #1 will be signalled for completion in 50 seconds;
ECB #2 will be signalled for completion in 50+25 or 75 seconds;
And ECB #3 will be signalled for completion in 50+25+17 or 92 seconds.

If timeout #2 is deleted, the timeout value in ECB #3 must have 25 seconds added to it, so that it will still be signalled for completion in 50+42 or 92 seconds.

Figure 5-2. Timeout List and Deletion Example

forcing it to an immediate completion, with a status of ERR-DELETED. However, this does not free the event flag associated with the original timeout; it must be freed by calling GETSTATUS.

The REMOVEDELAY structure chart is shown in Figure 5-3.

USER INTERFACE STUB

The user interface stub will set up the following fields in the Event Control Block for the REMOVEDELAY call:

EVENTFLAG - set to zero
STATUSPTR - address of user's STATUS variable
PARAM1 - event flag given by user as call parameter

SPECIFIC PROCESSING

REMOVEDELAY first performs an error check on the event flag given in PARAM1 to ensure that it is a valid event flag. (Note: although this is an event flag, since it is given as a parameter in the ECB rather than as the EVENTFLAG subfield, the stub has not performed the normal EVENTFLAG error check). If invalid, the STATUS field in the ECB is set to ERR-FLAG, and execution proceeds at label 100. If valid, REMOVEDELAY obtains the ECB pointed to by the PARAM1 entry in the EFLAGPTR array of the Task Control Block. If no such ECB exists, or if it is not an ECB for a SETDELAY call, the STATUS field in the REMOVEDELAY ECB is set to ERR-WRONGFLAG, and execution proceeds at label 100. Otherwise, the STATUS of the SETDELAY ECB is checked to determine whether or not it has completed. If it has completed, the STATUS of the REMOVEDELAY ECB is set to ERR-TOOLATE, and execution proceeds at label 100. If it has not completed, the SETDELAY ECB is removed from the Timeout queue and its STATUS is set to ERR-DELETED. If a subsequent timeout ECB occurred (see Figure 5-4, TIMEOUTASYNC), that ECB's PARAM1 field is updated to reflect the new inter-arrival time. The SETDELAY ECB is then passed to the EVENTCOMPLETE routine to force immediate completion. Finally, the STATUS field of the REMOVEDELAY ECB is set to NO-ERROR, and execution proceeds at label 100. At label 100, REMOVEDELAY exits.

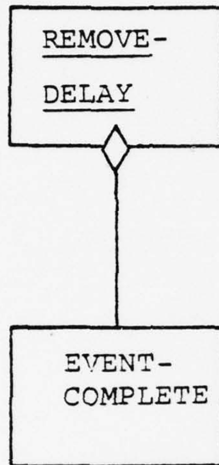


Figure 5-3. REMOVEDELAY

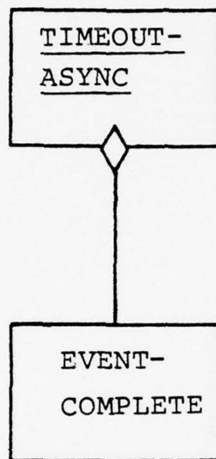


Figure 5-4. TIMEOUTASYNC

5.1.2.3 PAUSE

GENERAL

This routine enables the calling task to relinquish the CPU in order to avoid monopolizing it for long periods during extended computational sequences. It does not cause any change in the status of the task.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for this call in the following manner:

EVENTFLAG - set to zero
STATUSPTR - set to NIL

SPECIFIC PROCESSING

The Task Control Block for the calling task is delinked from the head of the Ready queue, and linked to the tail of the Ready queue. The STATUS in the ECB is set to NO-ERROR.

5.1.3 Event-terminated Waits

5.1.3.1 WAITANY

GENERAL

This routine enables the calling task to suspend itself until the occurrence of any one of a set of events. Upon the return to the calling task, the condition that led to termination of

suspension will be indicated to the task. This routine will always cause the calling task to relinquish the CPU.

This routine uses bit-position-indices to specify events. In each of the two parameters, the low-order bit corresponds to event flag 1; the next-to-high-order bit corresponds to event flag 15.

The task specifies the set of events in which it is interested by OR'ing together the bit-position index for each event and passing this combination of indices as the first parameter, SETOFEVENTS. If the suspension is terminated because an event occurred, the bit-position index for that event will be returned in the second parameter, OCCURREDEVENT.

There are three conditions which can lead to termination of the suspension. The first possibility is that none of the events specified in SETOFEVENTS is capable of occurring. In this case, the task will be placed at the end of the Ready queue, and OCCURREDEVENT will contain a zero. The second possibility is that one or more of the events specified in SETOFEVENTS has already occurred at the time of the WAITANY call. In this case, the task will be placed at the end of the Ready queue, and OCCURREDEVENT will contain the bit-position index of the lowest-numbered event. Note that this will not necessarily be the first of the events to have occurred. This places an implicit priority upon event flags: the lowest-numbered event flag will be reported first. The third and final possibility is that none of the events specified in SETOFEVENTS has yet occurred, but that at least one of these events can occur. In this case, the task will be placed on the Suspend queue until any of these events occurs. At that time, the task will be placed on the Ready queue, and OCCURREDEVENT will contain the bit-position index of the event.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for this call as follows:

EVENTFLAG - set to zero
STATUSPTR - set to NIL
PARAM1 - contains SETOFEVENTS
PARAM2 - contains the address (in the user's address space) of OCCURREDEVENT

SPECIFIC PROCESSING

The STATUS field in the ECB is set to NO-ERROR, since no real error is possible with this call. The set of events specified in PARAM1 is then AND'ed against those events that can possibly occur, as specified by the EFLAGSPOSSIBLE field in the Task Control Block. If this is zero, OCCURREDEVENT is set to zero, and the PAUSE routine (q.v.5.1.2.3) is called to move this task to the end of the Ready queue. If it is not zero, a check is made to see if any of these possible events has already occurred. If none has occurred, the TCB is moved from the Ready queue to the Suspend queue, the WAITTYPE is set to WAITANY, and PARAM2 is saved in the TCB. If one or more has occurred, a check is made to identify the lowest-numbered event. Its index is returned in OCCURREDEVENT. The index corresponding to that event is cleared in EFLAGSACTUAL and EFLAGSPOSSIBLE so that a subsequent call to WAITANY will not return that same event. Finally, the PAUSE routine is called to move this task to the end of the Ready queue.

5.1.3.2 WAITALL

GENERAL

This routine enables the calling task to suspend itself until the occurrence of all possible events in a specified set of events. Upon the return to the calling task, the condition that

lead to termination of the suspension will be indicated to the task. This routine will always cause the calling task to relinquish the CPU.

This routine uses bit-position-indices to specify events. In each of the two parameters, the low-order bit corresponds to event flag 1; the next-to-high-order bit corresponds to event flag 15.

The task specifies the set of events in which it is interested by OR'ing together the bit-position index for each event and passing this combination of indices as the first parameter, SETOFEVENTS. If the suspension is terminated because some of the events have occurred, the bit-position indices for those events will be returned in the second parameter, OCCURREDEVENT.

Three conditions can lead to termination of the suspension. The first possibility is that none of the events specified in SETOF-EVENTS is capable of occurring. In this case, the task will be placed at the end of the Ready queue, and OCCURREDEVENT will contain a zero. The second possibility is that all of the events specified in SETOFEVENTS that are capable of occurring have already occurred at the time of the WAITALL call. In this case, the task will be placed at the end of the Ready queue, and OCCURREDEVENT will contain the bit-position indices of all of those events. The third possibility is that not all of the events specified in SETOFEVENTS that are capable of occurring have occurred. In this case, the task will be placed on the Suspend queue until all such events have occurred. At that time, the task will be placed on the Ready queue, and OCCURREDEVENT will contain the bit-position indices of all of these events.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for this call as follows:

EVENTFLAG - set to zero
STATUSPTR - set to NIL
PARAM1 - contains SETOFEVENTS
PARAM2 - contains the address (in the user's address space) of OCCURREDEVENT

SPECIFIC PROCESSING

The STATUS field in the ECB is set to NO-ERROR, since no real error is possible with this call. The set of events specified in PARAM1 is then AND'ed against those events that can possibly occur, as specified by the EFLAGSPOSSIBLE field in the Task Control Block. If this is zero, then OCCURREDEVENT is set to zero, and the Pause routine (q.v.5.1.2.3) is called to move this task to the end of the Ready queue. If it is not zero, a check is made to see if all of the possible events have already occurred. If all have occurred, the indices of all the events are returned in OCCURRED-EVENT, and PAUSE is called to move the task to the end of the READY queue. In addition, the indices corresponding to those events events are cleared in EFLAGSACTUAL and EFLAGSPOSSIBLE. If not all have occurred, the TCB is moved from the Ready queue to the Suspend queue, the WAITTYPE is set to WAITALL, and PARAM2 is saved in the TCB.

5.1.3.3 GETSTATUS

GENERAL

This call is used by a task to obtain the status of an outstanding or completed service call. If the service call is completed, this call will also free the event flag associated with the service call, thereby allowing it to be used for subsequent calls.

USER INTERFACE STUB

The user interface stub will set up the following fields of the Event Control Block for this call:

EVENTFLAG - set to zero

STATUSPTR - address (in user's address space) of user's STATUS variable

PARAM1 - event flag specified by call as first parameter

SPECIFIC PROCESSING

GETSTATUS first performs an error check on the event flag given in PARAM1 to ensure that it is a valid event flag. (Note: although this is an event flag, since it is given as a parameter in the ECB rather than as the EVENTFLAG subfield, the stub has not performed the normal EVENTFLAG error check.) If invalid, the STATUS in the ECB is set to ERR-EFLAG, and execution proceeds at label 100. If valid, the entry in the EFLAGPTR array in the TCB specified by PARAM1 is checked to see whether or not a service call is associated with that event flag. If not, the STATUS in the ECB is set to ERR-FLAG-NOT-IN-USE, and execution proceeds at label 100. If so, the status from the ECB pointed to by the EFLAGPTR array entry is placed in the STATUS field of the GETSTATUS ECB. The former ECB is then checked to determine whether or not its service call has completed. If not, execution proceeds at label 100. If so, the bit-position indices corresponding to the event flag are cleared in EFLAGSPOSSIBLE and EFLAGSACTUAL in the Task Control Block. The entry in the EFLAGPTR array is also cleared, and the ECB is returned to the system free-ECB queue. Execution now proceeds at label 100, where the STATUS field from the GETSTATUS ECB is moved to the user's address space.

5.2 PROGRAM AND TASK MANAGEMENT

5.2.1 Program Management Calls

5.2.1.1 INSTALL

GENERAL

This routine is used by a task to cause a specified program to be loaded into the processor in which the task is running. If possible, all the fixed resources required by the program will be assigned to this routine, and it will be added to the list of programs available in that processor.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for the INSTALL call as follows:

EVENTFLAG - the event flag parameter of the call
STATUSPTR - the address of the user's STATUS variable
PARAM1 - the ASCII name of the proram to be INSTALLED (note: this will probably require more than one word of parameter space)

SPECIFIC PROCESSING

The synchronous INSTALL structure chart is shown in Figure 5-5. This routine simply enqueues the ECB for the asynchronous routine to operate on and then exits to the main Exec loop.

The asynchronous routine of INSTALL has been waiting for an entry to appear on the queue which drives it. When the synchronous routine enqueues the ECB, the asynchronous routine begins execution, competing for the CPU in exactly the same manner as the applications tasks.

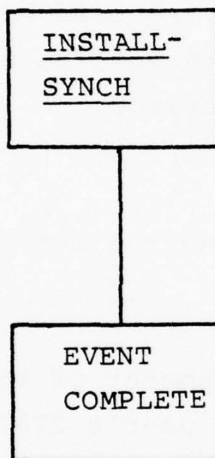


Figure 5-5. Synchronous INSTALL

The first error check performed by the asynchronous routine is to determine whether or not this program has already been installed. For this, it searches the queue of active Image Control Blocks, comparing the TASKNAME field with PARAM1 from the ECB. If a match is found, the STATUS in the ECB is set to ERR-TAI, and execution proceeds at label 200. If no match is found, a check is made to determine whether a free ICB is available. If not, the STATUS in the ECB is set to ERR-NMR, and execution proceeds at label 200; if so, the ICB is captured for subsequent use.

INSTALL then attempts to open the file containing the program. If an error is detected, the STATUS in the ECB is set to ERR-IO, the second word of the STATUS is set to the actual error code returned by OPEN, the ICB is returned to the free list, and execution proceeds at label 200. If no error is detected, the header is read to determine the size of the program root. If an error occurs in the READ, the STATUS in the ECB is set to ERR-IO, the second word of the STATUS is set to the actual error code returned by READ, the ICB is returned to the free list, a CLOSE is issued for the file, and execution proceeds at label 100. If no error occurred, an attempt is made to obtain sufficient memory for the root. If no memory is available, a STATUS of ERR-NMR is returned, all resources are returned (i.e., the ICB is returned and the file CLOSED), and execution proceeds at label 100. If the memory is available, as much of the ICB is filled in as is possible.

A READCODE is then issued to read in the root code. If an error occurs, STATUS is set to ERR-IO, all resources are returned, and execution proceeds to label 100. If no error occurs, a check is made to see whether or not there are any overlays. If so, a loop is entered in which an attempt is made to get memory for an overlay and read in that overlay. At any point in the loop, if an error occurs, the STATUS is set appropriately, all resources are returned, and execution proceeds at label 100.

If the loop is successfully executed, the ICB is linked to the queue of in-use ICBs. The file is then CLOSEd and execution proceeds at label 100 where a wait is performed. This is immediately ended if the file has never been opened. After the wait is satisfied, execution proceeds at label 200, where the ECB is sent to the EVENTCOMPLETE routine to signal completion to the calling task.

INSTALL also uses several subordinate routines, which are shown in Figure 5-6 and described briefly here. GETROOTMEM acquires real memory for the root if any is available, and then calls FILLREGS2 to set up the map registers for the root. A STATUS of either ERR-NMR or NO-ERROR is then returned. FILLREGS2 sets up map registers in the ICB. OVERLAYPROC is the routine that performs the actual processing loop in which real memory and an Overlay Control Block are obtained and the overlay code is read into memory. A STATUS of either ERR-NMR, ERR-IO, or NO-ERROR is then returned. It actually calls GETOLAYMEM, which gets the memory and in turn calls FILLREGS3 to set up map registers in the OCB. RETURNALLRESOURCES is called, if necessary, to return the ICB, any OCBs, and real memory to the system free-lists.

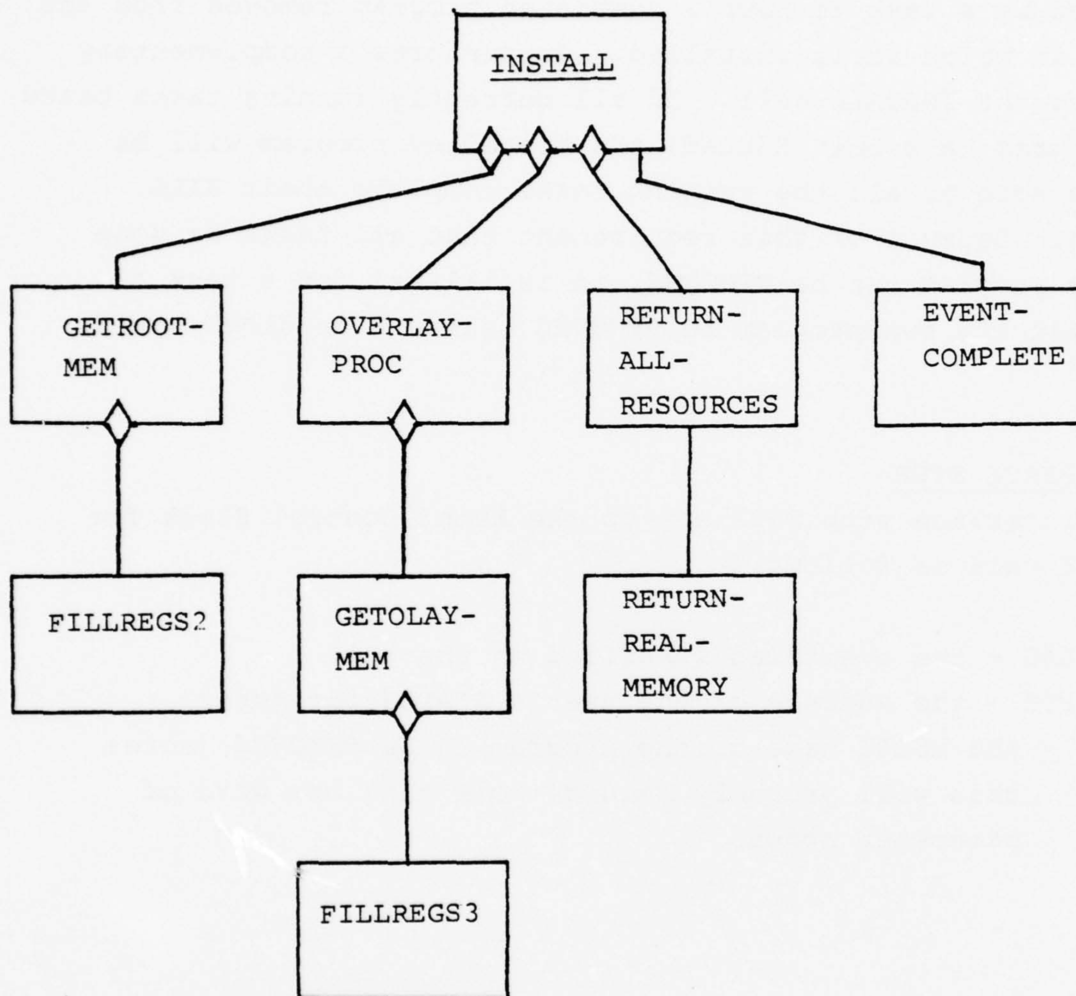


Figure 5-6. INSTALL

5.2.1.2 REMOVE

GENERAL

The REMOVE structure chart is shown in Figure 5-7. This routine may be used by a task to have a specified program removed from the processor in which it is installed. It performs a complementary function to the INSTALL call. If all currently running tasks based on the program have been KILLed, the specified program will be REMOVEd as soon as all the running tasks complete their KILL processing. Because of this requirement that all tasks be gone before the program can be REMOVEd, it is illegal for a task to request that its own program be REMOVEd, since a deadlock would result.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for the REMOVE call as follows:

EVENTFLAG - the eventflag specified in the call
STATUSPTR - the address of the user's STATUS parameter
PARAM1 - the ASCII name of the program to be REMOVEd (note: this will probably require more than one word of parameter space)

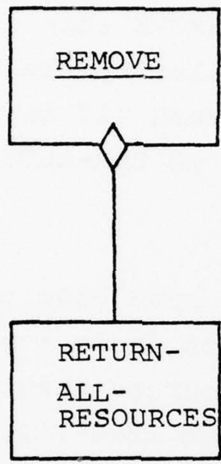


Figure 5-7. REMOVE

SPECIFIC PROCESSING

REMOVE begins processing by attempting to find the ICB for the program that is to be removed. If one cannot be found, REMOVE sets the STATUS in the ECB to ERR-TNI, and execution continues at label 5. Having located the ICB, REMOVE then checks to ensure that the task is not attempting to remove its own program. If so, REMOVE sets the STATUS to ERR-SUICIDE, and execution continues at label 5. If not, REMOVE then checks each of the tasks running the program controlled by the ICB to ensure that a KILL has been issued for each task (if any). If this is not the case, REMOVE sets the STATUS to ERR-ALIVE, and execution continues at label 5.

If all of the above error checks have been passed, REMOVE now checks to determine whether or not there are any running copies of the program. If not, all resources associated with this program are returned to the system free-lists, STATUS is set to NO-ERROR, and execution continues at label 5. When running copies are detected, STATUS is set to REQUEST-IN-PROGRESS, the address of the ECB is put in the REMOVEECBPTR field of the ICB (which indicates that a REMOVE is in progress), and execution continues at label 5. At label 5, REMOVE exits to the main Exec loop.

5.2.2 Task Management Calls

5.2.2.1 SCHEDULE

GENERAL

This call is used by a task to create and execute a task using the specified program. The invoking task may make the invoked task either a child or completely independent. The invoked task will then be placed on the Ready queue, where it will operate exactly like all the other tasks already running.

USER INTERFACE STUB

The user interface stub will set up the following fields in the Event Control Block:

EVENTFLAG - the event flag parameter from the call

STATUSPTR - the address of the user's STATUS variable

PARAM1 - the ASCII name of the program (note: this may require more than one word)

PARAM2 - the address of the user's TASKID variable

SPECIFIC PROCESSING

SCHEDULE begins processing by attempting to locate the ICB of the program whose name is in PARAM1. If it cannot be found, SCHEDULE sets the STATUS to ERR-TNI, and execution continues at label 5. If it is found, SCHEDULE then checks the REMOVEECBPTR field of the ICB. If this field is not NIL, it contains the address of an ECB from a previous REMOVE call, in which case SCHEDULE sets the STATUS to ERR-PERMIT, and execution continues at label 5.

After the above checking, SCHEDULE attempts to get a Task Control Block, an Event Control Block, and a Buffer Control Block. If it cannot get one of each, it sets the STATUS to ERR-NMR, and execution continues at label 5. If all of the above have been obtained, SCHEDULE sets up part of the TCB and BCB, and then attempts to get sufficient memory for the task's variable space. If the memory is not available, SCHEDULE returns the TCB, ECB, and BCB, sets the STATUS to ERR-NMR, and execution continues at label 5. If the memory is available, SCHEDULE fills in the rest of the TCB. If the invoked task is to be a child of the invoking task, the family linkages are set up (i.e., a pointer to the parent, connection to the brother queue, and saving the address of the SCHEDULE ECB in the POSTECB field of the TCB). It also places the newly assigned task ID in the caller's address space, using PARAM2. Finally, the new TCB is connected to the Ready queue. At label 5, SCHEDULE exits to the main Exec loop.

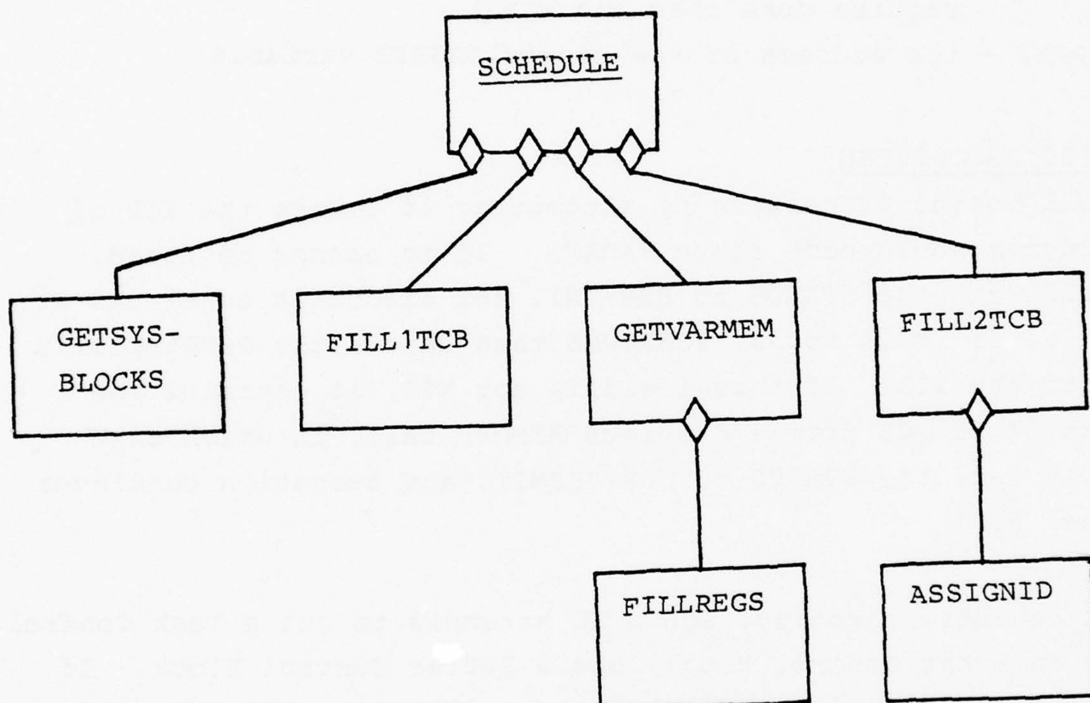


Figure 5-8. SCHEDULE

SCHEDULE uses several subordinate routines, which are shown in Figure 5-8 and described briefly here. The first is GETSYSBLOCKS, which acquires a blank TCB, ECB, and BCB, only if all three are available. A status of either NO-ERROR, or ERR-NMR is then returned. The next procedure is FILL1TCB, which sets up some fields in the TCB and BCB, but does not set up any fields that interact with any other currently in-use system data structure entries. The third procedure is GETVARMEM, which gets real memory for the task variable space, if it is available, and has the map registers in the BCB filled in using FILLREGS. It returns a status of either NO-ERROR, or ERR-NMR. The next procedure is FILL2TCB, which sets up all remaining fields in the TCB (with the exception of the family linkage fields), and updates the appropriate entries in the relevant ICB and BCB. At this point, since all system resources have been successfully allocated, the TCB can be linked to other in-use system data structure entries. The final routine is ASSIGNID, which assigns a task ID for the task which is unique within the processor.

5.2.2.2 POST

GENERAL

The POST structure chart is shown in Figure 5-9. This routine is used by a child task to communicate three words of data to its parent. The significance of the three words must be mutually decided upon by the two tasks. Since the parent obtains this information as the STATUS of its SCHEDULE call, care should be taken to avoid placing in the first word of the STATUS any value that is also one of the possible error codes resulting from the STATUS call. A child may only POST to its parent once.

USER INTERFACE STUB

The user interface stub will set up the POST Event Control Block as follows:

EVENTFLAG - set to zero
STATUSPTR - set to address of calling task's STATUS variable
PARAM1 - address of the calling task's DATAWORDS array.

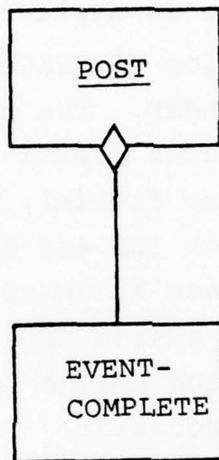


Figure 5-9. POST

SPECIFIC PROCESSING

POST first checks the TCB of the calling task to determine whether or not it has a parent. If the PARENT field of the TCB is NIL, POST sets the STATUS in the ECB to ERR-INDEP and exits to the main Exec loop. If the calling task has a parent, POST then checks the POSTECB field of the calling task's TCB. If this is NIL, the calling task has already POSTed once. In this case, POST sets the STATUS to ERR-AUU and exits to the main Exec loop. If all of the above error checks are passed, POST then takes three words from the calling task's address space, as indicated by PARAM1, and places them in the ECB pointed to by the POSTECB field of the TCB. This is the SCHEDULE ECB used when the parent SCHEDULEd this task. EVENTCOMPLETE is then called to signal completion to the parent task. It sets the POSTECB field of the calling task's TCB to NIL, so that the ECB will not be re-used. Finally, the STATUS field of the ECB for the POST call is set to NO-ERROR, and POST exits to the main Exec loop.

5.2.2.3 KILL

GENERAL

This routine is used by a task to stop a "victim" who must be either the calling task itself or a first-generation descendant of the calling task. In either case, every descendant of the "victim" will also automatically be stopped as a result of this call.

USER INTERFACE STUB

The user interface stub will set up the KILL Event Control Block as follows:

EVENTFLAG - taken from the call

STATUSPTR - address of the caller's STATUS variable

PARAM1 - the TASKID parameter from the call

SPECIFIC PROCESSING

KILL begins processing by checking the TASKID value in PARAM1 of the KILL ECB. If zero, the "victim," whose TCB address will be kept in ROOTTCB, is the calling task itself. In this case, the SUICIDE subfield of the KILL ECB is set to TRUE, and the KILLCNT subfield of the KILL ECB is set to 0. (Note: the KILL routine uses some of the available workspace in the KILL ECB for internal record keeping. The SUICIDE subfield indicates whether or not the calling task is KILLing itself; the KILLCNT subfield contains a count of the number of tasks to be KILLed, excluding the calling task if SUICIDE is true.) If PARAM1 is not zero, KILL examines the TCB's of each of the calling task's children in order to identify the "victim." If no match is found, KILL sets STATUS to ERR-NYC, and execution continues at label 200, where it returns to the main Exec loop. If a match is found, KILL sets the address of the "victim" TCB in ROOTTCB, sets the SUICIDE subfield to FALSE, and sets the KILLCNT subfield to 1. Once the KILLECB is set up, KILL then sets the ORIGINALCALL subfield to TRUE, and calls the ROOTKILL procedure. Upon the return from ROOTKILL, KILL examines the KILLCNT subfield of the KILLECB. If it has gone to zero, every task has been KILLed (if SUICIDE is false), or else every task except the calling task has been KILLed (if SUICIDE is true). If so, KILL sets the STATUS to NO-ERROR and exits to the main Exec loop where CALLCOMPLETE will indicate that the request has completed. If KILLCNT has not gone to zero, the number of tasks that are now in the process of running down are indicated by the count. In this case, KILL sets the ORIGINALCALL subfield to FALSE, so that when the last task has run down, one of KILL's subordinate procedures will call EVENTCOMPLETE to indicate that the request has completed. KILL then returns to the main Exec loop.

KILL uses several subordinate routines which are shown in Figure 5-10 and described briefly here. The first of these is ROOTKILL. This routine sets the address of the KILLECB in the KILLECBPTR subfield of the victim's TCB (thereby indicating that a KILL for this task is in progress), and then calls DESCKILL to begin a recursive descent algorithm which will initiate a KILL of each of the victim's descendants. Finally, it calls FAKEWAIT, which sets up the victim's TCB for an event run-down.

The routine DESCKILL is very similar to ROOTKILL, except that DESCKILL, when called with a TCB, sets up an event run-down for each descendant and each sibling of that TCB. DESCKILL calls itself in order to perform the recursive descent of the TCB tree.

The FAKEWAIT routine examines a single TCB to determine whether the task has any oustanding events, or owns a shared buffer in which a descendant has any outstanding I/O. If neither of these conditions is true, FAKEWAIT calls DISMEMBER, which actually returns all task resources to the system free-lists. If the task has outstanding events, or if it owns a shared buffer in which a descendant has outstanding I/O, FAKEWAIT simply places the TCB on the Suspend queue. It also sets up a WAITALL call for all outstanding events, if any.

DISMEMBER first removes the TCB which is passed to it from its ICB's queue of all tasks on the ICB. It then checks to see whether this allows an outstanding REMOVE of the program to be completed. If so, it completes the REMOVE, and then returns to processing the TCB itself. First, any ECBs are returned to the ECB free-list. Next, the POSTECB subfield is examined to determine whether this task's parent is waiting for a POST operation from this task. If so, the STATUS of the ECB identified by the POSTECB subfield is set to ERR-CHILD, and EVENTCOMPLETE is called to notify the parent.

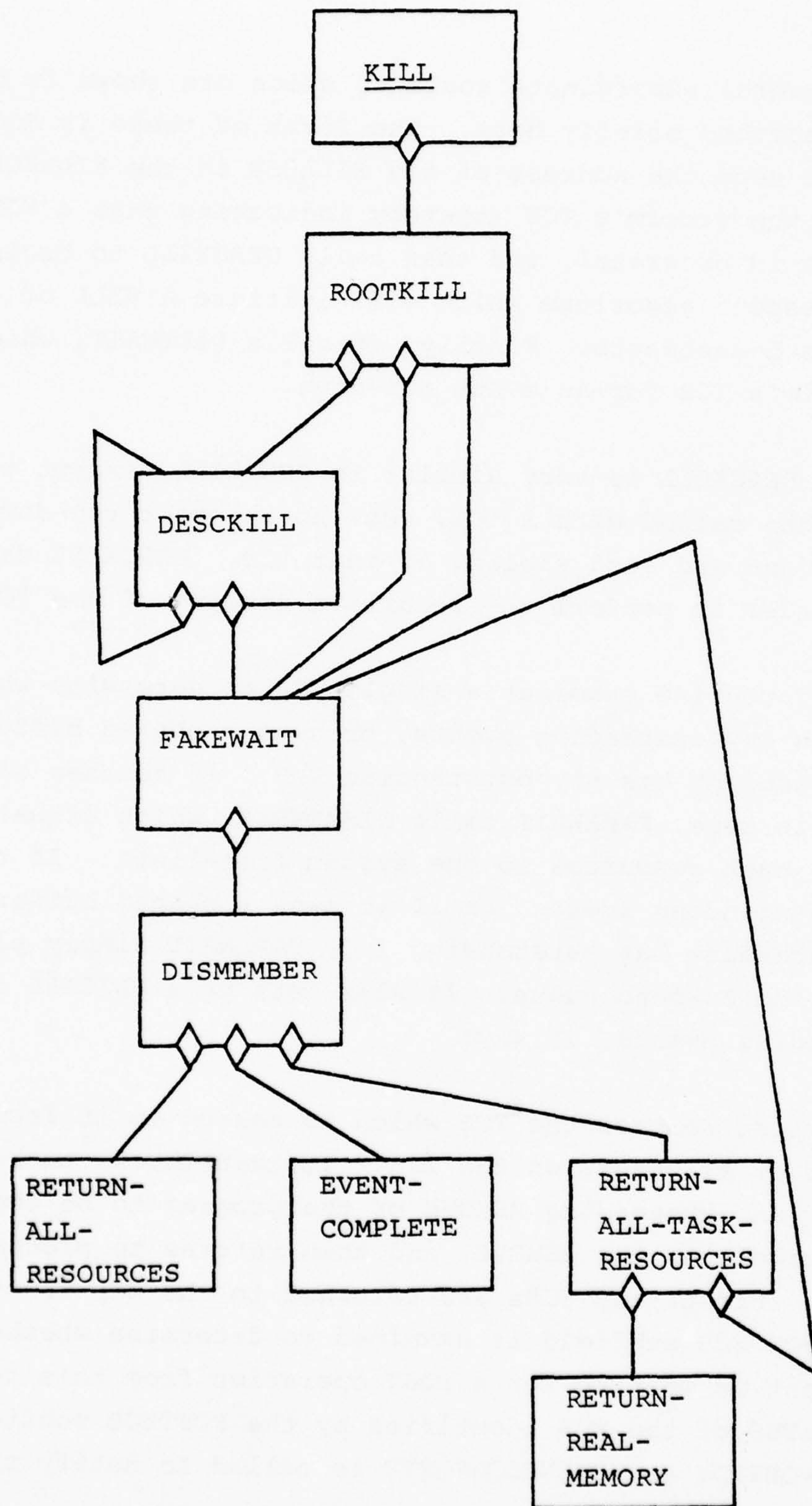


Figure 5-10. KILL

Next, DISMEMBER decrements the KILLCNT subfield of the KILL ECB. If this has now gone to zero and ORIGINALCALL is false, DISMEMBER calls EVENTCOMPLETE to indicate completion of the original KILL request. Thus, if SUICIDE was true, the calling task will now receive completion notification for its KILL request since no descendant tasks are left. In a SUICIDE, therefore, the calling task will always be the last task to complete. Finally, DISMEMBER calls RETURNALLTASKRESOURCES.

RETURNALLTASKRESOURCES returns all BCBs, any buffers which the task owned, and the TCB. In addition, if the task had a BCB for a shared buffer, but did not own that buffer and there is no more I/O in that buffer, RETURNALLTASKRESOURCES passes the buffer owner's TCB to FAKEWAIT, which will examine that TCB to determine whether or not the event run-down is complete. While this call to FAKEWAIT may be redundant, it is the only way to force examination of the owner's TCB if the owner has no outstanding events.

5.3 MEMORY MANAGEMENT

5.3.1 System Memory Control

Within VTS/OS, real memory is divided into pages, where a page consists of the maximum amount of memory that a single map register can address. Memory cannot be allocated in units smaller than a single page. VTS/OS maintains control of this memory through a table called the PHYSPAGETABLE, which contains a single entry for each page of memory. This entry indicates the status of each page, as discussed below.

If a page of memory is not currently assigned to VTS/OS itself or to any program or task, VTS/OS places a zero in the entry for that page. In addition, VTS/OS maintains a count of the number of available pages of memory in an integer variable called REMAININGPAGES. Finally, since any page of memory not assigned to VTS/OS itself could conceivably be available at any given instant, VTS/OS uses a variable, referred to in the PDL in Chapter 6 as "FIRST-AVAILABLE-MEMORY-PAGE-INDEX", which contains the index of the first page of memory not assigned to VTS/OS.

If a page is assigned to VTS/OS, the entry for that page will contain a (currently undefined) value indicating this assignment.

If a page is assigned to a program, i.e., contains fixed code which may be used by several tasks, the entry for that page will contain the address of the Image Control Block for that program.

If a page is assigned to a task, i.e., contains either the variables for the task or part of a buffer which the task "owns", the entry for that page will contain the address of the Task Control Block for that task.

5.3.2 Overlay Control

5.3.2.1 LOADOVERLAY

GENERAL

This routine is used by a task to have itself mapped to a specified overlay.

USER INTERFACE STUB

The user interface stub will set up the LOADOVERLAY Event Control Block as follows:

EVENTFLAG - set to zero
STATUSPTR - address of caller's STATUS variable
PARAM1 - OVERLAYID parameter from call

SPECIFIC PROCESSING

LOADOVERLAY first finds the Image Control Block associated with the task's Task Control Block. It then traverses the chain of Overlay Control Blocks connected to the ICB, comparing PARAM1 from the LOADOVERLAY ECB with the OVERLAYID subfield of each OCB. If no match is found, it sets the STATUS in the LOADOVERLAY ECB to ERR-OLAYID and LOADOVERLAY exits to the main Exec loop. If a match is found, the map registers are copied from the OCB into the TCB, which contains the active task map. It then sets the STATUS to NO-ERROR and exits to the main Exec loop.

5.3.2.2 RELEASEOVERLAY

GENERAL

The RELEASEOVERLAY structure chart is shown in Figure 5-11. This routine allows a task to release an overlay (i.e., to free a portion of its logical address space) without mapping to another overlay.

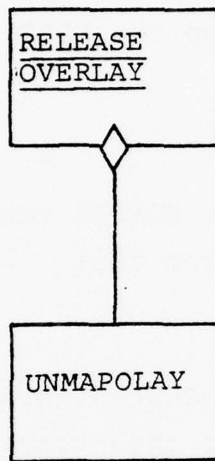


Figure 5-11. RELEASEOVERLAY

USER INTERFACE STUB

The user interface stub will set up the RELEASEOVERLAY Event Control Block as follows:

EVENTFLAG - set to zero
STATUSPTR - address of caller's STATUS variable
PARAM1 - OVERLAYID parameter from the call

SPECIFIC PROCESSING

RELEASEOVERLAY first finds the Image Control Block associated with the task's Task Control Block. It then traverses the chain of Overlay Control Blocks connected to the ICB, comparing PARAM1 from the RELEASEOVERLAY ECB with the OVERLAYID subfield of each OCB. If no match is found, it sets the STATUS in the RELEASE-OVERLAY ECB to ERR-OLAYID and exits to the main Exec loop. If a match is found, UNMAPOLAY is called to perform the actual unmapping.

UNMAPOLAY compares the map registers in the TCB with the map registers in the OCB. If they are not the same, a STATUS of ERR-MAP is returned, indicating that the task was not mapped to the specified overlay. If they are the same, an invalid value is set in the appropriate map registers in the TCB, and a STATUS of NO-ERROR is returned.

5.3.3 Buffer Control

5.3.3.1 GETBUFFER

GENERAL

The GETBUFFER structure chart is shown in Figure 5-12. This call allows a task to request a data buffer. The buffer may be made sharable (with descendants) or private. In either case, the requesting task will always be the "owner" of the buffer.

USER INTERFACE STUB

The user interface stub will set up the GETBUFFER Event Control Block as follows:

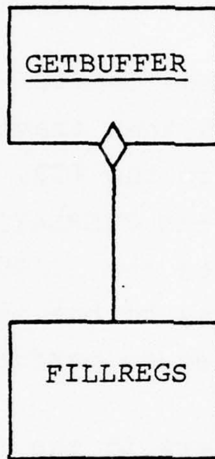


Figure 5-12. GETBUFFER

EVENTFLAG - set to zero

STATUSPTR - address of caller's STATUS variable

PARAM1 - LOGADDR parameter from call

PARAM2 - NWORDS parameter from call

PARAM3 - SHAREMODE parameter from call

PARAM4 - address of caller's BUFFERID parameter

SPECIFIC PROCESSING

GETBUFFER's first error check is on the value in PARAM2, the amount of memory requested. If this is negative, or if this amount will require more map registers than this program is allowed for data buffers (as specified by the MAPSFORBUFF subfield in the ICB), GETBUFFER sets the STATUS to ERR-NWORDS and exits to the main Exec loop. If the amount is legal, GETBUFFER checks to see whether or not the amount of memory requested is available. If not, it sets the STATUS to ERR-NMR and exits to the main Exec loop. If memory is available, GETBUFFER checks the logical address in the caller's address space at which the buffer is to start (PARAM1). If this address is not in the program's buffer area, or if it is not aligned on a map register boundary, GETBUFFER sets the STATUS to ERR-ADDR and exits to the main Exec loop. The final check is to see whether or not a free Buffer Control Block is available. If not, GETBUFFER sets the STATUS to ERR-NMR and exits to the main Exec loop. If available, GETBUFFER "captures" the BCB, links it to the TCB, and fills in the various subfields of the BCB. This includes assigning a unique ID to this buffer, which is also returned to the calling task through PARAM4 of the ECB.

GETBUFFER uses one lower-level procedure, FILLREGS, which actually goes through the list of real memory pages (the PHYSPAGETABLE), allocating pages and setting up the actual map register values in the BCB.

5.3.3.2 EQUATEBUFFER

GENERAL

This call is used by a task to gain access to a buffer originally allocated to one of its ancestors. The task may then map itself into the buffer and use the buffer freely. Coordinating several descendant tasks, all of which are using the same buffer, is the responsibility of the tasks themselves, as it is not possible for VTS/OS to protect the tasks from each other.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for the EQUATEBUFFER call as follows:

EVENTFLAG - set to zero
STATUSPTR - address of caller's STATUS variable
PARAM1 - PARENTBUFFERID parameter from the call
PARAM2 - PARENTTASKID parameter from the call
PARAM3 - address of caller's MYBUFFERID variable

SPECIFIC PROCESSING

EQUATEBUFFER first attempts to find an ancestor of the calling task whose TASKID is the same as the value in PARAM2. It does so by traversing upwards through the family tree, using the PARENT subfield of each succeeding Task Control Block. If no such ancestor can be found, EQUATEBUFFER sets the STATUS in the ECB to ERR-PARENT and exits to the main Exec loop. If the ancestor is found, EQUATEBUFFER then traverses the chain of Buffer Control Blocks attached to the ancestor's TCB, seeking one with a BUFFERID equal to that given in PARAM1. If no such ECB can be found, EQUATEBUFFER sets the STATUS in the ECB to ERR-BUFFID and exits to the main Exec loop. If the BCB is found,

EQUATEBUFFER checks to ensure that the buffer is sharable and that there is not a pending RELEASEBUFFER call for that buffer. If either condition is not met, EQUATEBUFFER sets the STATUS in the ECB to ERR-PERMIT and exits to the main Exec loop. If both conditions are met, EQUATEBUFFER attempts to acquire a blank BCB for the calling task. If none is available, EQUATEBUFFER sets the STATUS to ERR-NMR and exits to the main Exec loop. If a BCB is available, EQUATEBUFFER allocates it to the calling task, fills in all the subfields of the calling task's BCB, updates the USECOUNT subfield of the owning task's BCB, and links the new BCB to the calling task's queue of BCBs. Using PARAM3, it also returns to the calling task the new buffer ID by which it must refer to this buffer. It then exits to the main Exec loop.

5.3.3.3 MAPTOBUFFER

GENERAL

This routine is used by the calling task to map itself into a buffer which has been allocated to it by either a GETBUFFER call or an EQUATEBUFFER call. The latter calls create the mechanisms whereby a task can actually reference entries in a buffer, but do not modify the active map registers to permit addressing the buffer.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for the MAPTOBUFFER call as follows:

EVENTFLAG - set to zero
STATUSPTR - address of caller's STATUS variable
PARAM1 - BUFFERID parameter from call

SPECIFIC PROCESSING

MAPTOBUFFER traverses the queue of Buffer Control Blocks attached to the calling task's Task Control Block, attempting to find a

BCB whose BUFFERID subfield matches the value in PARAM1. If none is found, it sets STATUS to ERR-BUFFID and exits to the main Exec loop. If the BCB is found, the RELEASEECBPTR subfield is checked to determine whether or not a RELEASE is in progress for this buffer. If so, it sets STATUS to ERR-PERMIT and exits to the main Exec loop. If the buffer is a shared buffer, this check "locks" the owner out of the buffer once he has issued a RELEASEBUFFER call (in that he is not able to map back in once he has mapped himself out), but does not affect descendant tasks who have alreay obtained access to the buffer through an EQUATEBUFFER call. Each such descendant is able to map into the buffer until that task issues a RELEASEBUFFER call.

If all of the above error checks have been passed, MAPTOBUFFER will transfer the map register values from the BCB into the TCB, which contains the "active" task map. It will then set the STATUS to NO-ERROR and exit to the main Exec loop.

5.3.3.4 RELEASEBUFFER

GENERAL

This call allows a task to release a buffer when it is through using that buffer. While the actual mechanics of the release depend upon whether the buffer is private or sharable, whether or not the calling task is the owner, and whether or not more than one task has access to the buffer at the time of the call, the net effect of the call is the same. That is, if successful, the calling task is denied all further access to the buffer.

USER INTERFACE STUB

The user interface stub will set up the Event Control Block for the RELEASEBUFFER call as follows:

EVENTFLAG - EVENTFLAG parameter from the call
STATUSPTR - address of caller's STATUS variable
PARAM1 - BUFFERID parameter from the call

SPECIFIED PROCESSING

RELEASEBUFFER begins processing by attempting to locate the calling task's Buffer Control Block whose BUFFERID subfield matches the value in PARAM1. If no such BCB is found, RELEASE-BUFFER sets the STATUS to ERR-BUFFID and exits to the main Exec loop. If the BCB is found, RELEASEBUFFER checks to determine if the calling task is the owner.

If the calling task is the owner of the buffer, RELEASEBUFFER decrements the USECOUNT subfield of the BCB. If the USECOUNT is still not zero, descendant tasks must also have access to this buffer. In this case, RELEASEBUFFER sets the RELEASEECBPTR subfield of the BCB to the address of the RELEASEBUFFER ECB (which indicates that a release is in progress), unmaps the calling task from the buffer, sets the STATUS to REQUEST-IN-PROGRESS, and exits to the main Exec loop. If the decremented USECOUNT is now zero, RELEASEBUFFER checks to determine if any I/O is outstanding for this buffer.

If any I/O is outstanding, RELEASEBUFFER sets the STATUS to ERR-IOIP, increments the USECOUNT subfield (which brings it to one), and exits to the main Exec loop. This is done only when the calling task owns the buffer, no other task has access to the buffer, and there is outstanding I/O, which implies that the calling task itself has initiated the I/O.

If no I/O is outstanding, RELEASEBUFFER can completely release the buffer, since no other tasks have access to it (i.e., USECOUNT is zero). In this case, it unmaps the calling task from the buffer, returns the memory and the BCB to the system free-lists, sets the STATUS to NO-ERROR, and exits to the main Exec loop. Thus, all the possible alternatives if the calling task is the owner of the buffer have been addressed.

If the calling task is not the owner of the buffer, RELEASEBUFFER first checks to determine if the calling task has any I/O outstanding in the buffer. If so, RELEASEBUFFER sets the STATUS to ERR-IOIP and exits to the main Exec loop. If the calling task does not have any I/O outstanding in the buffer, RELEASEBUFFER is free to release the buffer from the calling task. It then unmaps the calling task from the buffer and returns the BCB to the system free-list. As a result, the buffer owner's BCB is checked to determine whether this action allows a pending RELEASE to complete. The check is performed by first decrementing the USECOUNT subfield of the owner's BCB. If the decremented USECOUNT is not zero, no I/O is in progress in the buffer and a RELEASE is pending, RELEASEBUFFER performs a complete release of the buffer. That is, it returns the memory and the owner's BCB to the system free-list, sets the STATUS in the owner's RELEASEBUFFER ECB to NO-ERROR, and calls EVENTCOMPLETE to notify the owner that his RELEASEBUFFER call has finally completed. It then exits to the main Exec loop.

RELEASEBUFFER uses two lower-level routines, UNMAPBUFFER and RETURNREALMEMORY, which are shown in Figure 5-13. UNMAPBUFFER is used to map a task out of a buffer by placing an invalid map register value in the task's active task map for every map register which is currently mapping the task into the buffer. RETURNREALMEMORY is a routine which is used simply to return pages of system memory to the system free-lists.

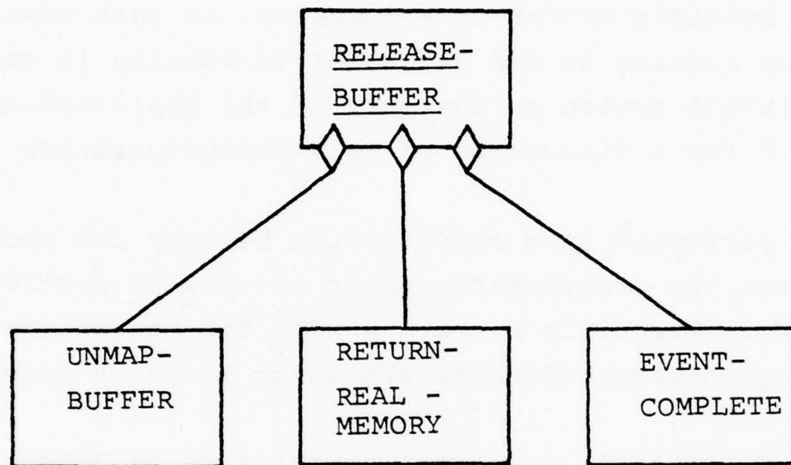


Figure 5-13. RELEASEBUFFER

5.4 INTER-TASK COMMUNICATION

The Inter-Task Communication (ITC) facility of VTS/OS is provided to allow communication among applications processes, and among operating system tasks, without requiring that the location of the parties to the conversation be known. This location transparency is necessary to allow dynamic reconfiguration in the multiple processor VTS system, so that tasks may be switched from running in one processor to running in another processor without action on the part of the applications programs. See Section 7 for a discussion of this reconfiguration procedure.

The actions performed in a conversation between two processes are described from the programmer's point of view in Section 4.4. This section is primarily concerned with the mechanisms for accomplishing conversations and providing location transparency.

Figure 5-14 depicts the most complex instance of Inter-Task Communication, where Task A and Task B are in difference processors and both the message and the associated answer are too long to be sent in instructions. In response to Task A's SENDMESSAGE call, an instruction is sent to the processor containing Task B at transmission 1. Task B is waiting for an incoming message, and is awakened by the arrival of the instruction. Task B gets access to the MESSAGEID of the message through its GETSTATUS call, and issues a TRANSFERMESSAGE request.

Transmission 2 is the buffer code assignment to allow A's message to be sent across the bus at transmission 3 and placed directly in the message buffer for B by the bus interface. B is notified that the message has been transferred, and may now process the message and formulate its answer. When Task B issues the SENDANSWER request, transmission 4 is sent, and consists of an instruction notifying A's processor that the answer is ready for transmission but requires more room than is available in the instruction.

```
TASK A                                        |   TASK B
                                              |
SENDMESSAGE (EFLAG1, ....                     |   REQUESTMESSAGE (EFLAG4, ....
    Router                                    |   WAITxxx (EFLAG4, ....
        Dispatcher                            |
            Bus ----------------------------> Bus
                           1                     Dispatcher
                                                     Router   /*awaken B*/
                                                  GETSTATUS (EFLAG4, ....
                                                  TRANSFERMESSAGE (EFLAG5, ....
                                                     Router
                                                 Dispatcher
            Bus <---------------------------- Bus
        Dispatcher         2                  |
            Bus ----------------------------> Bus   WAITxxx (EFLAG5, ....
                           3                     Dispatcher
                                                     Router   /*awaken B*/
                                                  /*B processes message*/
                                                  GETSTATUS (EFLAG5, ....

WAITxxx (EFLAG1, ....
                                                  SENDANSWER (EFLAG6, ....
                                                     Router
                                                 Dispatcher
            Bus <---------------------------- Bus
        Dispatcher         4                  |
            Bus ----------------------------> Bus
                           5                     Dispatcher
            Bus <---------------------------- Bus
        Dispatcher         6                     Dispatcher     /*awaken B*/
    Router   /*awaken A*/                         /*completion of conversation*/
/*A processes answer*/                            GETSTATUS (EFLAG6, ....
GETSTATUS (EFLAG1, ....
```

Notes - Event flags are shown for clarity only. Any event flag value other than INDEPENDENT may be substituted above.

- Transitions from Bus to Dispatcher to Router to applications process may bypass the actual router routines.

(wavy line) indicates the continued processing or waiting of Tasks A and B.

Figure 5-14. Inter-Task Communications

A's dispatcher readies the bus interface to receive B's answer and transmits the buffer code assigned at transmission 5. Transmission 6 is B's answer, and terminates the conversation between Tasks A and B.

Two tables are used extensively by ITC routines. The GLOBALTABLE array indicates processor assignments of tasks under the current system configuration. Associated with this table is MESSAGETABLE, containing the message queue heads for all tasks which are now active (or may become active after reconfiguration) in this processor. The PASCAL data definitions for these tables are:

```
GLOBALTABLE:  array [1..MAXCONFIGURATION, 1..MAXPROCESSNAME]
              of PROCESSORNUMBER;

MESSAGETABLE: array [1..MAXPROCESSNAME] of PROCESSORNUMBER;
```

5.4.1 ROUTER

As described in the Phase II Report, the Router consists of the SENDMESSAGE, REQUESTMESSAGE, TRANSFERMESSAGE, and SENDANSWER. The individual functions of these routines have been slightly altered to provide a more consistent and simple user interface, and to provide the capability of context-sensitive error detection.

The role of the Router has been limited somewhat from that formerly envisioned, in that the Router exists between the user task and the Dispatcher only for functions being originated by the user task. For instructions, buffer codes or messages received from another task causing actions affecting the user task, the Event Control Block basis of VTS/OS not only allows the Dispatcher to directly notify the user task, but indeed makes this case easier to supervise than that in which the Dispatcher would request the Router to perform the notification. This simplifies the user task notification process and makes it more efficient.

As is discussed in Section 5.4.3, a key underlying assumption in the ITC descriptions here is the correspondence between instructions and ECB's. The Router and Dispatcher use instruction ECB's to control the conversation, and in the case of messages transmitted across the bus, both Routers and Dispatchers have access to a copy of the original instruction in the ECB they use to control the conversation.

5.4.1.1 SENDMESSAGE

GENERAL

The SENDMESSAGE routine allows user tasks to communicate with other tasks regardless of their location. The event flag associated with the SENDMESSAGE call will be set when the answer from that other task has arrived, or when an error

has been detected by the ITC routines. This includes expiration of the user-defined TIMEOUT parameter, which allows the user to set the maximum time allowed for the conversation based on the context of the request or function being performed.

USER INTERFACE STUB

The user interface stub places the following information in the request ECB:

EFLAG - Event Flag
STATUSPTR - Pointer to user's STATUS parameter
PROCESSNAME - Name of the receiving task in the Global Table
MBUFFSPEC - Message buffer specification
ABUFFSPEC - Answer buffer specification
TIMEOUT - Maximum time for conversation

DETAILED DESCRIPTION

SENDMESSAGE receives the request ECB from routine ROUTER. A message number is assigned to this conversation, and the PROCESSORID field of MESSAGEID is filled in. The GLOBALTABLE is checked for the current configuration to determine the location of the receiving task, and if the task is in this processor, MESSAGERECEIVED is called to notify the receiving task that a message has arrived. If the receiving task is not local, the Dispatcher must supervise transmission across the bus, and therefore the ECB is placed on the MESSAGEOUTQ of the Dispatcher. The SENDMESSAGE function is complete, as the Dispatcher will conduct the remainder of the conversation.

5.4.1.2 REQUESTMESSAGE

GENERAL

REQUESTMESSAGE associates an event flag with the message input queue of the task issuing the call. If any messages are already on the queue, the event flag is set during the REQUESTMESSAGE call.

USER INTERFACE STUB

The following parameters are placed in the request ECB:

EFLAG - Event Flag
STATUSPTR - Pointer to user's STATUS parameter
MESSAGEIDPTR - Pointer to user's MESSAGEID parameter
MSLENGTHPTR - Pointer to user's MSLENGTH parameter
ANSLENGTHPTR - Pointer to user's ANSLENGTH parameter

DETAILED DESCRIPTION

REQUESTMESSAGE first verifies that the calling task has a corresponding entry in the GLOBALTABLE. If the task ICB field PROCESSNAME is NIL, the task is not identified to the operating system as being able to receive messages, and ERNOIPC is returned to the user task. Otherwise, the MESSAGETABLE entry indicated by ICBPTR↑.PROCESSNAME is used to set up the Queue Control Block for that message queue so that incoming messages will cause the specified event flag to be set. The actual linkage to the TCB and setting of event flag information in the TCB are performed by CALLCOMPLETE after REQUESTMESSAGE has changed ACTIVEECB to point to the QCB for this task. Pointers to the MESSAGEID, MSLENGTH and ANSLENGTH parameters in the user's call are placed in the QCB for use by the Dispatcher.

5.4.1.3 TRANSFERMESSAGE

GENERAL

Routine TRANSFERMESSAGE moves the text of the message into the buffer area specified by the calling task, regardless of the length of the message or the location of the sending task.

USER INTERFACE STUB

The user interface stub places the following parameters in the request ECB:

EFLAG - Event Flag

STATUSPTR - Pointer to user's STATUS parameter

MESSAGEID - Message identification

MBUFFSPEC - Specification of the receiving task's message buffer area

DETAILED DESCRIPTION

Upon being called by routine ROUTER, TRANSFERMESSAGE examines the MESSAGEQ for the calling task to find the ECB containing the specified MESSAGEID. If no match for MESSAGEID is found, ERMESSAGEID is returned to the user task and the function is terminated. The processor number in MESSAGEID is then checked and, if the sending task is local, the message is moved to the receiving task's message buffer and NOERROR is returned synchronously to the calling task. If the sending task is in a different processor, and if a short message was included in the instruction, this message is moved to the receiving task's message buffer and NOERROR is returned. Otherwise, TRANSFERMESSAGE places its request ECB on the TRANSFERMESSAGEQ of the Dispatcher and has completed its synchronous processing for this request.

5.4.1.4 SENDANSWER

GENERAL

SENDANSWER completes a conversation by transferring the receiving task's answer to the sending task, regardless of location. Return from the SENDANSWER call indicates that transmission of the answer was performed without error, and does not indicate confirmation on the part of the sending task.

USER INTERFACE STUB

The following information is placed in the request ECB:

EFLAG - Event Flag

STATUSPTR - Pointer to user's STATUS parameter

MESSAGEID - Message identification for this conversation

ABUFFSPEC - Answer buffer specification

DETAILED DESCRIPTION

SENDANSWER examines the MESSAGEQ for the calling task to find the ECB containing the specified MESSAGEID. If no such ECB is found, ERMESSAGEID is returned to the user task and the function is terminated. The processor number in MESSAGEID is then checked and, if the sending task is local, the answer is copied from the receiving task's answer buffer to the sending task's answer buffer, and the function terminates with NOERROR returned to the calling task. This termination includes removal of the instruction ECB from the MESSAGEQ and signalling of EVENTCOMPLETE to the sending task, awakening him with his answer present. If the sending task is not local, the request ECB is placed on the SENDANSWERQ of the Dispatcher, and SENDANSWER is done with its synchronous processing.

5.4.2 DISPATCHER

GENERAL

The DISPATCHER performs all asynchronous operations associated with ITC. All functions which require initiation of actions on the part of the bus interface driver software are in the DISPATCHER; details on the expected operation of the bus interface driver are presented in Section 5.4.3. Interface between the router routines and the DISPATCHER is accomplished through the use of several queues, allowing the DISPATCHER to process different conversations as if each were the only conversation in progress. The collocation of the various transmission and reception stages also allows the DISPATCHER to take advantage of the similarity which exists among some of these stages.

The transmissions on the bus which are received or originated by the DISPATCHER are shown in Figures 5-15 through 5-17. For all instructions and buffer codes assignments, the control byte determines on which queue the transmission is placed for the DISPATCHER.

DETAILED DESCRIPTION

The DISPATCHER sets up its timeout list and input queues before entering the main Exec loop for cyclic processing. The timeout list for messages in process must be maintained internally, due to the limitation which would be placed on the number of possible conversations by the use of a separate event flag for each conversation timeout.

DISPATCHER then issues a WAITANY call to suspend itself until an input ECB arrives on one of its queues. The first queue processed is the MESSAGEINSTRUCTIONQ. The ECB from this queue is placed on the MESSAGEQ of the destination task by routine MESSAGERECEIVED, which also notifies that task that a message has arrived.

```
INSTRUCTION

============
! SYNC      !
!-----------!
! SYNC      !
!-----------!
! R ADDR    !
!-----------!
! CONTROL   !
!-----------!
! T ADDR    !
!-----------!
! COUNT     !
!-----------!
! MSG ID    !
!-         -!
! MSG ID    !
!-----------!
! MBUFSIZE  !
!-         -!
! MBUFSIZE  !
!-----------!
! ABUFSIZE  !
!-         -!
! ABUFSIZE  !
!-----------!\
      .       !
      .        > MESSAGES OF 8 BYTES OR LESS
      .       !     MAY BE SENT HERE - IF NO
!-----------!/      DATA IS PRESENT, CKSUM
! CKSUM     !       FOLLOWS ABUFSIZE.
!-         -!
! CKSUM     !
============
```

Figure 5-15. Transmission on Bus Received by the Dispatcher

MESSAGE OR ANSWER BUFFER CODE ASSIGNMENT

```
============
! SYNC     !
!----------!
! SYNC     !
!----------!
! R ADDR   !
!----------!
! CONTROL  !
!----------!
! T ADDR   !
!----------!
! COUNT    !
!----------!
! MSG ID   !
!-        -!
! MSG ID   !
!----------!
! BUF CODE !
!----------!
! CKSUM    !
!-        -!
! CKSUM    !
============
```

Figure 5-16. Transmission on Bus Orignated by Dispatcher

MESSAGE OR ANSWER

- 0 IF LAST OR ONLY PACKET OF MESSAGE

> MESSAGES OF 502 BYTES OR LESS
MAY BE SENT HERE, OR PACKETS OF
OF LONGER MESSAGES (ALL BUT THE
LAST PACKET WILL BE 502 BYTES LONG).

Figure 5-17. Transmission on Bus
Originated by Dispatcher

Requests arriving on the MESSAGEOUTQ and SENDANSWERQ are examined next, as their processing is similar. For ECB's from the MESSAGEOUTQ, the timeout list must be updated to keep track of this new conversation. The timeout is sorted into the timeout list by its expiration time, and, if necessary, the current timeout is cancelled and a new timeout is requested. The contents of the ECB are the nucleus for the instruction to be sent; if the message or answer is short enough to be transmitted in the instruction, the data is moved to the ECB, the bus interface driver is notified that an instruction is to be sent, and the ECB returns on the SENTMESSAGEQ in the MESSAGEOUTQ case, or the COMPLETIONQ in the SENDANSWERQ case. If the message or answer is long, the bus interface driver is notified to send the instruction, and the ECB returns on the SENTINSTRUCTIONQ.

Next the TRANSFERMESSAGEQ and ANSINSTRUCTIONQ ECB's are processed. If the input is from the ANSINTRUCTIONQ, the ECB with a matching MESSAGEID is located on the SENTMESSAGEQ to supply the buffer address for the sending task's answer buffer. For both cases, a buffer code is assigned and the ECB is passed to the bus interface driver, which will send the buffer code assignment and set up one of the special-purpose DMA registers to receive the message or answer data when it is sent by the other processor. When the corresponding message or answer has arrived, the bus interface driver returns the ECB on the COMPLETIONQ.

Next the MSBUFFCODEQ and ANSBUFFCODEQ ECB's are processed. The matching ECB is located on the SENTINSTRUCTIONQ, and the bus interface driver is requested to transmit the message or answer with the appropriate buffer code. The ECB is returned on the SENTMESSAGEQ for the MSBUFFCODEQ case, and on the COMPLETIONQ for the ANSBUFFCODEQ case.

The last queue processed is the COMPLETIONQ. ECB's arrive on this queue when a message has been received, when an answer has been sent, or when an answer has been received by the sending task. In the

first two cases, the ECB arriving on the queue from the bus interface driver is used to signal completion of the receiver's TRANSFERMESSAGE call. In the second case, however, the conversation has been terminated on the receiving side, and so the corresponding entry from the MESSAGEQ for that task must be removed and released to the free ECB queue. In the case of a received answer, the ECB from the SENTMESSAGEQ must be used to signal event completion to the sending task. The timeout entered when the conversation began is removed from the timeout list, and the ECB arriving on the COMPLETIONQ is released to the ECB list.

The timeout list has an event flag and is processed in the same CASE statement with the ECB queues. However, if a timeout occurs, the ECB for that timeout is removed from its current queue and used to send an ERTIMEOUT return to the sending task. Any figure reference to that conversation's MESSAGEID will be ignored by the sending processor.

5.4.3 Bus Interface Driver Operation

The bus interface driver for the VTS Operating System is structured somewhat differently from the device drivers. The bus interface performs more complex functions than most devices, and therefore requires more intelligence in the driver. The driver has multiple output queues, which are selected for passing incoming instructions, buffer code assignments, and data arrival notices to the Dispatcher. The control byte of the transmission (which is not a part of the transmission moved into memory by DMA, but which is available in the bus interface status registers) is used to determine on which of the output queues the ECB is placed.

The bus interface itself is assumed to be responsible for the separation of long messages into packets, and for the supervision of packet sequencing and reception. If this is not the case for the bus interface selected, this function will have to be performed in the driver, and will necessarily be less efficient. As detailed in Section 7 of the Phase II Report, the bus interface is expected to contain intelligence sufficient to perform several types of error detection and to provide message-specific reception and DMA transfer of data into reserved memory areas; these functions also will have to be implemented in the bus interface driver software if they are in fact not performed by the bus interface.

One major assumption present in the ITC routines presented in this report is that the physical storage format for data used by PASCAL can be sufficiently controlled and that the relative positions and sizes of fields within a data structure may be predicted. This is necessary in order that the Event Control Block may be used as the storage unit to which and from which instructions are transferred by the bus interface. If this is not the case, the memory allocation and event control functions within the ITC routines become more complex and less efficient.

5.5 FILE MANAGEMENT AND INPUT/OUTPUT (I/O) OPERATIONS

This section describes the detailed design and operation of the file management and I/O sections of the VTS Operating System. The file structures supported by VTS/OS and the conventions for device and file I/O are presented first, followed by the detailed descriptions of the associated operating system functions and tasks.

Where potential differences in I/O structures and organizations exist between processors or devices selected by the Coast Guard which will affect the operating system functions, sections of the Program Design Language description are presented as comments detailing the necessary operation, rather than as PASCAL code. The file management and I/O functions will be somewhat more sensitive to the equipment selected than other functions.

5.5.1 File Management

Within VTS/OS, all input and output data transfers are performed on the basis of logical I/O channels. A channel may be assigned either to a non-file-structured device or to a file within a file-structured device. This allows a common set of channel-oriented I/O operations to be used for data transfer, regardless of the physical organization and physical characteristics of the device involved. Since the differences among different device types cannot be completely ignored, VTS/OS will not perform some functions for some device types, such as random I/O functions on non-directory devices.

The File Management routines provide single-user access to a non-directory device or file within a directory device. Sharing of a device or file among tasks within a processor is possible, since one user task may inform another user task of the channel number

assigned when the device or file was opened; however, VTS/OS provides no synchronization or coordination to support this capability, and so the user tasks must coordinate themselves if shared access is desired at this level. Within the VTS application, there will be one user task supervising all transfers to and from a device or file, and so no inter-task coordination will be required.

Where file names are required in system calls, there are several ways of specifying the file or device desired. Devices which are not file-structured are referenced by a specification of the form "DVn:", where DV is the mnemonic assigned to that device class, and the number "n" specifies the unit number within that device class. Files within a file-structured device are referenced by names which consist of a root segment of from one to ten alphanumeric characters, and an optional extension of one or two alphanumeric characters, separated from the root by a period. The device on which a named file resides may be explicitly specified; if no device is specified, the system supplies the specification for the master system disc in that system call. Thus, "DK1:VESSELSORT.TI" specifies a named file residing on disc unit one, with a root of "VESSELSORT" and an extension of "TI". Magnetic tapes are considered to be file-structured but non-directory devices, and as such, will recognize only numeric file names. The first file on a magnetic tape would be MTn:0, the second MTn:1, and so on.

Certain extensions will be reserved for system use, particularly in the area of task image names. The reserved extensions are not specified here, since it may be desirable to have them conform to the conventions of the development operating system.

Two types of disc file structures are supported by VTS/OS. Segmented files are provided primarily due to their extendability; the size of a segmented file is not known when the file is created, and the file is extended each time a sequential write operation is performed. This capability is not currently of importance to

the VTS application; however, its availability allows later system modifications to make use of extendable disc file structures. Contiguous files are of fixed length, and that length must be specified when the file is created. However, access to contiguous files is faster than access to segmented files, since pointer blocks need not be read from disc to indicate the location of the data area to be read or written.

There are certain items of file information which must be present in some form in order to support the file structures necessary for VTS/OS. These items are described here, as they will be assumed in the descriptions below. The overall directory structure of disc devices is assumed to consist of a system directory area on the disc containing the data shown in the FDB description for each file on the disc, and a map area consisting of one bit per sector on the disc indicating whether that sector is allocated to a file or is free for allocation to future files. The PASCAL data structures presented in this report must therefore correspond to the physical representation of these items of information on the disc.

The structure assumed for system directory blocks is:

```
DIRBLOCK:  ARRAY [1..ENTRIESINDIRBLOCK] of;
                     DISCDIRECTORYENTRY;
```

The map block format is:

```
MAPBLOCK:  ARRAY [1..SECTORSINMAPBLOCK] of FREE,USED;
```

where FREE := TRUE and USED := FALSE. It is expected that FREE and USED will be implemented as bit flags to prevent excessive size of the map area.

Segmented File Structures

Figure 5-18 shows the pointer-based structure of segmented files. The disc directory entry contains STARTADDR and ENDADDR, which are the disc addresses of the first and last Segment Pointer Blocks (SPB's) in the segmented file. Each SPB has 2 pointers to the next SPB in the SPB chain, the last SPB in the chain, and an array of Data Segment Pointers (DSP's) containing the disc addresses of the data segments which contain data written to the file. The size of an SPB has not been set, but is anticipated to be several disc blocks, to allow one SPB to contain DSP's for about 500 data segments. The PASCAL data definition for the disc directory entry is shown in Figure 5-19, and that for SPB's is shown in Figure 5-20. The end of a segmented file is indicated by a DSP or NEXTSPB pointer whose value is NIL.

All types of read and write operations are supported for segmented files. Random-access writing to segmented files is restricted to the pre-existing file for two reasons. First, an error in the FILEADDRESS field of the WRITER system call could cause inordinate amounts of disc space to be allocated for SPB's and data segments, if the erroneous FILEADDRESS is considerably beyond the current end of the file. Second, an ambiguity exists concerning any data segments allocated to fill the gap between the end of the file and the random record specified: do these data segments contain data which may be read and misinterpreted as valid information by the user task, or must data be written into these data segments before they may be read? Therefore, no extension of segmented files by WRITER is permitted, and an error code of ERFILEADDR will be returned to the user task if such is attempted.

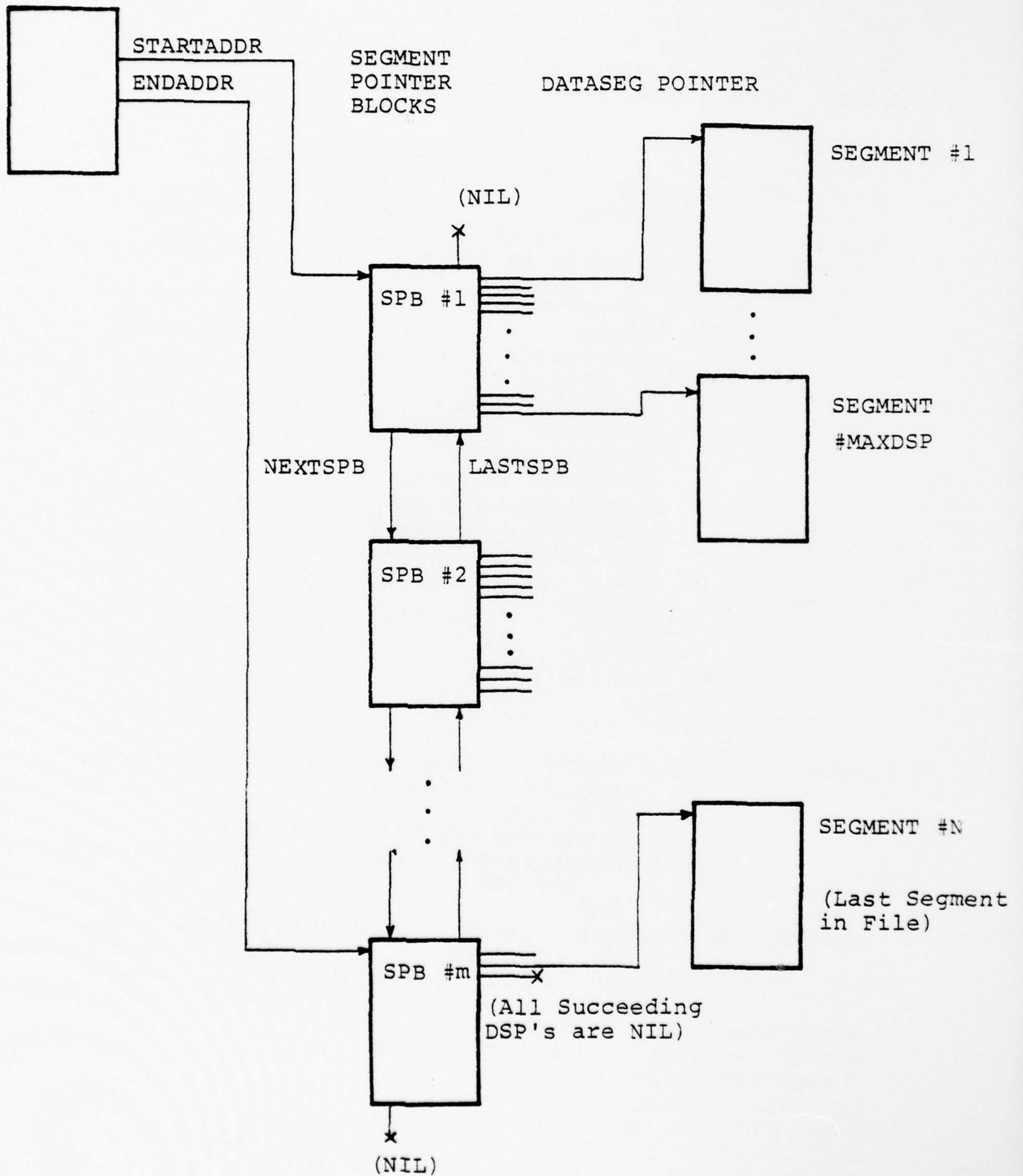


Figure 5-18 Segmented File Structure

FILE DESCRIPTOR BLOCK

```
/*THE FILE DESCRIPTOR BLOCK IS USED BY THE I/O ROUTINES
  TO CONTAIN INFORMATION ON THE PROGRESS OF I/O REQUESTS
  AND TO MAKE ACTION REQUESTS TO DEVICE DRIVERS.  FDB'S
  ARE LINKED TO THE DDB FOR THE DEVICE INVOLVED, AND THE
  FDB POINTER IS PLACED IN THE CHANNEL TABLE FOR THE
  OPENED CHANNEL INVOLVING THAT FILE OR DEVICE.*/

FDB= RECORD
    FDBQLINK: QLINK;        /*LINK FOR DDB FDB QUEUE*/
    DDBPOINTER: POINTER;       /*POINTER TO DDB*/
    CHANNEL: 1..MAXCHANNEL;      /*CHANNEL FOR THIS FDB*/
    ECBPTR: POINTER;       /*USER REQUEST ECB*/
    /*INFORMATION CONTAINED IN THE DISC DIRECTORY*/
    DISCDIRECTORYENTRY = RECORD
       FILENAME: FILENAMETYPE;
       ATTRIBUTES= RECORD
                   READPROTECT: BOOLEAN;
                   WRITEPROTECT: BOOLEAN;
                   PROTECTED: BOOLEAN;
                   DIRECTORY: BOOLEAN;
                   BLOCKED: BOOLEAN;
                   SEGMENTED: BOOLEAN;
                   CONTIGUOUS: BOOLEAN;
                   SEQUENTIAL: BOOLEAN;
                   AVAILABLE: BOOLEAN
                   END;
       STARTADDR: DISCADDR;
       FILESIZE: INTEGER;
       ENDADDR: DISCADDR;
       DATASEGMENTSIZE: INTEGER;
       CRDATE: DATE;
       CRTIME: TIME;
       END;
    USERMEMORYSPEC = RECORD     /*OBTAINED AT TIME OF CALL*/
                     BUFADDR: POINTER;
                     BUFFSIZE: INTEGER;
                     NUMBEROFMAPREGISTERS: INTEGER;
                     MAPREGISTERS: ARRAY [1..MAXMAPREGISTERS] OF
                        INTEGER;
                     END;
    SEQPOSITION = RECORD   /*POSITION FOR SEQUENTIAL I/O*/
                  BLOCK: DISCADDR;        /*CURRENT DISC BLOCK*/
                  WORD: INTEGER        /*WORD WITHIN BLOCK*/
                  END;
    WORDSTRANSFERRED: INTEGER;
    BUFFER: POINTER;  /*TO BUFFER BCB*/
    CURRENTSPBBUFFER: POINTER;
    DEVICEDRIVERREQUEST: INTEGER;
    CURRENTDISCSECTOR: DISCADDR;  /*USED BY DRIVER*/
    WORDSWRITTEN: INTEGER;         /*FOR SPLIT READ/WRITE*/
    DRIVERRETURNQCB: POINTER;      /*FOR NOTIFICATION OF DRIVER
                                     REQUEST COMPLETION*/
    END;
```

Figure 5-19 FDB Data Definition

```
SEGMENT POINTER BLOCK

/*THE SEGMENT POINTER BLOCK IS A DISC-BASED DATA
  STRUCTURE CONTAINING POINTERS TO THE DATA
  SEGMENTS OF SEGMENTED FILES. */

SPB=  RECORD
        LASTSPB: DISCADDR;
        NEXTSPB: DISCADDR;
        DATASEGPTR: ARRAY [1..MAXDSP] OF DISCADDR
      END;
```

Figure 5-20 SPB Data Definition

Contiguous File Structures

Figure 5-21 shows the structure of a contiguous file. The file is composed of adjacent sectors, or blocks, on the disc, and its size is permanently fixed at the time of its creation. "Adjacent," when used in this context, does not necessarily mean physically adjacent; rather, it indicates that the sectors follow one another numerically in their addressing designation. Depending upon the disc storage device chosen by the Coast Guard, the number of sectors per track or the number of tracks per cylinder may vary, and therefore the exact algorithm for generating the address of adjacent sectors cannot be completely described here. However, the algorithm is simple to determine given the structure of the disc drive.

In the directory entry for contiguous files, STARTADDR contains the starting sector number and ENDADDR the last sector number. Thus, the exact location on the disc may be determined for any read or write request without the necessity of reading data from the disc. This gives contiguous files a much faster access to random records than is available with segmented files.

Tape File Structures

Magnetic tapes are considered to be file-structured but non-directory devices by VTS/OS. Division into files is indicated by the presence of an end-of-file mark written on the tape; a double end-of-file indicates the end of accessible data on the tape. The first tape file on tape drive zero would be designated as "MT0:0", the second as "MT0:1", etc.

Magnetic tape access is supported only through the sequential operations of READSQ, READLN, WRITESQ and WRITELN. Random-access writing to magnetic tape is difficult and unreliable,

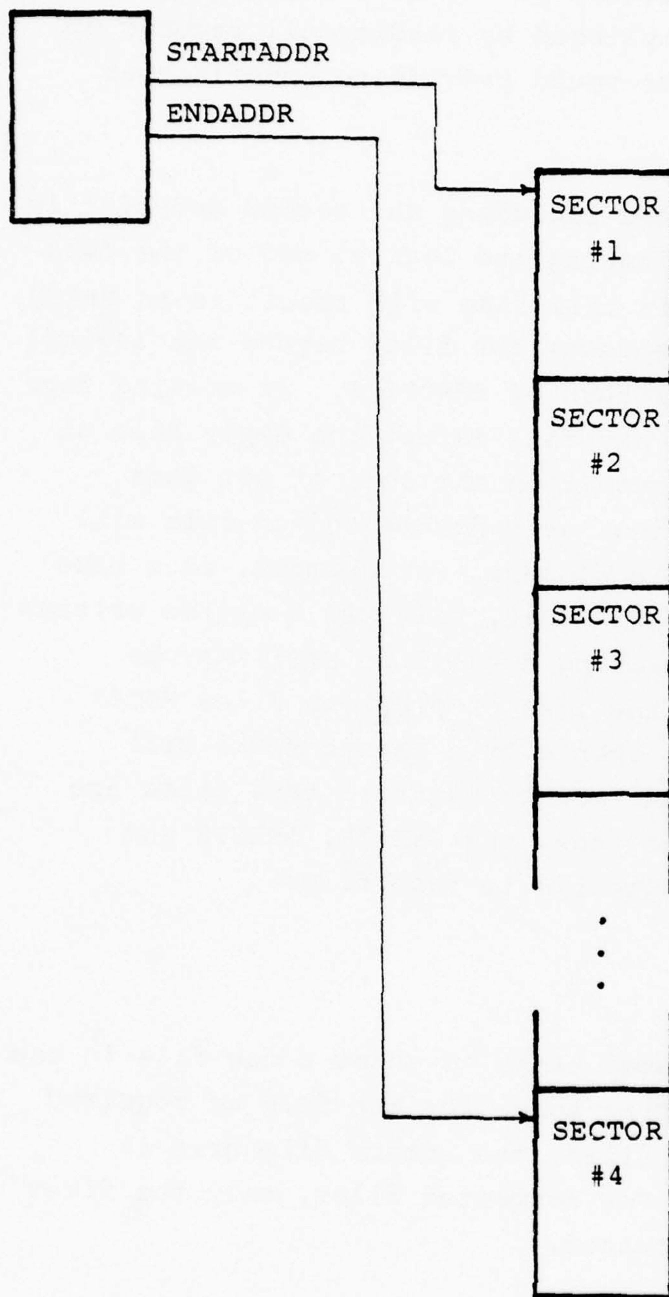


Figure 5-21 Contiguous File Structure

frequently resulting in destruction of the following data record; random access reading is accomplished by reading all records up to the desired record, and thus would provide no benefit over sequential access modes.

Tape files may be read up to and including the second end-of-file of the double end-of-file indicating the logical end of the tape (e.g., a read request issued in this case will result in an EREOF return with no data read). Requests for files beyond the logical end of tape will result in a return of ERNOFILE. In writing tape files, the writing of data to any file except the empty file at the logical end of tape will result in the loss of all data following the newly written file, as a double end-of-file will be written when the new file is closed. For example, on a tape with data files MTO:0, MTO:1 and MTO:2, file MTO:3 may be written without loss of previously existing data; file MTO:1 may be written, but would result in the loss of previous files MTO:1 and MTO:2; and any attempt to open a file beyond MTO:3 will result in a return of ERNOFILE. Since magnetic tape files are named by their position on the tape, the CREATE, DELETE and RENAME procedures are not applicable to tape files.

5.5.1.1 CREATE

GENERAL

The CREATE procedure allows user tasks to enter a new file in the disc directory and allocate disc space for the file as required by its type. For contiguous files, the entire file area is allocated at creation time. For segmented files, only the first Segment Pointer Block is allocated.

USER INTERFACE STUB

The user interface stub places the following information in the user task's ECB:

EFLAG	-	Event flag
STATUSPTR	-	Pointer to STATUS parameter of user call
FILENAME	-	ASCII specification of device, name and extension of file to be created
FILETYPE	-	SEGMENTED or CONTIGUOUS
FILESIZE	-	Size of contiguous file or of data segmrnt for segmented file

DETAILED DESCRIPTION

When a request is made for file creation, an Event Control Block is enqueued for the CREATE task by routine CREATESYNC. The ECB contains the parameters of the call, and is dequeued by CREATE when it is given control of the processor by the Executive. The buffer space required for disc directory information is allocated and mapped into the task address space.

CREATE then uses several subordinate routines which are shown in Figure 5-22 and described briefly here. First, FINDDIRECTORYENTRY is called to search the disc directory for an entry with a name matching that specified in the user's CREATE call; if a matching file is found to exist on the disc, the function is terminated and error ERFILEEXISTS is passed back to the user task. CREATE then calls routine GETCONTIGUOUSSPACE to allocate adjacent disc sectors for the entire file if the FILETYPE parameter of the user's call was set to CONTIGUOUS, or for the first Segment Pointer Block if the FILETYPE was specified as SEGMENTED. If there is not sufficient contiguous space available on the disc, the function terminates, and error ERCONTIG is returned to the user task.

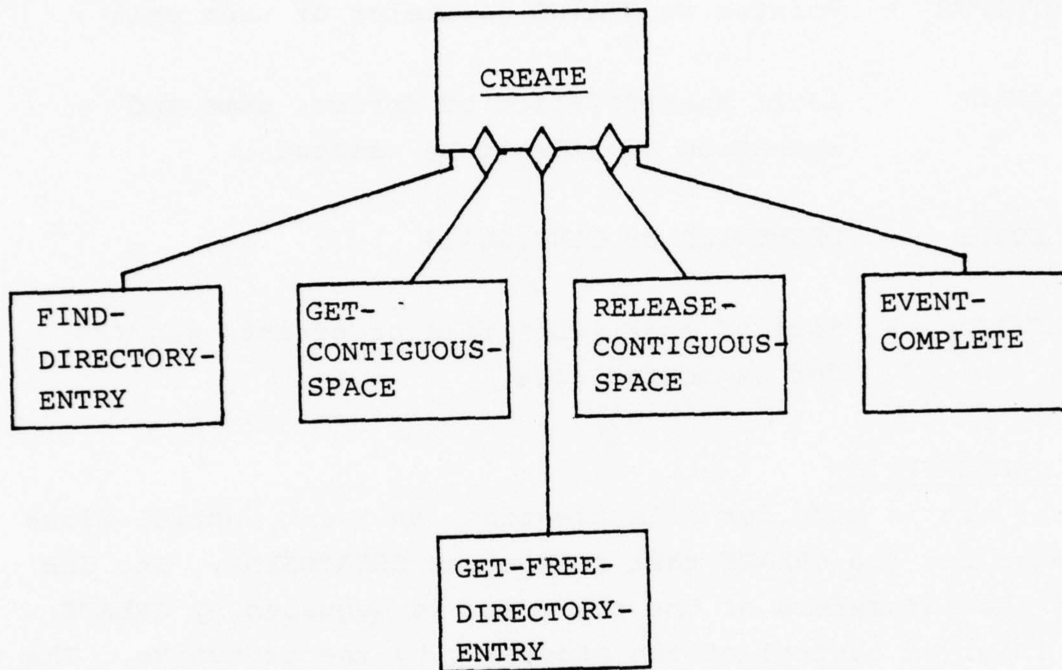


Figure 5-22 CREATE

The directory entry for the file is then created. Routine GETFREEDIRECTORYENTRY is called to locate an entry in the disc directory file which is not used. If there is not a free entry in the directory, error ERDIR is returned to the user, and all contiguous space on the disc allocated by GETCONTIGUOUSSPACE is marked FREE by RELEASECONTIGUOUSSPACE. This routine is described under DELETE, Section 5.5.1.2. Otherwise, the entry is filled in with the FILENAME from the user's call and the disc address of the first sector of the allocated file or SPB. The directory information dependent on the file type is filled in next. For contiguous files, the CONTIGUOUS attribute is set, the FILESIZE entry is set to the size specified by the user, and the ENDADDR entry is assigned the disc address of the last sector of the contiguous file. For segmented files, the SEGMENTED attribute is set, the FILESIZE is set to zero words, the DATASEGMENTSIZE is set to the size specified by the user, and the ENDADDR entry is set to the address of the allocated SPB. The creation date and time are filled in for either type of file, and the directory block containing the new entry is written back onto the disc. The CREATE function is complete at this point, and therefore CREATE signals EVENTCOMPLETE to awaken the calling task, releases its allocated buffer, and resumes waiting for an ECB on its input queue.

Routines FINDDIRECTORYENTRY and GETFREEDIRECTORYENTRY are virtually identical. Both begin reading the system directory file with FIRSTDIRBLOCK and proceed to check all entries in each block of the directory for a matching file name in the case of FINDDIRECTORY-ENTRY, or for a NIL file name in the case of GETFREEDIRECTORYENTRY. The directory block address and the entry index for the entry sought are passed back to the caller. If the entry sought is not found before the end of the directory, a value of NIL is returned in ENTRY to signal failure of the search. The directory structure assumed here is one of fixed length, for

simplicity of illustration. The actual organization of the directory file should be consistent with the development operating system of the computer selected for implementation, to make generation and maintenance of the system software convenient.

Routine GETCONTIGUOUSSPACE is passed the length of the desired file area and a buffer into which it can read the map file information describing which sectors on the disc are USED and which are FREE. Beginning with the first block of the map file, the routine checks for free sectors on the disc. STARTBLOCK is set to the disc address of the first free block found, and the routine continues checking for free sectors, reading in new blocks from the map file as necessary. If a USED sector is encountered in the map, STARTBLOCK is reset to NIL and the number of contiguous blocks found is reset to zero. If the sector being examined is shown to be free, and if STARTBLOCK has not been set to a disc address, STARTBLOCK is set to the disc address of the sector being examined. In any case, if the sector is FREE the count of free blocks found is increased by one, and the index of the current sector is incremented so that the next sector may be examined. If the number of free sectors found is equal to the number of sectors sought to satisfy the request, the search is terminated and the sectors found are marked USED in the map file. If space was found, the address of the first sector of the contiguous area is returned to the calling task. If sufficient contiguous space was not found, STARTBLOCK is set to a value of NIL to notify the calling routine that the request was not satisfied.

5.5.1.2 DELETE

GENERAL

The DELETE procedure allows the removal of a named file from disc directory. Any space allocated to that file is released when the file is deleted, and may be used immediately for creation of other files. Files which are protected or currently opened to a channel may not be deleted.

USER INTERFACE STUB

The user interface stub transfers the following information into the ECB:

EFLAG	-	event flag
STATUSPTR	-	pointer to STATUS parameter of the user call
FILENAME	-	ASCII description of device, name and extension for file to be deleted

DETAILED DESCRIPTION

When a request for file deletion is made, the ECB containing the call parameters for that request is enqueued for the DELETE task by routine DELETESYNC and DELETE removes the ECB from its input queue.

DELETE then uses several subordinate routines which are shown in Figure 5-23 and described briefly here. (Asynchronous routines GETBUFFER and MAPTOBUFFER are not shown in Figure 5-23 as subroutine calls since they are executed through the Executive.) First, routine FINDDEVICE is called to get the pointer to the disc Device Descriptor Block. The chain of File Descriptor Blocks linked to the disc DDB is then examined, to determine whether or not the file is currently open. If the file is open, the function is terminated and error ERFILEOPEN is returned to the user task. DELETE then allocates buffer space for disc directory blocks, disc map blocks, and Segment Pointer Blocks; this space is shown as three separate buffers for clarity, but could be allocated as one buffer large enough to accommodate the three requirements. Routine FINDDIRECTORYENTRY is then called to search the disc directory file for the specified file. If the file is not found, the function is terminated and error ERNOFILE is returned to the user. The attributes of the file are examined to determine whether the file is protected or unprotected; if PROTECTED, the file may not be deleted, and the function terminates with error ERPROTECTED returned to the user task.

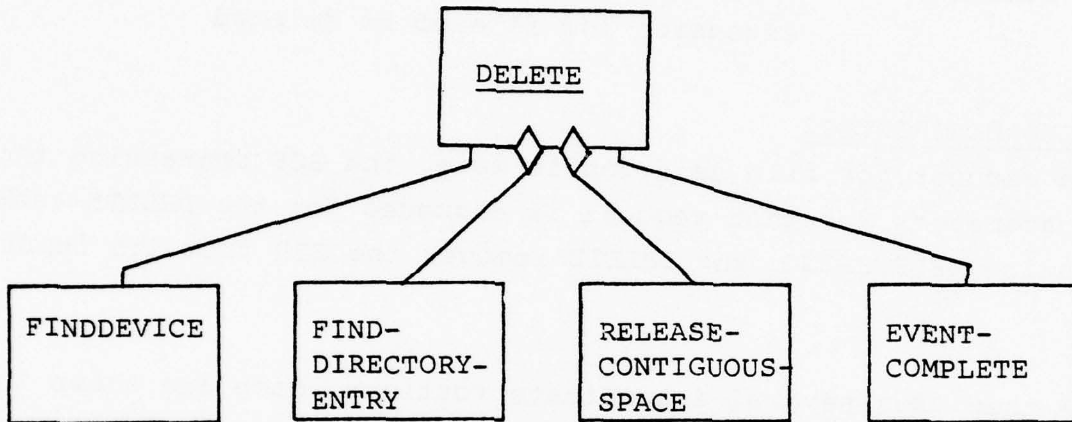


Figure 5-23. DELETE

The file is removed from the directory by setting its FILENAME to NIL and rewriting the directory entry to the disc. The file space allocated to the file must now be marked FREE in the map file. For contiguous files, this may be accomplished by one call to routine RELEASECONTIGUOUSSPACE with the FILESIZE given in the disc directory entry, beginning with the disc sector indicated by STARTADDR. For segmented files, DELETE must pass through all SPB's, releasing each data segment and then releasing the SPB itself. This process is terminated by a Data Segment Pointer with a value of NIL, or in the event that the last SPB in the file has all of its DSP's used, by a NEXTSPB pointer with a value of NIL.

The DELETE function is complete at this point. DELETE signals event completion to awaken the user task, and releases all buffer space allocated for the DELETE task. The task then waits for another delete request to enter its input queue.

Routine RELEASECONTIGUOUSSPACE receives the starting sector address and the size in words of the contiguous area to be released, as well as the buffer in which the disc map blocks may be read. MAPBLOCKINDEX is generated from STARTADDR, and represents the map block containing the FREE/USED flag for the starting sector. BITINDEX is the index into the array of flags which represents sector STARTADDR. For the number of sectors calculated from the passed SIZE parameter, sectors are set to FREE, and all affected map blocks are rewritten to the disc. When the number of blocks set to FREE is equal to the number of blocks the calling routine sought to release, the routine returns to the caller.

5.5.1.3 RENAME

GENERAL

The RENAME procedure allows the user to rename disc files, and affects only the disc directory entry for the specified file. Files which are currently open or are protected may not be renamed.

USER INTERFACE STUB

The user interface stub transfers the following information into the ECB:

EFLAG	-	event flag
STATUSPTR	-	pointer to STATUS parameter of the user call
OLDFILENAME	-	ASCII description of device, name and extension for file to be renamed
NEWFILENAME	-	ASCII description of device, name and extension to be given to the specified oldfile.

DETAILED DESCRIPTION

When a request to rename a file is made, the ECB containing the call parameters for that request is enqueued for the RENAME task by routine RENAMESYNC and RENAME removes the ECB from its input queue.

RENAME then uses several subordinate routines which are shown in Figure 5-24 and described briefly here. First, FINDDEVICE is called to get the pointer to the disc device descriptor block. The chain of File Descriptor Blocks linked to the disc DDB is then examined, to determine whether or not the file is currently open. If the file is open, the function is terminated and error ERFILEOPEN is returned to the user task. RENAME then allocates buffer space for a disc directory block, and routine FINDDIRECTORYENTRY is called to search the disc directory for the specified file. If the file is not found, the function is terminated and error ERNOFILE is returned to the user. The attributes of the file are examined to determine whether the file is protected or unprotected; if PROTECTED, the file may not be renamed, and the function terminates with error ERPROTECTED returned to the user task. The file is renamed by placing parameter NEWFILENAME in the FILENAME of the directory entry and rewriting the directory block to disc. RENAME signals EVENTCOMPLETE to awaken the user task, and releases its allocated buffer space. RENAME then awaits further rename requests on its input queue.

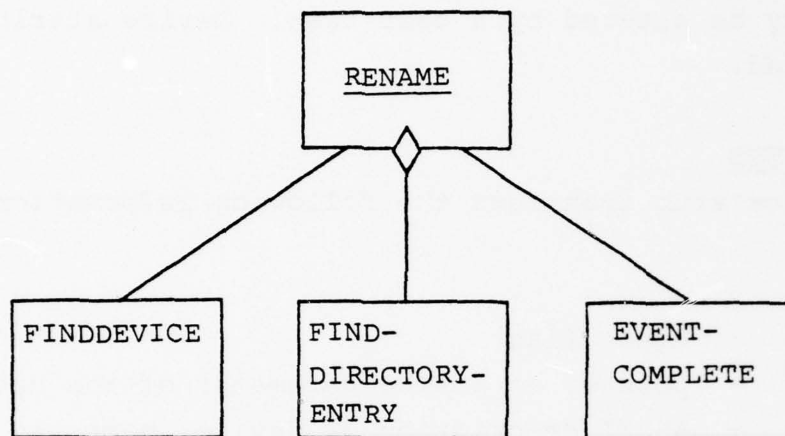


Figure 5-24 RENAME

5.5.1.4 SETATTRIBUTES

GENERAL

The SETATTRIBUTES procedure allows the user to set attributes of disc files. Only the attributes of READPROTECT, WRITEPROTECT and PROTECTED may be altered by a user task. Device attributes may not be altered.

USER INTERFACE STUB

The user interface stub transfers the following information into the ECB:

EFLAG	-	event flag
STATUSPTR	-	pointer to STATUS parameter of the user call
FILENAME	-	ASCII description of device, name and extension for the file whose attributes are to be altered
ATTRIBUTES	-	Boolean values of READPROTECT, WRITEPROTECT and PROTECTED to be set for the specified file

DETAILED DESCRIPTION

When a request to set the attributes of a file is made, the ECB containing the cell parameters for that request is enqueued for the SETATTRIBUTES task by routine SETATTRSYNC and SETATTRIBUTES removes the ECB from its input queue.

SETATTRIBUTES then uses several subordintate routines which are shown in Figure 5-25 and described briefly here. First, FINDDEVICE is called to get the pointer to the disc Device Descriptor Block. The chain of File Descriptor Blocks linked to the disc DDB is then examined, to determine whether or not the file is currently open. If the file is open, the function is terminated and error ERFILEOPEN is returned to the user task. SETATTRIBUTES then allocates buffer space for a disc directory block, and routine FINDDIRECTORYENTRY is called to search the disc directory file for the specified file. If the file is not found, the function is terminated and error [illegible] is returned to the user. The attributes of the file are replaced by the new attributes specified in the user call.

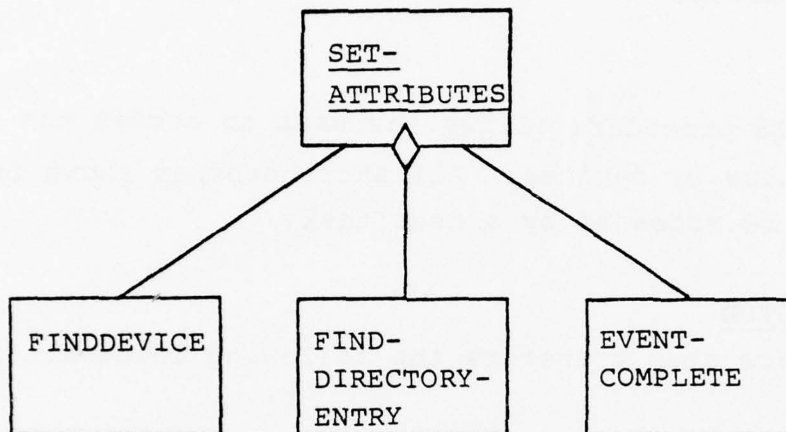


Figure 5-25 SETATTRIBUTES

SETATTRIBUTES then rewrites the directory block to disc, signals EVENTCOMPLETE to awaken the user task, and releases its allocated buffer space. The task then awaits further SETATTRIBUTES requests in its input queue.

5.5.1.5 GETATTRIBUTES

GENERAL

The GETATTRIBUTES procedure allows the user to access the attributes of disc files or devices. All attributes, as shown in the DDB and FDB, may be accessed by a user task.

USER INTERFACE STUB

The user interface stub transfers the following information into the ECB:

SUSPEND	- Event Flag supplied by user interface stub
STATUSPTR	- Pointer to STATUS parameter of the user call
FILENAME	- ASCII description of device, name and extension for the file whose attributes are to be accessed
ATTRIBUTESPTR	- Pointer for returned Boolean values of:

READPROTECT
WRITEPROTECT
PROTECTED
DIRECTORY
BLOCKED
SEGMENTED
CONTIGUOUS
SEQUENTIAL
AVAILABLE

as shown for the specified file or device.

DETAILED DESCRIPTION

When a request to get the attributes of a file is made, the ECB containing the call parameters for that request is enqueued for the GETATTRIBUTES task by routine GETATTRSYNC. GETATTRIBUTES removes the ECB from its input queue and checks to see if the FILENAME specified is a device or a named disc file. If a device was specified, the attributes of the device are moved from the Device Descriptor Block into the user's return value.

GETATTRIBUTES then uses several subordinate routines which are shown in Figure 5-26 and described briefly here. If a request is for a disc file, routine FINDDEVICE is called to get the pointer to the disc DDB. The chain of File Descriptor Blocks linked to the disc DDB is then examined, to determine whether or not the file is currently open. If the file is open, the attributes are moved to the user's return parameter from the files FDB. Otherwise, GETATTRIBUTES allocates buffer space for a disc directory block, and routine FINDDIRECTORYENTRY is called to search the disc directory file for the specified file. If the file is not found, the function is terminated and error ERNOFILE is returned to the user. The attributes of the file are transferred to the user's return parameter from the disc directory entry, if the file was found. GETATTRIBUTES then signals EVENTCOMPLETE to awaken the user task, and releases its allocated buffer space. The task then awaits further GETATTRIBUTES requests in its input queue.

5.5.2 Input/Output Operations

Input and output operations within VTS/OS are performed on the basis of a logical data path, termed a channel. This logical path is established by means of the OPEN call; thereafter, any requests involving the device or file assigned to that channel are specified by referencing the channel number, including termination of the path by the CLOSE call.

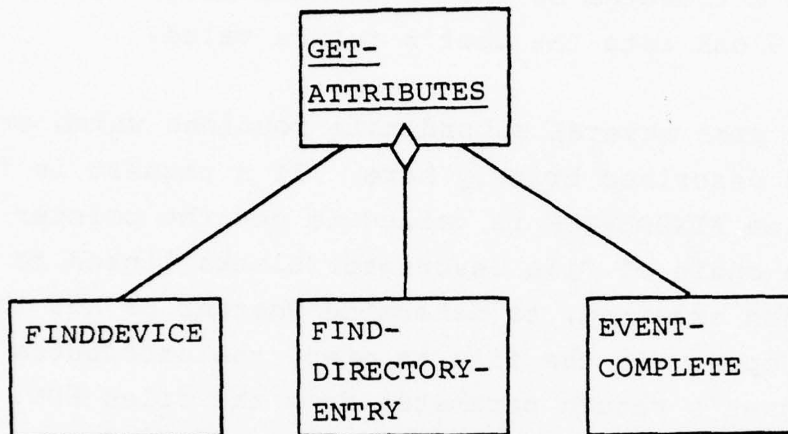


Figure 5-26 GETATTRIBUTES

Two modes of data access are supported by the I/O system calls of the VTS Operating System. Sequential I/O requests treat the device or file as a linear data stream, where the position of data in the stream cannot be dictated by the user task. Random access I/O applies only to disc files, and allows the user to transfer data to and from any point within the file. Certain restrictions are placed on the use of these access modes in conjunction with specific device types or disc file structures; the disc file restrictions are described in Section 5.5.1. The restrictions involving device I/O modes result from the nature of the device, e.g., since a line printer does not generate data, any attempt to perform a read operation from the printer will result in an error return. Device restrictions are generally implemented by protection attributes in the Device Descriptor Block (which are transferred to the File Descriptor Block when the channel is opened).

Figure 5-27 shows the interrelationships between the major data structures used to control I/O operations. The channel table contains pointers to File Descriptor Blocks for all devices or files which are currently open. The presence of a NIL pointer in the channel table indicates that that particular channel is closed. The FDB contains the FILENAME used to open the channel (see Section 5.5.1), the channel number for backward reference from search functions, and a pointer to the DDB for that file or device. There is also an FDBQLINK field, which is used for disc files to link all open files on a particular disc. The DDB has the FDBQHEAD field used for this queue of FDB's; for all devices other than discs, only one FDB may be linked to the FDB queue.

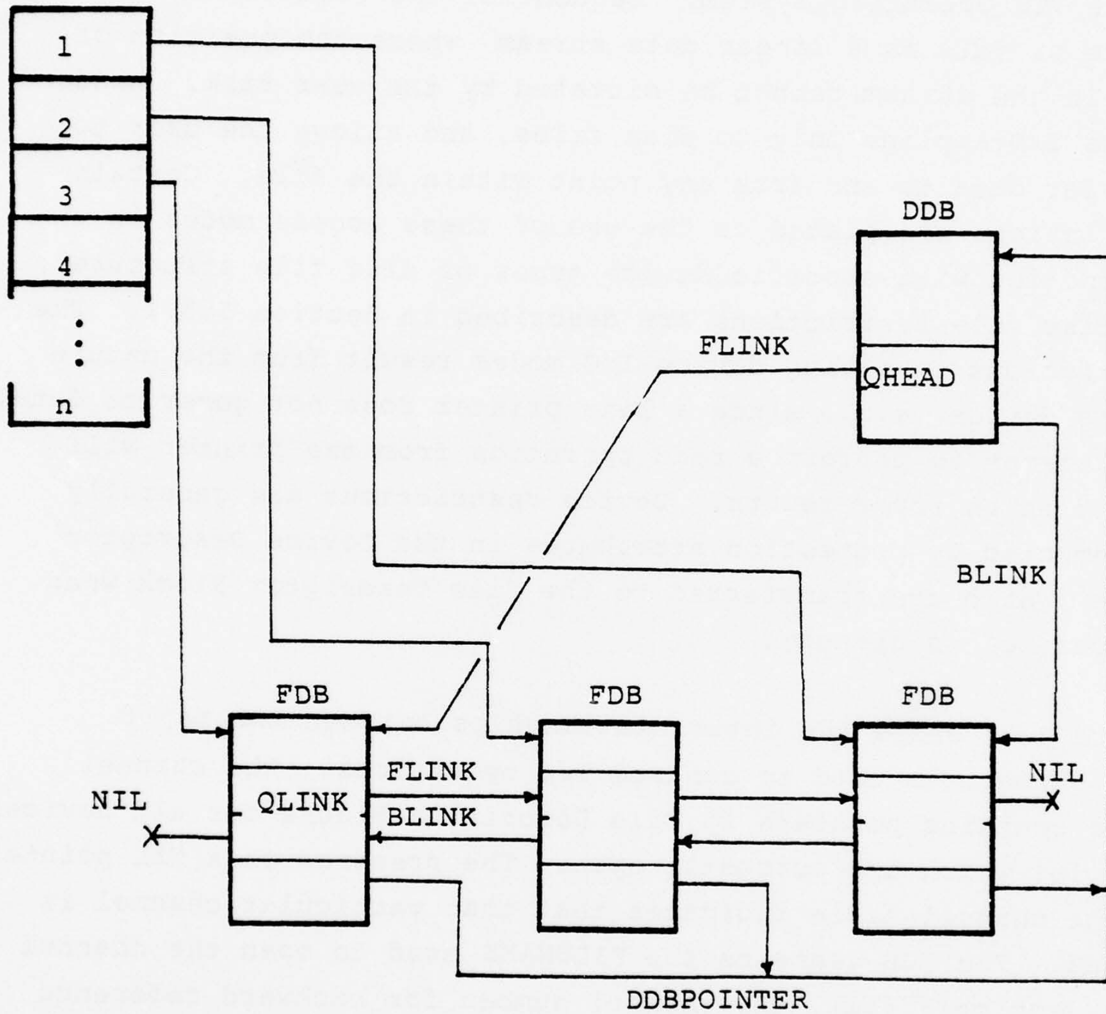


Figure 5-27 Channel Data Structure

5.5.2.1 OPEN

GENERAL

The OPEN system call establishes the logical path between the user task and the device or file to be opened. A channel number unique to the processor is assigned by the OPEN routine. Any initialization of a device which may be necessary to prepare it for I/O operations is performed when a channel is opened to that device.

USER INTERFACE STUB

The user interface stub places the following information in the request ECB:

EFLAG	- event flag parameter
FILENAME	- ASCII description of the device, FILENAME and extension of the device or file to be opened
CHANNELPTR	- A pointer to the user task's CHANNEL parameter, in which the assigned channel number for this logical data path will be returned to the user.

DETAILED DESCRIPTION

The synchronous OPEN structure chart is shown in Figure 5-28. The synchronous OPEN routine is called by IOFUNCTIONS, with a pointer to the request ECB as a parameter. OPEN then scans through the channel table to ensure that a free channel exists, and places the number of the first free channel encountered in the user's channel parameter. If no free channel exists, the function is terminated and error ERNOCHANNEL is returned to the user task. Next, OPEN calls routine FINDDEVICE to get the DDB pointer for the specified device or file, and checks to see if the device is directory structured (disc) or another device. If the device is not a disc, the FDBQHEAD of the DDB for that device is examined to see if the device (or, for magnetic tape, a numbered file on the device) has already been opened. If so,

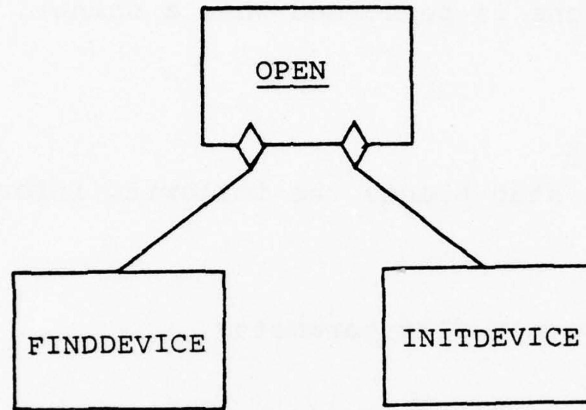


Figure 5-28 OPEN

ERFILEOPEN is returned to the user task and the function terminates. Next, the blocked attribute of the DDB is checked to see if the device is block-structured. For block-structured devices (disc or tape), the open function will generally be performed asynchronously; a dummy FDB is entered in the channel table to prevent use of the channel before OPENASYNC can perform the open. The ECB is placed on the input queue of the OPENASYNC task, and OPENASYNC is notified via EVENTCOMPLETE that an ECB has arrived for processing. Processing by the OPEN routine is complete for disc or tape files at this point.

For other devices, OPEN proceeds to check that the device has not already been opened. If the forward link field of the DDB's FDBQHEAD field is not equal to NIL, the function terminates and error ERFILEOPEN is returned to the user task. The AVAILABLE attribute is set true, in case the device failed or was removed for maintenance prior to this OPEN system call, and INITDEVICE is called to perform any initialization required for this device. Should an error be detected during the device initialization process, that error is returned in the status parameter of the INITDEVICE call; the error field will contain ERDEV, and VALUE1 and VALUE2 will contain pertinent status register information from the device interface hardware. This information is returned to the user task, and OPEN terminates processing after setting the AVAILABLE attribute in the DDB to FALSE.

An FDB is then allocated to represent the device being opened. The FILENAME and ATTRIBUTES fields of the FDB are filled in from the DDB, and the error field of the STATUS parameter is set to NOERROR to indicate to CALLCOMPLETE that the function has completed synchronously. OPEN then returns to IOFUNCTIONS.

OPENASYNC uses several subordinate routines which are shown in Figure 5-29 and described briefly here. First, OPENASYNC calls FINDDEVICE to regain the DDB pointer, and checks the directory attribute in the DDB to determine whether the device for the requested file is disc or tape. If the device is a disc, the FDB queue of that disc is checked to ensure that

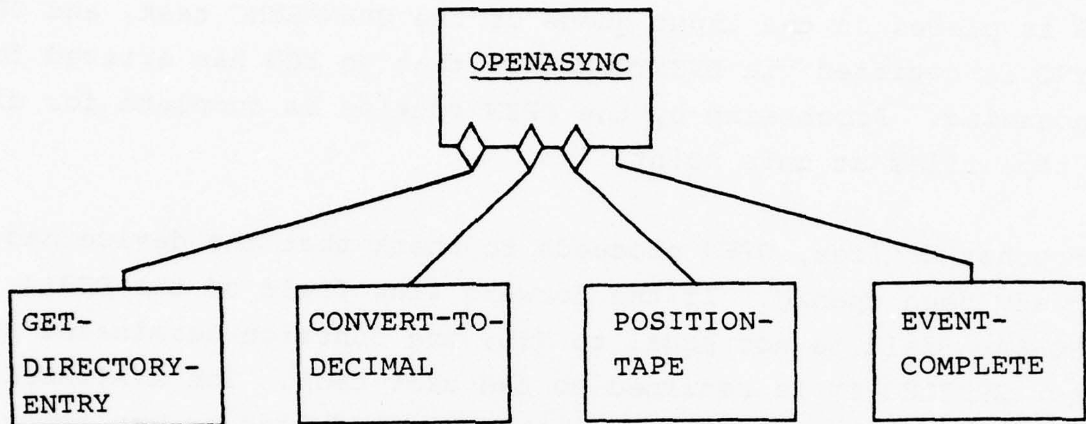


Figure 5-29 OPENASYNC

the file is not already open. If the file is open, ERFILEOPEN is returned to the user task and the function terminates. Next, OPENASYNC obtains a buffer into which GETDIRECTORYENTRY reads directory blocks from the disc in the process of finding the directory entry for the specified file. If the file is not found in the disc directory, ERNOFILE is returned to the user task and the function terminates. The FDB for this channel is now obtained from the free FDB queue. The information from the disc directory entry for this file is copied into the DISCDIRECTORYENTRY area of the FDB. The channel number found in the search performed by routine OPEN is then placed in the CHANNEL field of the FDB, and the entry in the CHANNEL table corresponding to this channel is set to point to the allocated FDB. The FDB DDBPOINTER field is set to point to the DDB, and the FDB is linked to the DDB's FDB queue. The sequential read and write file addresses are then set as indicated by the disc file type, and the TCB pointer of the calling user task is entered into the FDB. The error field of the return status for the OPEN call is set to NOERROR, and EVENTCOMPLETE is called to awaken the user task.

If the request indicated a tape file is to be opened, OPENASYNC calls function CONVERTTODECIMAL to return the numeric value of the tape file. A return of NIL from this routine indicates a non-numeric file name, and causes ERFILENAME to be returned to the user task. Routine POSITIONTAPE is then called, to interface with the magnetic tape driver. POSITIONTAPE will skip to the file designated numerically in TAPEFILE, looking for a double end-of-file. If this logical end of tape is found, EREOF is returned by POSITIONTAPE, or if a device error is returned, that error is passed to the user task and the device is marked unavailable. Otherwise, the FDB is allocated and filled in as for other non-directory devices. NOERROR is returned to the user task and the OPEN function is complete.

5.5.2.2 CLOSE

GENERAL

The CLOSE system call allows the user task to cancel the channel assignment performed by OPEN. For block structured devices, CLOSE must check to see whether the last block used in sequential operations must be written to the device. For magnetic tapes, CLOSE supplies the double end-of-file indicating the logical end of the tape if the file was written during this channel assignment. All resources which were allocated for this channel and its I/O requests are returned to the free state by the CLOSE routine.

USER INTERFACE STUB

The User Interface Stub places the following information in the request ECB:

EFLAG	- Event Flag
STATUSPTR	- Pointer to the user's STATUS parameter
CHANNEL	- The number of the channel to be closed

DETAILED DESCRIPTION

The synchronous portion of the CLOSE system call is called by IOFUNCTIONS with a pointer to the request ECB. The channel table is examined to ensure that the designated channel is open; if not, ERNOTOPEN is returned to the user task and the function is terminated. CLOSE then checks the attributes in the FDB assigned to that channel to see if the device is block structured. If so, sequential I/O may have been performed which has not yet resulted in the writing of the final block to the device; CLOSE then enqueues the request ECB for task CLOSEASYNC, which will complete the close operation for blocked devices.

For unblocked devices, CLOSE sets the channel table entry to NIL to release the channel assignment. The FDB in use for

this channel is removed from the FDB queue in the DDB, and the FDB is returned to the free FDB queue. Close sets the user's STATUS error return to NOERROR to indicate successful closing of the channel, and returns to IOFUNCTIONS.

Task CLOSEASYNC dequeues the request RCB from its input queue and checks the directory attribute in the FDB to separate disc files from magnetic tape files. If the channel was opened to a disc file, the sequential I/O pointers are checked to see if a block needs to be written to the disc, and if this is the case the block is written by the device driver. The buffer used for sequential I/O is then returned to the free buffer list described in the READWRITE routine description, Section 5.5.2.4. If the disc file is segmented, the Segment Pointer Block buffer is also returned to the free buffer list. CLOSEASYNC then sets the channel table entry to NIL, unlinks the FDB from the DDB's FDB queue, and returns the FDB to the free FDB list. NOERROR is returned to the user task, which is notified by EVENTCOMPLETE that the close operation has terminated. (See Figure 5-30 for the CLOSEASYNC structure chart.)

For magnetic tape files, CLOSEASYNC checks the sequential I/O pointers in the FDB to see if a final tape block should be written. If so, the remainder of the block is filled with zeroes and is written to the tape by the device driver. The sequential I/O buffers used for this channel's I/O are released to the free buffer list maintained by READWRITE, and the channel table entry is set to NIL. The FDB is unlinked from the DDB's FDB queue, and is returned to the free FDB list. NOERROR is returned to the user task by EVENTCOMPLETE, which notifies the user that the close request has been performed.

For either disc or tape files, only I/O errors detected by the device driver in writing data to the device are returned to the user with ERDEV in the ERROR field of STATUS, and with pertinent device status indications in the VALUE1 and VALUE2 fields.

AD-A076 501 INTERNATIONAL COMPUTING CO BETHESDA MD F/G 9/2
VESSEL TRAFFIC SERVICES PROCESSING/DISPLAY SUBSYSTEM DETAILED S--ETC(U)
JUL 79 D A COHN , F T MICKEY DOT-CG-81-78-1833
UNCLASSIFIED USCG-D-66-79 NL

3 OF 4
AD A
076501

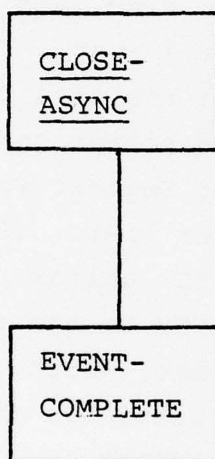


Figure 5-30 CLOSEASYNC

5.5.2.3 IORESET

GENERAL

IORESET will close all files or devices opened by a designated task. Since IORESET is primarily supplied to be used in killing a task, no errors from the COSE routine are returned to the user.

USER INTERFACE STUB

The user interface stub places the following information in the request ECB:

TASKID - ID of the task for which all channels are to be closed

DETAILED DESCRIPTION

IORESETASYNC receives the request from IORESET on its input queue. Routine GETTCBPOINTER is called to return the TCBPTR of the designated task, or NIL if no such TASKID is found. ERTASKID is returned to the user task and the function is terminated if TCBPTR is found to be NIL otherwise, IORESET scans the channel table to find FDB's containing the TCBPTR of the specified task. For each match found, a CLOSE request is issued for than channel. The IORESET function is complete when all channels for that task have been closed, and calls EVENTCOMPLETE to awaken the user task. (See Figure 5-31 for the IORESET structure chart.)

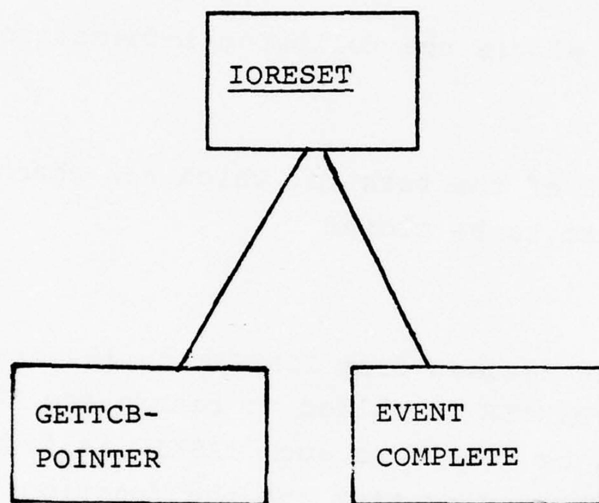


Figure 5-31 IORESET

5.5.2.4 READ AND WRITE OPERATIONS

GENERAL

This task handles processing for READR, WRITER, READSQ, WRITESQ, READLN and WRITELN requests. Data is read or written according to the description for the requested function as presented in Section 4.5.5.

USER INTERFACE STUB

The user interface stub places the following information in the request ECB:

EFLAG - event flag
STATUSPTR - pointer to the user's STATUS parameter
CHANNEL - the number of the channel to be closed
BUFFER - pointer to the user's data area
WORDS - for READR, WRITER, READSQ and WRITESQ:
    number of words to be transferred;
  for READLN and WRITELN:
    maximum number of words to be transferred if no carriage return or null is encountered
FILEADDR - (present for READR and WRITER only) the word within the disc file at which the specified transfer is to begin.

DETAILED DESCRIPTION

The synchronous portion of the READWRITE system task is called by IOFUNCTIONS with a pointer to the request ECB. The channel table is examined to ensure that the designated channel is open; if not, ERNOTOPEN is returned to the user task and the function is terminated. READWRITE then checks the AVAILABLE attribute of the Device Descriptor Block to see if an I/O error has caused the device to be removed from the system for maintenance. If AVAILABLE is false, ERNOTAVAIL is returned and the function is

terminated. READWRITE then checks the protection attributes of the FDB; if the read or write protection conflicts with the user's request, the error message for that protection type is returned and the function is terminated. (The user must close the channel and re-open it before the AVAILABLE attribute will be re-evaluated.)

In the absence of errors, READWRITE links the ECB to the request queue of routine READWRITEASYNC, which will complete processing of the request. The IOINPROGRESS flag of the user's buffer area is set, to prevent release of the buffer while I/O is in progress.

READWRITEASYNC has three input queues which it must supervise, to provide overlapping of I/O requests by multiple users for multiple devices or files. READWRITEQ contains user request ECB's which have been submitted by READWRITE for initiation; FURTHER-ACTIONQ contains FDB's for operations which are partially complete; and COMPLETEQ contains FDB's for actions which have completed. Thus, READWRITEASYNC can perform initiation, continuation or completion checking and notification for a request regardless of functions outstanding for other I/O requests.

Before entering its main loop, READWRITEASYNC sets up wait requests for incoming control blocks on all three queues. Inside the main loop, a WAITANY call is issued on the three queues. When input is present on a queue, the event flag for that queue is returned in EFFOUND, and is used to obtain the incoming control block from the queue and reissue the WAITFORTOPENTRY call on that queue. Depending on the event flag found, pointers are set for the request ECB and, for FURTHERACTIONQ and COMPLETEQ, the FDB associated with the request. The STATUS.ERROR field of the ECB is checked at each stage of processing to see if an error has

been found by the device driver; if so, the operation is cancelled, the AVAILABLE attribute of the device is set to false, and the user is notified of the device error.

Processing is then divided into stages of initiation, continuation and completion.

Initiation Stage:

Random-access I/O requests for disc files are checked first to ensure that the requested transfer is completely within the current file. If the request is outside the file, ERFILEADDR is returned to the user and the function terminates. READR and WRITER requests for an integral number of device blocks can be performed directly into the user's buffer area; these requests are set up for the device driver and initiated, with the FDB for the request returned on the COMPLETEQ when the driver has performed its functions. If the READR or WRITER request is not aligned with the device block structure, a system buffer must be allocated for temporary data storage. In both READR and WRITER requests, data must be read from the disc; this request is set up for the device driver, the READR FDB return is on the COMPLETEQ, and the WRITER FDB return is on the FURTHERACTIONQ.

For disc and tape sequential I/O requests, the buffer pointer of the FDB is checked to determine whether or not the system buffer required for sequential I/O has been allocated. If the pointer is NIL, then one of the buffers in the free buffer list is assigned to that channel. If no buffer is available, ERNOBUFFER is returned to the user and the function terminates. (It should be noted that, with buffers and FDB's, the number of open files within processors of the VTS system is known such that a sufficient number of buffers and FDB's can be included in each processor to prevent resource blocking of VTS/OS.) If

this is the first I/O request on a magnetic tape file, the protection status in the FDB is set to prevent the user from mixing read and write operations, and also to notify the CLOSE routine that sequential writes were performed and so the double end-of-file indicating the logical end of the tape must be written when the file is closed. If there is enough data in the sequential buffer to satisfy a sequential read request, or if a sequential write request does not exceed the current sequential buffer, the sequential data is transferred and the operation is complete. If the current buffer space is exhausted by this request, the FDB is enqueued for the device driver to read a new buffer or to write the current buffer, and the operation will proceed once this device operation has completed.

For all other devices, sequential I/O is performed at the device driver level, since the interface hardware is assumed to be of the interrupt-per-character variety. READWRITEASYNC sets up the device driver request in the FDB, indicates that the FDB is to be returned on the COMPLETEQ, and notifies the device driver that the operation must be performed.

Continuation Stage:
Only disc and tape operations require continuation processing. Random requests for read and write are treated first. Data from READR requests which required a system buffer is transferred to the user's data area, and the system buffer is released. READR processing is complete at this point. WRITER requests using system buffers arrive with the previous contents of all affected sectors in the system buffer. The user's data is copied into the appropriate subsection of the system buffer, and READWRITEASYNC passes a request to the device driver to write the system buffer back onto the disc. Return from this request will be on the COMPLETEQ.

Sequential operations are performed next. For sequential read operations, data has been read into the sequential I/O buffer and must be transferred to the user's data area. Should the data in the buffer not satisfy the sequential word count, the FDB WORDSTRANSFERRED count is updated, and a read request is issued to the device driver for the next portion of the file. If the operation is complete, the sequential read request is terminated successfully. For sequential write operations, data has been written to the file, but the transfer is incomplete. If the file is a segmented disc file, additional disc space is allocated by GETDATASEGMENT before further data is transferred into the sequential buffer. The remaining data from the user's data area is transferred into the system buffer until the buffer is full or the requested amount of data has been transferred. If the buffer becomes full with more data to be transferred, the count of words transferred is updated and a request is made to the device driver to write the buffer to the device. Return from this driver request will be on the FURTHERACTIONQ. If the request amount of data has been transferred, the sequential write function has been completed.

Completion Stage:
The only check made in the completion stage of processing is for WRITER operations which required a system buffer. The write operation is complete, and so READWRITEASYNC releases the system buffer. The IOINPROGRESS flag for the user's buffer is reset on completion of all requests, and EVENTCOMPLETE is called to notify the user task of the completion of his operation.

5.5.2.5 Device Drivers

The device driver implementation for VTS/OS will depend upon the hardware selected for the system. The nature of interrupt handling and of device controller hardware will determine the structure of the drivers and what functions they must perform, but they may be described in terms of the capabilities relied upon by the input/output system calls of VTS/OS.

The following assumptions are made about the devices in the system:

- The magnetic tape and disc device controllers possess Direct Memory Access capability, and must be presented with the memory address to be used in the transfer and the number of words in the transfer;

- The disc device controller also requires the disc address at which the transfer begins, specified as the sector, track and surface;

- Serial, non-block structured devices such as consoles, displays, and printers will have interrupt-per-character device controllers, and the device driver must therefore perform the transfer of data to or from memory when an interrupt for that device occurs.

Device drivers possess input queues on which they are passed File Descriptor Blocks containing the request type and all information needed to perform the request. Device drivers will perform retries of operations resulting in device errors for disc and tape, based upon the retry count in the DDB; for write operations, read-after-write verification will be performed. If a tape record is written incorrectly, an end-of-file mark will

be written and a "skip backward" operation performed, to place a gap with no data on the tape where the error occurred, and the data will be written again; this generally will provide recovery from bad areas on the tape, and will not affect the tape's readability. If this process does not produce a successful write within the retry count of the tape DDB, an error is returned. For read operations, the read will be retried the specified number of times and, if unsuccessful, will be aborted with an error returned to the user. There is generally no retry capability for unblocked devices, as the data is serial.

The general operation of a device driver will be to remove the FDB from the input queue and examine the device address in the DDB. The hardware operations for that device will be performed corresponding to the request issued, and the FDB will be linked to the designated RETURNQHEAD for return to the system task issuing the REQUEST. NOERROR is returned in FDBPTR↑.ECBPTR.STATUS.-ERROR if the operation was successful, and if not, ERDEV is returned in its place, with pertinent device status register information in VALUE1 and VALUE2. Device operations which may be requested are READBLOCK, WRITEBLOCK, READSEQUENTIAL, WRITESEQUENTIAL, READLINE and WRITELINE. If the system buffer ID field in the FDB is NIL, then the operation is performed from the user's data area.

Device drivers must take care of setting the map registers of the I/O map in the processor to the map values of the user area or system buffer for this transfer.

The disc device driver must have one special capability, based on the organization of disc controller hardware. If the read or write request covers sectors which exist on two different tracks or surfaces, the disc device driver may be required by the controller hardware to break the transfer into two transfers, and reposition the disc heads or reinitialize the controller between the two transfers which result.

5.6 ERROR REPORTING AND CONTROL

This section describes the detailed design and operation of the VTS/OS error reporting and error control mechanisms. Section 5.6.1 discusses the data structure required for this error control philosophy, and Section 5.6.2 provides the detailed descriptions of the PDL procedures in Section 6.6.2.

The only procedure to which the applications programs have access is FAILUREDETECTED, which is used to report errors to VTS/OS. Procedures LOGN and TASKRECOVERYN are tasks which process incoming messages from FAILUREDETECTED and its subroutines. The Local Error Reporting Center and Master Error Reporting Center procedures, LERCUPDATE and MERCUPDATE, are called by the Executive on a periodic basis. For this reason, only FAILUREDETECTED has a user interface description in Section 4.

5.6.1 Data Structures for Error Control

The major data structure used to control and coordinate error reports is the Failure Control Block (FCB). Each unique error report received by FAILUREDETECTED has an FCB created for it and placed on an FCB queue within that processor. Successive reports of the same error are accumulated in that FCB until a threshold is passed, at which time action is taken to recover from this persistent error condition. LERCUPDATE periodically scans the FCB queue for FCBs representing transient errors, and removes these FCBs from the queue. All failure reports cause their FCBs to be logged to disc.

The other major data structures used to supervise the reconfiguration of the system are described in Section 7.7.1.

5.6.2 Procedure Declarations

There are three levels to the error control software of VTS/OS. The first level consists of FAILUREDETECTED and its support routines. It performs the initial processing and logging of error reports. The second level consists of LERCUPDATE, which periodically assesses the health of its processor. The need for any system-level recovery, such as removing a processor from the system, is communicated to MERCUPDATE at the third level of error control software.

5.6.2.1 FAILUREDETECTED

GENERAL

Procedure FAILUREDETECTED is called by the applications programs to report errors. Error reports are sent via ITC to LOGN, which resides in processor N and places the error FCB on disc. The FCB is then examined to see whether or not the error has been verified and recovery should be attempted.

USER INTERFACE STUB

There is no event flag associated with FAILUREDETECTED, since it is not desirable to allow applications tasks to proceed until the error being reported has been logged and assessed. Similarly, no error return from FAILUREDETECTED is possible, since the error control software handles its own errors internally. The User Interface Stub places the following parameters in the request ECB:

SENDERID	-	Task ID of reporting task
DESTNAME	-	NIL or task name of the other party for ITC errors
ERRORCODE	-	BADDATAPRODUCED, BADDATARECEIVED, MSGNOTANSWERED, UNABLETOINSTALL UNABLETOSCHEDULE;

FAILURENUMBER - Caller-assigned ID for this failure.

DETAILED DESCRIPTION

An FCB is allocated by FAILUREDETECTED to contain information on the current error report. If no unused FCB is available, the oldest FCB is removed from the list of errors being monitored and is used for this error. In operation, this limit of unused FCBs would be set such that the processor causing the flood of errors would be removed from the system before all free FCBs could be allocated. The FCB is filled in with the reported error information, and is sent as a message to procedure LOGN for logging to disc.

FAILUREDETECTED calls two subordinate procedures, FCBNOTFOUND and TASKRECOVERY, as shown in Figure 5-32.

A search is made through the list of active FCBs by function FCBNOTFOUND to see if the error has already been reported. If not, the new FCB is added to the FCB queue. If a matching FCB is found, the information reflecting this error report is updated in the existing FCB and the new FCB is returned to the available queue.

The RETRYCOUNT parameter in the FCB is updated to reflect this error, and is checked against the limit for retries before attempting recovery. If RETRYLIMIT has not been exceeded and the error type is not BADDATAPRODUCED, another attempt of the requested operation may be made by the applications task. Tasks identifying the BADDATAPRODUCED condition are not permitted to retry their operations before recovery. If RETRYLIMIT has been exceeded or bad data produced, the RECOVERYCOUNT parameter is incremented and RETRYCOUNT is set to zero. RECOVERYCOUNT is checked against its corresponding limit to see whether or not this processor should be disconnected from the system. If

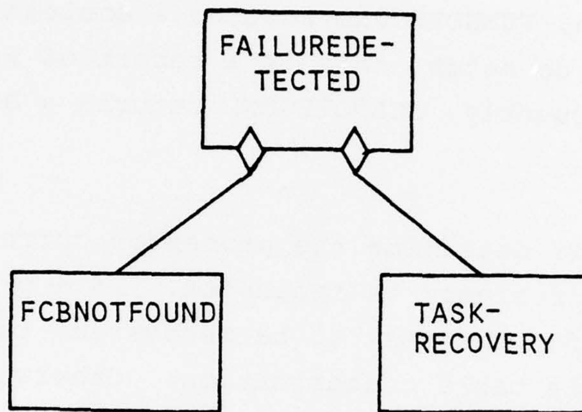


Figure 5-32. FAILUREDETECTED

RECOVERYCOUNT has been exceeded, the processor is disconnected, and the absence of LERC messages to the MERC will signal a system congiguration change. Otherwise, routine TASKRECOVERY is called to instigate recovery of this task.

Function FCBNOTFOUND compares the SENDERID, DESTNAME, ERROR-CODE and FAILURENUMBER fields of the FCB of the reported error to those fields in the FCB pointed to by CURRENTFCB. If any fields do not match, FCBNOTFOUND returns a Boolean value of true. If the FCBs do match, this is a report of an error after a retry, and consequently, FCBNOTFOUND returns a Boolean true to FAILUREDETECTED.

Routine TASKRECOVERY determine the processor currently running the task for which recovery is indicated. If a processor assignment is not found for the task to be recovered, the LERC is notified of possible table contamination. Otherwise, a message is sent to procedure TASKRECOVERYN, where N is the processor number. TASKRECOVERYN will reschedule the task causing the reported error.

Procedure LOGN runs in each of the three processors having discs, where N is the processor number. LOGN in each of these processors maintains a disc log of all reported errors for those processors sending their FCBs to processor N.

5.6.2.2 LERCUPDATE

GENERAL

LERCUPDATE forms the middle level of error control software. The health of the components of each processor is the responsibility of the LERCUPDATE routine executing in that processor.

USER INTERFACE STUB

None.

DETAILED DESCRIPTION

LERCUPDATE is called periodically by the Executive, and begins its processing by running diagnostic routines to verify the functioning of the processor hardware. If any diagnostic check fails, it is run again to verify the error. Verified errors cause LERCUPDATE to remove its processor from the system. If no diagnostic routine finds an error, LERCUPDATE examines the active FCB list to remove those FCBs which have been on the list for longer than the designated error retention time. This prevents sporadic correctable errors from causing a processor to be removed from the system. FCBs which exceed the FCBRETAIN-TIME parameter are returned to the queue of available FCBs.

LERCUPDATE then resets the hardware watchdog timer. The expiration of this timer disconnects its processor from the system bus; by resetting the timer, LERCUPDATE reaffirms its processor's health. Similarly, a message is sent to all MERCs in the system to notify them that this processor is still functioning properly.

5.6.3 MERCUPDATE

GENERAL

Procedure MERCUPDATE exists in all processors in the system, and is called periodically by the Executive. Only one copy is "active," and this copy monitors the overall health of the system. The other copies of MERCUPDATE maintain identical tables but

are effectively slaves, and can become active only if the previously active copy is disabled by a hardware or software failure.

USER INTERFACE STUB

None.

DETAILED DESCRIPTION

Figure 5-33 shows the structure of MERCUPDATE. Upon being called by the Executive, MERCUPDATE determines whether or not the time period (T1) for reception of messages from all the LERCs in the system has expired. If so, these messages are used to update the PCSTATUS table; any processor from which no message was received is marked as "DOWN". If this copy of MERCUPDATE is the active MERC, it calls routine COMPUTECONFIGANDSENDIT to notify the slave copies of the system configuration.

If this copy is not the active copy, MERCUPDATE looks for a configuration message from the active MERC. If the message has arrived, the current configuration number and the map of active Display Station processors are updated to reflect the current configuration. In most cases, this will not represent a new configuration; however, if the configuration has changed, routine RECONFIGURE is called to perform any special processing required by the configuration change.

If no configuration message has been received from the active MERC, the slave copies of MERCUPDATE check to see whether or not their timers for T2PROCESSORNUMBER have expired. These timer values differ for each processor. If the timer has expired, the previous active MERC is assumed to have failed, and this copy of MERCUPDATE marks itself active and calls COMPUTECONFIGANDSENDIT to assume its active role. This activation process is the reason for the differing values of the T2PROCESSORNUMBER timer, since

it is not desirable to have several copies of MERCUPDATE declaring themselves as active MERCs. The values should be selected such that sufficient time passes between their successive expirations that the first MERCUPDATE copy may activate itself and send configuration messages before the next T2PROCESSOR-NUMBER timer expires.

Routine COMPUTECONFIGANDSENDIT examines the UP/DOWN status of the two Main processors and the Position/Collision processor to calculate the column index for the Configuration Table described in Section 7.7.1. The statuses of the Display Station processors are then examined to determine whether or not any of these processors have failed. If the Watch Supervisor's station is down, its functions are reassigned to the first active Display Station processor found by COMPUTECONFIGANDSENDIT. The Watch Supervisor may reassign his station after this occurs, but alerts may be sent to his station before this reassignment can be made, and these alerts should not be lost. The other Display Station processors are then scanned to see if they have failed, and if so, their functions are reassigned to the Watch Supervisor's station. This is again to prevent the loss of any alerts for which a downed Display Station processor may have been responsible. The computed configuration and the table containing Display Station logical assignments are then transmitted to all other MERCUPDATE copies.

Routine RECONFIGURE is called with the new configuration number when a reconfiguration occurs. The Configuration Table for the new configuration is scanned, and all tasks which must be made active are SCHEDULEd. Conversely, all tasks which were active in this processor but are not active under the new configuration are "WOUNDed". These tasks will be allowed to complete processing of any messages on their input queues before being KILLed.

6 PROGRAM DESIGN LANGUAGE SPECIFICATIONS FOR VTS/OS

This chapter contains the Program Design Language specification for all VTS/OS routines other than the Executive itself. The chapter is organized along the same lines as Chapter 5, with a PASCAL-like PDL used to present the actual algorithm for each routine.

It is important to recognize that although an effort has been made to conform to the syntax of PASCAL wherever reasonable in the PDL, actual PASCAL syntax has in some places been knowingly violated on the grounds that it is occasionally less readable, and at other times overly restrictive. Major areas in which the PDL deliberately does not conform to PASCAL are described below.

PASCAL is a very strongly typed language, requiring that every label, data type, variable, procedure, and function be declared before the beginning of the executable code. In general, ICC has not specifically declared every label, data type, variable, procedure and function, since it is generally obvious from the PDL itself when and how such elements are used. Where the possibility exists for ambiguity, declarations have been made at a level of detail sufficient to resolve this ambiguity.

During the implementation of VTS/OS, some of this typing will have to be defeated or weakened. For example, the GETSTATUS routine always returns a three-word variable of type STATUSTYPE, which is defined as three integer variables. However, in certain contexts, one of these variables may contain an address to be used as a pointer, rather than an integer. VTS/OS requires this capability; hence, it has been assumed in the PDL.

In general, when traversing a queue, seeking a node with a specific value in one field of the node, the PDL in this chapter does not conform to pure PASCAL syntax. Let a queue node be defined by:

```
    TYPE NODE= RECORD
                 LINK:  POINTER; /*pointer to next node or NIL*/
                 TARGET:  INTEGER; /*field of interest*/
               END;
```

Then, in this report, the queue traversal is generally coded:

```
PTR := FIRSTNODE;
WHILE PTR <> NIL AND PTR↑.TARGET<>VALUE
    DO PTR := PTR↑.LINK;
IF PTR = NIL
    THEN BEGIN
       /*error:  node not found*/
    END
    ELSE BEGIN
       /*PTR points to desired node*/
    END;
```

Properly coded in PASCAL, this operation would be:

```
PTR := FIRSTNODE;
B := TRUE;
WHILE ( PTR<>NIL ) AND B DO
    IF PTR↑.TARGET = VALUE THEN B := FALSE ELSE PTR := PTR↑.LINK;
IF PTR = NIL
    THEN BEGIN
       /*error:  node not found*/
    END
    ELSE BEGIN
       /*PTR points to desired node*/
    END;
```

In the interests of readability, the former version has been used.

At times, the restrictions imposed by PASCAL have been unacceptable. In such cases, the assumption has been made that these restrictions would be avoided in the actual implementation either by performing small critical sections in assembly language or by modifying the PASCAL compiler. One example of this restrictiveness involves returning values through an address pointer, a capability which is required repeatedly but not available in pure PASCAL.

A specific example of this is the requirement that a VTS/OS system task return a status to another task which has called it. In this case, assuming that the address of the calling task's STATUS variable is in the STATPTR subfield of an Event Control Block, the value is returned in the PDL by:

```
    ECB↑.STATPTR↑.STATUS :=VALUE;
```

Another example of PASCAL's restrictiveness which will have to be overcome can be found in the area of variable storage allocation. PASCAL does not allow the programmer any control whatsoever over the actual allocation of variable storage; however, certain VTS/OS routines, notably in the areas of Input/Output and Inter-Task Communication, require that certain variables be contiguous in memory in order to take advantage of DMA transfers. This will have to be enforced in some manner during the implementation of VTS/OS.

Further differences between the PDL used here and PASCAL lie in the area of record subfields. The PDL used here assumes that if a record field is assigned a specific value, then all subfields of that field are assigned a specific value, and furthermore, that

if a record field in one variable is assigned the values from that record field in another variable, corresponding subfields will receive the appropriate values. For example, let a record and a new variable type be defined as:

```
TYPE NODE= RECORD
               FIELD1:  INTEGER;
               STATUS:  STATUSTYPE;
          END;

TYPE STATUSTYPE= RECORD
               ERROR:  INTEGER;
               VALUE1:  INTEGER;
               VALUE2:  INTEGER;
          END;
```

Then, if ANODE and BNODE are both of type NODE,

```
    ANODE↑.STATUS :=BNODE↑.STATUS;
```

is assumed to be equivalent to:

```
    ANODE↑.STATUS↑.ERROR := BNODE↑.STATUS↑.ERROR;
    ANODE↑.STATUS↑.VALUE1 :=BNODE↑.STATUS↑.VALUE1;
    ANODE↑.STATUS↑.VALUE2 :=BNODE↑.STATUS↑.VALUE2;
```

Finally, since it is not possible to predict to what extent the compiler chosen will perform compile-time and run-time error checking, specific error checks, primarily on the validity of a given value for a given variable, have been written into the PDL. It is entirely possible that these will prove to be unneeded, once an actual compiler is chosen.

6.1 EVENT CONTROL

6.1.1 TIME-INTERVAL WAITS

6.1.1.1 SETDELAY

```
PROCEDURE SETDELAY (ECBPTR)

/*  INPUT:    ECBPTR, THE CURRENT ECB, WHICH SPECIFIES:
                SECONDS - NUMBER OF SECONDS TO SET DELAY FOR -
                MUST BE BETWEEN 1 AND "MAXSEC" - IN ECB^.PARAM1
                EVENTFLAG - FLAG TO USE FOR COMPLETION SIGNALLING
    OUTPUT:   STATUS - SUCCESS OR FAILURE CODE - ONE OF:
                NO-ERROR
                ERR-TIME - NUMBER OF SECONDS SPECIFIED EITHER LESS
                   THAN ONE OR GREATER THAN MAXSEC -
                ERR-DELETED - TIMEOUT DELETED BY REMOVEDELAY CALL

                ECB IS SORTED INTO THE ACTIVE TIMEOUT LIST AND
                THE TIMEOUT VALUES ARE UPDATED TO REFLECT ITS
                PRESENCE - TIMEOUTVALUES ARE KEPT IN PARAM1, BUT
                MODIFIED TO BE INTERVALS BETWEEN SUCCESSIVE TIMEOUTS */

BEGIN
      SECS := ECBPTR^.PARAM1;
      IF SECS > 0 AND SECS <= MAXSEC   /* THEN IS LEGAL */
         THEN BEGIN  /*INSERT ECB ON ACTIVE TIMEOUT LIST*/
            TIMEOUTQPTR := TIMEOUTQHD^.FIRST;
            TIMEREMAINING := 0;      /*ACCUMULATOR FOR TIMEOUT VALUES*/
            WHILE   /*SCAN LIST FOR INSERTION POINT*/
               TIMEREMAINING < ECBPTR^.PARAM1
               AND TIMEOUTQPTR <> NIL
               DO BEGIN /*UPDATE TIME REMAINING AND GET NEXT ENTRY*/
                  TIMEREMAINING := TIMEREMAINING + TIMEOUTQPTR^.PARAM1;
                  TIMEOUTQPTR := TIMEOUTQPTR^.ECBQLINK^.FLINK;
               END;
            /*HAVE FOUND INSERTION POINT IN QUEUE OR END OF QUEUE*/
            ENQBEFORE (TIMEOUTQHD, ECBPTR, TIMEOUTQPTR, ECBQLINK);
            /*ADJUST INCREMENTAL TIMEOUT VALUES*/
            ECBPTR^.PARAM1 := SECS - TIMEREMAINING;
            IF    /*IS THERE A TIMEOUT AFTER THE NEW ENTRY?*/
               ECBPTR^.QLINK^.FLINK <> NIL
               THEN      /*ADJUST TIMEOUT VALUE OF THAT TIMEOUT*/
                  TIMEOUTQPTR^.PARAM1 := TIMEOUTQPTR^.PARAM1 -
                  ECBPTR^.PARAM1;
            ECBPTR^.STATUS^.ERROR := NO-ERROR;
         END      /*END OF TIMEOUT INSERTION*/
         ELSE ECBPTR^.STATUS^.ERROR := ERR-TIME;  /* ERROR IN PARAM1*/
END.
```

6. 1. 1. 2 REMOVEDELAY

```
PROCEDURE REMOVEDELAY ( ECBPTR )

/*   INPUT:    ECBPTR, WHICH SPECIFIES A TIMEOUT THAT HAS BEEN
                 REQUEST EARLIER AND IS NOW TO BE CANCELLED.  THIS IS
                 DONE BY GIVING AS PARAM1 THE EVENTFLAG THAT WAS
                 ASSOCIATED WITH THE ORIGINAL CALL.
     OUTPUT:   STATUS - SUCCESS/FAILURE CODE - ONE OF:
                 NO-ERROR
                 ERR-EFLAG - EVENTFLAG GIVEN AS PARAM1 NOT A VALID EFLAG
                 ERR-WRONGFLAG - NO TIMEOUT WAS ASSOCIATED WITH THE
                    EVENTFLAG GIVEN AS PARAM1
                 ERR-TOOLATE - TIMEOUT HAS ALREADY COMPLETED

     NOTE:  THE ORIGINAL TIMEOUT IS CLEARED BY SETTING THE
                 STATUS TO ERR-DELETED IN THE ECB, WHICH IS THEN SENT
                 TO EVENTCOMPLETE.   */

BEGIN       /* ATTEMPT TO DELETE TIMEOUT */
      /* FIRST MAKE SURE EVENTFLAG IN PARAM1 IS LEGAL */
      EFLAG := ECBPTR^. PARAM1;
      IF EFLAG < EFLAG1 OR EFLAG > EFLAG15 /* IS ILLEGAL */
         THEN BEGIN   /* ERROR RETURN */
            ECBPTR^. STATUS^. ERROR := ERR-EFLAG;
            GO TO 100;   /* COMMON EXIT POINT */
         END; /* CHECK FOR ERR-EFLAG */
      ECBDEL := ECBPTR^. TCBPTR^. EFLAGPTR[EFLAG];
      /* ECBDEL NOW HAS THE ECB TO DELETE - FIRST MAKE SURE
         IT WAS A TIMEOUT ECB TO BEGIN WITH */
      IF ECBDEL = NIL /* THAT EVENTFLAG WASN'T IN USE AT ALL */
      OR ECBDEL^. SUBFUNC <> SETDELAY /* IT WAS, BUT NOT TIMEOUT */
         THEN BEGIN   /* ERROR RETURN */
            ECBPTR^. STATUS^. ERROR := ERR-WRONGFLAG;
            GO TO 100;   /* COMMON EXIT POINT */
         END;   /* CHECK FOR ERR-WRONGFLAG */
      IF ECBDEL^. STATUS^. ERROR <> REQUEST-IN-PROGRESS
         THEN BEGIN /* CAUGHT AN ERR-TOOLATE */
            ECBPTR^. STATUS^. ERROR := ERR-TOOLATE;
            GO TO 100;
         END  /* ALL CHECKING FOR ERRORS */
         ELSE BEGIN  /* ACTUAL DELETION PROCESS */
            IF ECBDEL^. QLINK^. FLINK <> NIL  /* THERE IS A FOLLOWING
               TIMEOUT, SO UPDATE ITS DELTA-TIME */
               THEN ECBDEL^. QLINK^. FLINK^. PARAM1 :=
                  ECBDEL^. QLINK^. FLINK^. PARAM1 +
                  ECBDEL^. PARAM1;
            /* NOW REMOVE THIS ECB FROM THE TIMEOUT-Q */
            DELINK(ECBDEL, TIMEOUTQHD, ECBDEL^. ECBQLINK);
            ECBDEL^. STATUS^. ERROR := ERR-DELETED;
            EVENTCOMPLETE(ECBDEL);
            /* THAT TOOK CARE OF THE DELETED ONE; NOW SET
               STATUS FOR THIS CALL */
```

```
            ECBPTR^.STATUS^.ERROR := NO-ERROR;
         END; /* ACTUAL DELETION */
100:  /* COMMON EXIT POINT USED IF ERRORS CAUGHT ABOVE */
END.
```

6.1.1.3 TIMEOUTASYNC

```
        PROGRAM TIMEOUTASYNC

      /* THIS IS THE ASYNCHRONOUS TIMEOUT TASK.  IT
         DECREMENTS THE TIMEOUT VALUE OF THE ECB AT THE HEAD
         OF THE ACTIVE TIMEOUT LIST.  IF THE RESULTING VALUE
         IS ZERO, SIGNALS EVENT COMPLETION FOR ALL ECB'S
         AT THE HEAD OF THE ACTIVE TIMEOUT LIST WHOSE
         TIMEOUT VALUES ARE ZERO*/

BEGIN
      WHILE TRUE    /*INFINITE LOOP*/
      DO BEGIN
         WAITFORTOPENTRY (SUSPEND, REALTIMECLOCKQID, STATUS);
         /*THE ENTRY IN THE REAL-TIME CLOCK QUEUE SERVES ONLY
            TO INDICATE THE OCCURRENCE OF AN INTERRUPT, AND
            CARRIES NO PERTINENT INFORMATION.  THE ECB IS NOT
            RELEASED BY TIMEOUTASYNC, BUT IS REUSED ENDLESSLY
            BY THE REAL-TIME CLOCK INTERRUPT ROUTINE*/
         TIMEOUTQPTR := TIMEOUTQHD^.FIRST;
         IF       /*NO PROCESSING IF TIMEOUT LIST IS EMPTY*/
            TIMEOUTQPTR <> NIL
         THEN BEGIN     /*DECREMENT TIMEOUT VALUE FOR FIRST TIMEOUT*/
            TIMEOUTQPTR^.PARAM1 := TIMEOUTQPTR^.PARAM1 - 1;
            WHILE TIMEOUTQPTR <> NIL  /* THERE IS AN ENTRY */
            AND TIMEOUTQPTR^.PARAM1 = 0  /* THAT HAS EXPIRED */
            DO BEGIN  /* SIGNAL EVENT COMPLETION */
               TIMEOUTQPTR^.STATUS^.ERROR := NO-ERROR;
               DELINK(TIMEOUTQPTR, TIMEOUTQHD, TIMEOUTQPTR^.ECBQLINK);
               OLDPTR := TIMEOUTQPTR;
               TIMEOUTQPTR := TIMEOUTQPTR^.QLINK^.FLINK;
               EVENTCOMPLETE (OLDPTR);
            END;  /* SIGNALLING */
         END;  /* PROCESSING TIMED-OUT LIST ENTRIES */
      END;       /*END OF INFINITE LOOP*/
END.
```

6.1.1.4 PAUSE

```
PROCEDURE PAUSE(ACTIVEECB)
/*  INPUT:    ACTIVEECB - IDENTIFIES(IN TCBPTR) THE CALLING
              TASK
    OUTPUT:   NONE - THE CALLING TASK IS MOVED FROM THE HEAD
              OF THE READY-Q TO THE TAIL OF THE READY-Q */

BEGIN
      TCB := ACTIVEECB^.TCBPTR;
      DELINK(TCB,RDYQHD,TCB^.TCBQLINK);
      LINK(TCB,RDYQHD,TCB^.TCBQLINK);
      ACTIVEECB^.STATUS^.ERROR := NO-ERROR;
END.
```

6.1.2 EVENT-TERMINATED WAITS

6.1.2.1 WAITANY

```
PROCEDURE WAITANY(ACTIVEECB)
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              SETOFEVENTS - THE OR'ED TOGETHER BITFLAGS OF THE
                EVENTS, ONE OF WHICH HE IS TO WAIT FOR - IS PARAM1
    OUTPUT:   OCCURREDEVENT - THE BITFLAG OF THE LOWEST-
                ORDER OR FIRST EVENT TO OCCUR - ADDR IS PARAM2 */

BEGIN
      TCB := ACTIVEECB^.TCBPTR;
      SET := ACTIVEECB^.PARAM1;
      ACTIVEECB^.STATUS^.ERROR := NO-ERROR; /* REALLY, NO
         ERROR IS POSSIBLE WITH THIS CALL */
      TEMP1 := SET AND TCB^.EFLAGSPOSSIBLE;
      /* TEMP1 NOW HAS THOSE EFLAGS IN WHICH HE IS INTERESTED
         AND WHICH HAVE THE POTENTIAL FOR OCCURRING */
      IF TEMP1 = 0 /* THEN NONE CAN HAPPEN */
         THEN BEGIN /* DUMP HIM IMMEDIATELY */
            ACTIVEECB^.PARAM2^.OCCURREDEVENT := 0; /* NONE OCCURRED */
            PAUSE(ACTIVEECB); /* MAKE HIM PAUSE */
         END /* DUMPING HIM */
         ELSE BEGIN /* SOME COULD HAPPEN; SEE IF ANY DID */
            TEMP2 := TEMP1 AND TCB^.EFLAGSACTUAL;
            IF TEMP2 = 0 /* THEN NONE HAS YET */
               THEN BEGIN /* MAKING HIM WAIT */
                  TCB^.WAITTYPE := WAITANY;
                  TCB^.WAITFLAGS := TEMP1;
                  TCB^.OCCURREDEVENTPTR := ACTIVEECB^.PARAM2;
                  DELINK(TCB,RDYQHD,TCB^.TCBQLINK);
                  LINK(TCB,SUSPENDQHD,TCB^.TCBQLINK);
               END /* MAKING HIM WAIT */
               ELSE BEGIN /* ONE HAS OCCURRED: IDENTIFY AND TELL HIM */
                  FOR IFLAG := EFLAG1 TO EFLAG15 DO
                  IF MASKS[IFLAG] AND TEMP2 <> 0 /* IDENTIFIED */
                     THEN BEGIN /* TELL CALLER WHICH */
                        IFMASK := MASKS[IFLAG]; /* GET THE MASK */
                        TCB^.EFLAGSACTUAL := TCB^.EFLAGSACTUAL AND
                           NOT IFMASK;
                        TCB^.EFLAGSPOSSIBLE := TCB^.EFLAGSPOSSIBLE AND
                           NOT IFMASK;
                        TCB^.WAITFLAGS := NIL;
                        ACTIVEECB^.PARAM2^.OCCURREDEVENT := IFMASK;
                        PAUSE(ACTIVEECB);
                        GO TO 100; /* GET OUT OF THE LOOP, OR WE'LL
                           TELL HIM ABOUT ALL THAT HAVE OCCURRED */
                     END; /* TELLING CALLER WHICH */
100:           END; /* IDENTIFYING */
         END; /* SEEING IF ANY OCCURRED */
END.
```

6.1.2.2 WAITALL

```
PROCEDURE WAITALL(ACTIVEECB)
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              SETOFEVENTS - THE OR'ED TOGETHER BITFLAGS OF THE
                EVENTS, ALL OF WHICH THAT ARE POSSIBLE HE IS TO WAIT
                FOR - IS PARAM1
    OUTPUT:   OCCURREDEVENT - THE BITFLAGS OF ALL THE EVENTS THAT
                OCCURRED TO SATISFY THE WAIT - ADDR IS PARAM2 */

BEGIN
      TCB := ACTIVEECB^.TCBPTR;
      SET := ACTIVEECB^.PARAM1;
      ACTIVEECB^.STATUS^.ERROR := NO-ERROR; /* REALLY, NO
         ERROR IS POSSIBLE WITH THIS CALL */
      TEMP1 := SET AND TCB^.EFLAGSPOSSIBLE;
      /* TEMP1 NOW HAS THOSE EFLAGS IN WHICH HE IS INTERESTED
         AND WHICH HAVE THE POTENTIAL FOR OCCURRING */
      IF TEMP1 = 0 /* THEN NONE CAN HAPPEN */
         THEN BEGIN /* DUMP HIM IMMEDIATELY */
            ACTIVEECB^.PARAM2^.OCCURREDEVENT := 0; /* NONE OCCURRED */
            PAUSE(ACTIVEECB); /* MAKE HIM PAUSE */
         END /* DUMPING HIM */
         ELSE BEGIN /* SOME COULD HAPPEN; SEE IF ALL DID */
            TEMP2 := TEMP1 AND TCB^.EFLAGSACTUAL;
            IF TEMP2 <> TEMP1 /* THEN NOT ALL HAVE YET */
               THEN BEGIN /* MAKING HIM WAIT */
                  TCB^.WAITTYPE := WAITALL;
                  TCB^.WAITFLAGS := TEMP1;
                  TCB^.OCCURREDEVENTPTR := ACTIVEECB^.PARAM2;
                  DELINK(TCB,RDYQHD,TCB^.TCBQLINK);
                  LINK(TCB,SUSPENDQHD,TCB^.TCBQLINK);
               END /* MAKING HIM WAIT */
               ELSE BEGIN /* ALL HAVE OCCURRED, SO TELL HIM */
                  TCB^.EFLAGSACTUAL := TCB^.EFLAGSACTUAL AND
                     NOT TEMP1;
                  TCB^.EFLAGSPOSSIBLE := TCB^.EFLAGSPOSSIBLE AND
                     NOT TEMP1;
                  TCB^.WAITFLAGS := NIL;
                  ACTIVEECB^.PARAM2^.OCCURREDEVENT := TEMP1;
                  PAUSE(ACTIVEECB);
               END; /* TELLING HIM */
         END; /* SEEING IF ALL OCCURRED */
END.
```

6.1.2.3 GETSTATUS

```
PROCEDURE GETSTATUS(ACTIVEECB)
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              EVENTFLAG - THE EVENTFLAG WHOSE ASSOCIATED CALL'S STATUS
                THE USER IS INTERESTED IN - NOTE IS PARAM1, AND NOT THE
                USUAL KIND OF EVENTFLAG
    OUTPUT:   STATUS - THREE WORDS IN WHICH TO PUT THE
                STATUS OF THE ASSOCIATED CALL, IF THERE IS NOTHING
                WRONG WITH THIS CALL - IF SOMETHING IS WRONG,
                THE STATUS RETURNED WILL BE FOR THE GETSTATUS CALL
                - ADDR IS IN STATUSPTR */

BEGIN
      EFLAG := ACTIVEECB^.PARAM1;
      IF EFLAG < EFLAG1 OR EFLAG > EFLAG15 /* VALIDITY CHECK */
         THEN BEGIN /* FAILED - GETSTATUS CALL ERROR */
            ACTIVEECB^.STATUS^.ERROR := ERR-EFLAG;
            ACTIVEECB^.STATUS^.VALUE1 := 0;
            ACTIVEECB^.STATUS^.VALUE2 := 0;
            GO TO 100;  /* COMMON EXIT POINT */
         END;
      /* FLAG IS VALID - SEE IF IT HAS AN ASSOCIATED CALL */
      TCB := ACTIVEECB^.TCBPTR;
      IF TCB^.EFLAGPTR[EFLAG] = NIL /* THEN IT DOESN'T */
         THEN BEGIN
            ACTIVEECB^.STATUS^.ERROR := ERR-FLAG-NOT-IN-USE;
            ACTIVEECB^.STATUS^.VALUE1 := 0;
            ACTIVEECB^.STATUS^.VALUE2 := 0;
            GO TO 100;
         END;
      /* IT HAS A CALL - WHATEVER THE STATUS OF THE CALL IS,
         RETURN IT - IF IT IS COMPLETE, WE ALSO CLEAR THE INDICATORS
         IN THE TCB AND RETURN THE ECB TO THE FREE-Q */
      ECB := TCB^.EFLAGPTR[EFLAG];
      ACTIVEECB^.STATUS := ECB^.STATUS;
      IF ECB^.STATUS^.ERROR <> REQUEST-IN-PROGRESS
         THEN BEGIN /* IS COMPLETE - CLEAR TCB INDICATORS */
            EFMASK := MASKS[EFLAG];
            TCB^.EFLAGSPOSSIBLE := TCB^.EFLAGSPOSSIBLE AND NOT EFMASK;
            TCB^.EFLAGSACTUAL := TCB^.EFLAGSACTUAL AND NOT EFMASK;
            TCB^.EFLAGPTR(EFLAG) := NIL;
            /* ALSO RETURN THE ECB */
            LINK(ECB,ECBAVAILQHD,ECB^.ECBQLINK);
         END; /* CLEANING UP */
100:  /* NOW RETURN THE STATUS TO THE USER'S ADDRESS SPACE */
      ACTIVEECB^.STATUSPTR^.STATUS := ACTIVEECB^.STATUS;
END.
```

6.2 PROGRAM AND TASK MANAGEMENT

6.2.1 PROGRAM MANAGEMENT CALLS

6.2.1.1 INSTALL

```
PROCEDURE INSTALLSYNCH(ACTIVEECB)

/*  INPUT:    CURRENTLY ACTIVE ECB - SPECIFIES:
              NAME - ASCII NAME OF PROGRAM TO BE INSTALLED
              EVENTFLAG - BY WHICH CALLING TASK IS TO
                 BE NOTIFIED OF COMPLETION
    OUTPUT:   STATUS - COMPLETION CODE - ONE OF:
              NO-ERROR
              ERR-EFLAG - ILLEGAL EVENTFLAG
              ERR-TAI - PROGRAM ALREADY INSTALLED
              ERR-NMR - SYSTEM RESOURCES REQUIRED NOT AVAILABLE
              ERR-IO - I/O ERROR OCCURRED DURING INSTALLATION -
                 IN THIS CASE, STATUS.VAL1 CONTAINS ACTUAL
                 I/O ERROR CODE

    NOTE:  THIS IS ONLY THE SYNCHRONOUS PORTION OF INSTALL.
           ALL IT CAN DO IS QUEUE THE ECB ON THE INSTALL-
           IN-PROGRESS QUEUE FOR THE ASYNCHRONOUS TASK */

BEGIN
      LINK(ACTIVEECB, INSTALLQHD, ACTIVEECB^.ECBQLINK);
      EVENTCOMPLETE(INSTALLQHD); /* SIGNAL NEW ENTRY ON Q */
      ACTIVEECB^.STATUS^.ERROR := NO-ERROR;
END.
```

```
PROGRAM INSTALL

/*  INPUT:    ECB'S ON THE INSTALLQ, WHICH SPECIFY:
              NAME - NAME OF PROGRAM TO INSTALL IS PARAM1
              EVENTFLAG - BY WHICH REQUESTING TASK IS TO
                 BE NOTIFIED OF COMPLETION
    OUTPUT:   STATUS RETURNED TO REQUESTOR - ONE OF:
              NO-ERROR
              ERR-TAI - PROGRAM ALREADY INSTALLED
              ERR-NMR - SYSTEM RESOURCES NOT AVAILABLE
              ERR-IO - I/O ERROR OCCURRED DURING INSTALL
                 PROCEDURE.  IN THIS CASE, THE TRUE I/O
                 ERROR CODE WILL BE IN STATUS^.VAL1.
    NOTE:   THIS IS THE ASYNCHRONOUS TASK PORTION OF
            THE INSTALL PROCEDURE.  IT IS QUEUE DRIVEN,
            WITH THE QUEUE BEING BUILT BY THE SYNCHRONOUS
            PORTION.  IT IS SINGLE-THREADED.  */

VAR MYSTATUS: STATUSTYPE; HEADER: ARRAY [1..HDRSIZE]
   OF INTEGER; /* A COUPLE OF LOCAL VARIABLES */
BEGIN
      WHILE TRUE DO BEGIN /* INFINITELY EXECUTING TASK */
         WAITFORTOPENTRY(SUSPEND, INSTALLQHD, MYSTATUS);
         /* WAIT FOR SOMETHING TO SHOW UP ON THE Q */
         ACTUAL := MYSTATUS^.VALUE1; /* ACTUAL HAS ECB ADDR */
         STATUS := NO-ERROR;
         VAL1 := 0;
         /* FIRST CHECK FOR ERR-TAI */
         ICB := ICBQHD^.FIRST;
         WHILE ICB <> NIL DO
            IF ICB^.TASKNAME = ACTUAL^.PARAM1
               THEN BEGIN
                  STATUS := ERR-TAI;
                  GO TO 200; /* COMMON EXIT POINT IF NO WAIT REQUIRED */
               END
               ELSE ICB := ICB^.ICBQLINK^.FLINK;
         IF ICBAVAILQHD^.FIRST = NIL
            THEN BEGIN /* HAVE NO ICB'S LEFT */
               STATUS := ERR-NMR;
               GO TO 200; /* ONLY SOME OF THE ERR-NMR'S DIE HERE */
            END
            ELSE BEGIN /* GET A FRESH ICB */
               ICB := ICBAVAILQHD^.FIRST;
               DELINK(ICB, ICBAVAILQHD, ICB^.ICBQLINK);
            END;
         /* NOW OPEN THE FILE */
         OPEN(SUSPEND,ACTUAL^.PARAM1,READ,CHANNEL,MYSTATUS);
         /* NOTE THE ASSUMPTION HERE THAT THE PROGRAM NAME WILL
            CORRESPOND EXACTLY TO THE FILE NAME */
         IF MYSTATUS.ERROR <> NO-ERROR
            THEN BEGIN /* ERROR IN OPENING - MUST SEND BACK */
               STATUS := ERR-IO; /* THE ERROR CODES AND GIVE BACK */
               VAL1 := MYSTATUS.ERROR; /* THE ICB AS WELL */
               /* NOTE PROGRAM NOT FOUND COMES BACK AS AN ERR-IO */
```

```
        LINK(ICB, ICBAVAILQHD, ICB^.ICBQLINK);
        GO TO 200;
     END;
  /* NOW FIND OUT HOW BIG THE ROOT IS BY READING THE HEADER */
  READSQ(SUSPEND,CHANNEL,HEADER,HDRSIZE,MYSTATUS);
  IF MYSTATUS.ERROR <> NO-ERROR
     THEN BEGIN /* CAN'T GET HEADER - MUST DO ALL THE ABOVE */
        VAL1 := MYSTATUS.ERROR;
        CLOSE(EFLAG1,CHANNEL,MYSTATUS); /* AND CLOSE THE FILE -
        NOTE USING EFLAG1 - WAIT WILL BE AT END OF TASK */
        STATUS := ERR-IO;
        LINK(ICB,ICBAVAILQHD,ICB^.ICBQLINK);
        GO TO 100; /* COMMON EXIT POINT IF WAIT REQUIRED */
     END;
```

```
/* IF HERE, WE HAVE OPENED THE FILE AND READ IN THE
   HEADER.  ALTHOUGH THE EXACT FORMAT OF THE HEADER IS
   NOT YET DEFINED, SINCE IT IS LARGELY A FUNCTION OF THE
   COMPILER AND LINKER PROVIDED BY THE SYSTEM, WE ARE MAKING
   CERTAIN ASSUMPTIONS ABOUT THE INFORMATION THAT IT PROVIDES.
   IF THESE PROVE TO BE FALSE, APPROPRIATE MODIFICATIONS
   WILL HAVE TO BE MADE TO THIS TASK.  THESE ASSUMPTIONS
   ARE:
   (1)  THE FORMAT OF AN OBJECT FILE IS AS FOLLOWS:
      ROOTHEADER - ROOTCODE - OLAY1HEADER - OLAY1CODE -
      OLAY2HEADER - OLAY2CODE - . . . - EOF.  IF THERE
      ARE NO OVERLAYS, THE ROOTCODE IS FOLLOWED BY THE
      EOF.
   (2)  EACH HEADER CONTAINS THE SIZE OF THE CODE
      PORTION FOLLOWING IT IN TERMS OF THE NUMBER OF
      MAP REGISTERS REQUIRED FOR THE CODE, NOT THE
      NUMBER OF ACTUAL WORDS OF CODE.
   (3)  THE ROOTHEADER ALSO CONTAINS ADDITIONAL INFORMATION
      ABOUT THE ENTIRE PROGRAM, INCLUDING THE NUMBER OF
      OVERLAYS, THE LENGTH OF THE LONGEST OVERLAY LOAD
      PATH, AND THE NUMBER OF MAP REGISTERS REQUIRED FOR THE
      VARIABLE SPACE OF THE PROGRAM.  */
/* GET REAL MEMORY FOR THE ROOT CODE, AND
   SET UP THE ICB, AS MUCH AS POSSIBLE.  */
GETROOTMEM(ICB,STATUS);
IF STATUS <> NO-ERROR /* THEN THERE'S NO MEMORY */
   THEN BEGIN  /* GIVE BACK ICB AND CLOSE FILE */
      CLOSE(EFLAG1,CHANNEL,MYSTATUS); /* WAIT AT 100 */
      LINK(ICB,ICBAVAILQHD,ICB^.ICBQLINK);
      GO TO 100;
   END;
ICB^.MAPSFORROOT := HEADER[CODESIZE];
ICB^.MAPSFOROLAY := HEADER[MAXOLAYPATH];
ICB^.MAPSFORVAR := HEADER[VARSPACESIZE];
ICB^.TASKNAME := ACTUAL^.PARAM1;
ICB^.MAPSFORBUFF := MAXMAPS - (ICB^.MAPSFORROOT
   + ICB^.MAPSFOROLAY + ICB^.MAPSFORVAR);
ICB^.USECOUNT := 0;
/* NOW WE CAN READ IN THE ROOT CODE - USE A
   SPECIAL READ CALL WHICH ENABLES US TO READ INTO
   MEMORY TO WHICH WE ARE NOT MAPPED */
READCODE(SUSPEND,CHANNEL,HEADER[CODESIZE],ICB^.TASKMAP,
   MYSTATUS); /* PARAMS ARE (EFLAG,CHANNEL,AMOUNT TO READ
   IN MAPREGSFULL,WHERE TO READ INTO(SPECIFIED BY MAPREGS),
   AND STATUS) */
IF MYSTATUS.STATUS <> NO-ERROR
   THEN BEGIN
      VAL1 := MYSTATUS.STATUS; /* GET THIS FIRST */
      CLOSE(EFLAG1,CHANNEL,MYSTATUS); /* FOR OBVIOUS REASONS */
      STATUS := ERR-IO;
      RETURNALLRESOURCES(ICB); /* THIS ROUTINE WITH REMOVE -
      WILL RETURN ICB AND MEMORY */
      GO TO 100;
   END;
IF HEADER[NUMBEROFOVERLAYS] <> 0 /* ANY OLAYS TO DO? */
```

```
            THEN FOR IOLAY := 1 TO HEADER[NUMBEROFOVERLAYS]
            DO BEGIN
               OVERLAYPROC(ICB,HEADER,CHANNEL,MYSTATUS);
               /* THIS WILL TAKE CARE OF
               GETTING MEMORY AND READING IN THE OVERLAY */
               IF STATUS <> NO-ERROR
                  THEN BEGIN /* ERROR IN READING IN OVERLAY */
                     VAL1 := MYSTATUS.VAL1;
                     STATUS := MYSTATUS.STATUS; /* SAVE ERROR CODES */
                     CLOSE(EFLAG1,CHANNEL,MYSTATUS); /* CLOSE FILE */
                     RETURNALLRESOURCES(ICB); /* GIVE BACK ALL STUFF */
                     GO TO 100;
                  END;
            END;

         /* IF WE'RE HERE, ALL OVERLAYS ARE LOADED SUCCESSFULLY */
         CLOSE(EFLAG1,CHANNEL,MYSTATUS); /* CLOSE THE FILE */
         LINK(ICB,ICBQUEUE,ICB^.ICBQLINK); /* LINK ICB TO IN-USE-Q */
100:     /* COMMON EXIT POINT IF WAIT NEEDED */
         WAITANY(EFLAG1,EFGOT);
         GETSTATUS(EFLAG1,MYSTATUS); /* CLEAR THE FLAG */
200:     /* COMMON EXIT POINT IF NO WAIT NEEDED */
         ACTUAL^.STATUS^.ERROR := STATUS;
         ACTUAL^.STATUS^.VAL1 := VAL1;
         EVENTCOMPLETE(ACTUAL); /* SIGNAL THE REQUESTOR */
      END;
END.
```

```
PROCEDURE GETROOTMEM(ICB, STATUS)
/*  INPUT:    ICB WHICH SPECIFIES NUMBER OF PAGES OF
              MEMORY NEEDED
    OUTPUT:   IF AVAILABLE, REAL MEMORY IS ASSIGNED TO
              ICB AND MAPREGISTERS ARE FILLED IN
              STATUS - ONE OF:
              NO-ERROR
              ERR-NMR - MEMORY NOT AVAILABLE */

BEGIN
      IF ICB^.MAPSFORROOT > REMAININGPAGES
         THEN BEGIN /* MEMORY NOT AVAILABLE  */
            STATUS := ERR-NMR;
         END
         ELSE BEGIN
            FILLREGS2(ICB^.MAPSFORROOT, ICB);
            STATUS := NO-ERROR;
         END;
END.

PROCEDURE FILLREGS2(NPAGES, ICB)
/*  INPUT:    NPAGES - NUMBER OF PAGES NEEDED
              ICB - ICB TO FILL IN MAPREGS OF
    OUTPUT:   ICB MAPREGS ARE FILLED IN */

BEGIN
      REMAININGPAGES := REMAININGPAGES - NPAGES;
      IPAGE := FIRST-AVAILABLE-MEMORY-PAGE-INDEX;
      NEXTREG := 0;
      UNTIL NEXTREG = NPAGES DO
         BEGIN
            IF PHYSPAGETABLE[IPAGE] = 0
               THEN BEGIN
                  NEXTREG := NEXTREG + 1;
                  PHYSPAGETABLE[IPAGE] := ICB;
                  ICB^.TASKMAP[NEXTREG] := MAP VALUE ASSOCIATED WITH
                  THIS PHYSICAL PAGE;
               END
               ELSE IPAGE := IPAGE + 1;
         END;
END.
```

```
PROCEDURE OVERLAYPROC(ICB, HEADER, CHANNEL, MYSTATUS)
/*  INPUT:    ICB - OF A PROGRAM BEING INSTALLED
              HEADER - AN ARRAY OF WORKING SPACE
              CHANNEL - I/O CHANNEL FOR PROGRAM CODE
    OUTPUT:   MYSTATUS - INDICATES SUCCESS/FAILURE AT
                 INSTALLING NEXT OVERLAY - ONE OF:
              NO-ERROR
              ERR-NMR - SYSTEM RESOURCES NOT AVAILABLE
              ERR-IO - I/O ERROR IN INSTALL */

BEGIN
      /* FIRST, GET AN OCB */
      IF OCBAVAILQHD^.FIRST = NIL
         THEN BEGIN /* FAILED RIGHT AWAY */
            MYSTATUS^.ERROR := ERR-NMR;
            GO TO 100; /* COMMON EXIT POINT */
         END
         ELSE BEGIN
            OCB := OCBAVAILQHD^.FIRST;
            DELINK(OCB, OCBAVAILQHD, OCB^.OCBQLINK);
            OCB^.NUMBEROFMAPREGS := 0; /* THIS PROTECTS US IN
            CASE WE CAN'T GET REAL MEMORY - ALLOWS US TO USE
            RETURNALLRESOURCES ROUTINE */
            LINK(OCB, ICB^.OCBQHD, OCB^.OCBQLINK);
         END;
      /* NOW READ IN THE OVERLAY HEADER AND SEE WHAT WE NEED */
      READSQ(SUSPEND, CHANNEL, HEADER, HDRSIZE, MYSTATUS);
      IF MYSTATUS^.ERROR <> NO-ERROR THEN GOTO 100;
      /* NOW TRY FOR ENOUGH REAL MEMORY */
      OCB^.NUMBEROFMAPREGS := HEADER[CODESIZE];
      GETOLAYMEM(OCB, MYSTATUS^.ERROR);
      IF MYSTATUS^.ERROR <> NO-ERROR THEN GOTO 100;
      /* GOT MEMORY, SO NOW READ IN THE OVERLAY CODE ITSELF */
      READCODE(SUSPEND, CHANNEL, HEADER[CODESIZE], OCB^.MAPREGS,
         MYSTATUS);
      IF MYSTATUS^.ERROR <> NO-ERROR THEN GOTO 100;
      OCB^.OVERLAYID := HEADER[OVERLAYID];
      OCB^.ICBPTR := ICB;
100:  /* COMMON EXIT POINT */
END.
```

```
PROCEDURE GETOLAYMEM(OCB,STATUS)
/*  INPUT:    OCB WHICH SPECIFIES NUMBER OF PAGES OF
              MEMORY NEEDED
    OUTPUT:   IF AVAILABLE, REAL MEMORY IS ASSIGNED TO
              OCB AND MAPREGISTERS ARE FILLED IN
              STATUS - ONE OF:
              NO-ERROR
              ERR-NMR - MEMORY NOT AVAILABLE - IN THIS CASE,
                        OCB^.NUMBEROFMAPREGS IS ZEROED */

BEGIN
      IF OCB^.NUMBEROFMAPREGS > REMAININGPAGES
         THEN BEGIN /* MEMORY NOT AVAILABLE  */
            STATUS := ERR-NMR;
            OCB^.NUMBEROFMAPREGS := 0;
         END
         ELSE BEGIN
            FILLREGS3(OCB^.NUMBEROFMAPREGS,OCB);
            STATUS := NO-ERROR;
         END;
END.

PROCEDURE FILLREGS3(NPAGES,OCB)
/*  INPUT:    NPAGES - NUMBER OF PAGES NEEDED
              OCB - OCB TO FILL IN MAPREGS OF
    OUTPUT:   OCB MAPREGS ARE FILLED IN */
BEGIN
      REMAININGPAGES := REMAININGPAGES - NPAGES;
      IPAGE := FIRST-AVAILABLE-MEMORY-PAGE-INDEX;
      NEXTREG := 0;
      UNTIL NEXTREG = NPAGES DO
         BEGIN
            IF PHYSPAGETABLE[IPAGE] = 0
               THEN BEGIN
                  NEXTREG := NEXTREG + 1;
                  PHYSPAGETABLE[IPAGE] := OCB^.ICBPTR;
                  OCB^.MAPREGS[NEXTREG] := MAP VALUE ASSOCIATED WITH
                  THIS PHYSICAL PAGE;
               END
               ELSE IPAGE := IPAGE + 1;
         END;
END.
```

6.2.1.2 REMOVE

```
PROCEDURE REMOVE ( ACTIVEECB )
/* INPUT:   ACTIVEECB, WHICH SPECIFIES:
            NAME - ASCII NAME OF A PROGRAM TO BE REMOVED
            FROM THE PROCESSOR - IS PARAM1
            EVENTFLAG - FLAG OF CALLING TASK TO BE SET AT
            COMPLETION OF REMOVE
   OUTPUT:  STATUS - SUCCESS OR FAILURE CODE
            NO-ERROR
            ERR-TNI - PROGRAM IS NOT INSTALLED
            ERR-ALIVE - PROGRAM HAS LIVE TASKS WHICH HAVE NOT
               HAD "KILL" CALLS ISSUED
            ERR-EFLAG - ILLEGAL EVENTFLAG
            ERR-SUICIDE - PROGRAM CANNOT REMOVE ITSELF */
BEGIN
      TASKNAME := ACTIVEECB^.PARAM1;
      ICBVICTIM := ICBQHD^.FIRST; /* ICBVICTIM MUST BE REMOVED */
      WHILE ICBVICTIM <> NIL AND TASKNAME <> ICBVICTIM^.TASKNAME
         DO ICBVICTIM := ICBVICTIM^.ICBQLINK^.FLINK;
      IF ICBVICTIM = NIL
         THEN BEGIN
            STATUS := ERR-TNI;
            GO TO 5;
         END;
      TCB := ACTIVEECB^.TCBPTR;
      ICB := TCB^.ICBPTR;
      IF ICB = ICBVICTIM
         THEN BEGIN
            STATUS := ERR-SUICIDE;
            GO TO 5;
         END;
      TCBPTR := ICBVICTIM^.TCBQHD^.FIRST;
      WHILE TCBPTR <> NIL DO
         IF TCBPTR^.KILLECBPTR = NIL
            THEN BEGIN
               STATUS := ERR-ALIVE;
               GO TO 5;
            END
         ELSE TCBPTR := TCBPTR^.CLONEQLINK^.FLINK;
/* BY THE TIME WE GET HERE, WE HAVE WEEDED OUT THE CASES OF
   (1)  NO SUCH ICB -- ERR-TNI;
   (2)  TASK REMOVING ITSELF -- ERR-SUICIDE;
   (3)  NOT ALL INVOCATIONS OF PROGRAM KILLED -- ERR-ALIVE.
   HOWEVER, THIS DOES NOT IMPLY THAT THERE ARE NO RUNNING
   COPIES OF THE PROGRAM.   */
      TCBPTR := ICBVICTIM^.TCBQHD^.FIRST;
      IF TCBPTR = NIL /* THEN THERE ARE NO RUNNING COPIES */
         THEN BEGIN
            RETURNALLRESOURCES(ICB); /* GIVE BACK EVERYTHING */
            STATUS := NO-ERROR;
         END
         ELSE BEGIN /* THERE ARE, SO SAVE ECB */
```

```
          STATUS := REQUEST-IN-PROGRESS;
          ICBVICTIM^.REMOVEEECBPTR := ACTIVEEECB;
        END;
5:    ACTIVEEECB^.STATUS^.ERROR := STATUS;
END.



PROCEDURE RETURNALLRESOURCES ( ICB )
/*   INPUT:    ICB OF A TASK TO BE REMOVED
     OUTPUT:   NONE -- ALL RESOURCES(OCBS, ICB, AND
               MEMORY) ARE RETURNED TO SYSTEM FREE-LISTS */

BEGIN
    OCBPTR := ICB^.OCBQHD^.FIRST;
    IF OCBPTR <> NIL /* THERE ARE OVERLAYS TO DO */
      THEN BEGIN
        WHILE OCBPTR <> NIL DO BEGIN
          OCBNXT := OCBPTR^.OCBQLINK^.FLINK;
          /* KEEP TRACK OF NEXT ONE WHILE DOING THIS ONE */
          RETURNREALMEMORY ( OCBTYPE, OCBPTR);
          /* THERE GOES MEMORY */
          LINK ( OCBPTR, OCBAVAILQHD, OCBPTR^.OCBQLINK);
          /* AND THERE GOES THE OCB */
          OCBPTR := OCBNXT; /* NOW DO THE NEXT ONE */
        END;
      END; /* THROUGH WITH OCBS HERE */
    RETURNREALMEMORY ( ICBTYPE, ICB ); /* NOW GIVE BACK THE
    MEMORY CONTAINING THE ROOT */
    LINK ( ICB, ICBAVAILQHD, ICB^.ICBQLINK); /* AND THE ICB */
END.
```

6.2.2 TASK MANAGEMENT CALLS

6.2.2.1 SCHEDULE

```
PROCEDURE SCHEDULE ( ACTIVEECB )

/* INPUT:    ACTIVEECB, WHICH SPECIFIES:
             TASKNAME - ASCII NAME OF PROGRAM TO RUN - IN PARAM1
             EVENTFLAG - IF 1-15 OR "SUSPEND", TASK IS CHILD OF
             CALLER AND POST USES THIS FLAG; IF "INDEPENDENT",
             TASK IS INDEPENDENT OF CALLER

   OUTPUT:   TASKID - ID OF SCHEDULED TASK; MAY BE USED
             FOR KILL - ADDR IN PARAM2
             STATUS - COMPLETION CODE
             IF EFLAG IS 1-15 OR INDEP:
                NO-ERROR
                ERR-PERMIT - PROGRAM FLAGGED FOR REMOVE
                ERR-TNI - TASK NOT INSTALLED
                ERR-EFLAG - ILLEGAL EVENTFLAG
                ERR-NMR - NO MORE ROOM IN MEMORY
             IF EFLAG IS SUSPEND:
                3 WORDS OF DATA FROM CHILD'S POST OPERATION
                ERR-CHILD - CHILD DIED BEFORE POSTING */

BEGIN
     TASKNAME := ACTIVEECB^.PARAM1;
     ACTIVEECB^.PARAM2^.TASKID := NIL; /* INITIALLY NONE */
     ICBPTR := ICBQHD^.FIRST;
     WHILE ICBPTR <> NIL AND TASKNAME <> ICBPTR^.TASKNAME
        DO ICBPTR := ICBPTR^.ICBQLINK^.FLINK;
     IF ICBPTR = NIL
        THEN BEGIN
           STATUS := ERR-TNI;
            GO TO 5;
         END;
     /* IF DESIRED, CHECK HERE FOR MAXIMUM INVOCATIONS
        OF A GIVEN PROGRAM. */
     IF ICBPTR^.REMOVEECBPTR <> NIL
        THEN BEGIN
           STATUS := ERR-PERMIT;
           GO TO 5;
        END
        ELSE GETSYSBLOCKS(TCB, ECB, BCB, STATUS);
     IF STATUS <> NO-ERROR /* GOTTEN RID OF MANY ERRORS BY NOW */
        THEN GOTO 5; /* IF HERE, MOST RESOURCES AVAILABLE*/
     LINK(BCB,TCB^.BCBQHD,BCB^.BCBQLINK);
     FILL1TCB(ICBPTR,TCB); /* SET UP PART OF TCB & BCB */
     GETVARMEM(TCB,STATUS); /* GET LAST RESOURCE */
     IF STATUS = ERR-NMR
        THEN BEGIN /* COULDNT GET IT, SO */
           LINK(TCB,TCBAVAILQHD,TCB^.TCBQLINK);/* GIVE BACK */
```

```
         LINK(ECB,ECBAVAILQHD,ECB^.ECBQLINK); /* EVERYTHING */
         LINK(BCB,BCBAVAILQHD,BCB^.BCBQLINK); /* ELSE */
      END
      ELSE BEGIN /*GOT IT, SO FINISH NOW */
         FILL2TCB(TCB); /* FILL MOST OF TCB */
         TCB^.EFLAGPTR[SUSPEND] := ECB;
         ACTIVEECB^.PARAM2^.TASKID := TCB^.TASKID;
         FAMILY := ACTIVEECB^.PARAM2;
         IF FAMILY <> INDEPENDENT
            THEN BEGIN /* SET UP FAMILY LINKAGES */
               TCB^.POSTECBPTR := ACTIVEECB;
               FATHER := ACTIVEECB^.TCBPTR;
               TCB^.PARENT := FATHER;
               LINK(TCB,FATHER^.CHILDQHD,TCB^.BROTHERQLINK);
            END
            ELSE BEGIN /* SET UP INDEPENDENT */
               TCB^.POSTECBPTR := NIL;
               TCB^.PARENT := NIL;
               TCB^.BROTHERQLINK^.FLINK := NIL;
               TCB^.BROTHERQLINK^.BLINK := NIL;
            END;
         LINK(TCB,TCBREADYQHD,TCB^.TCBQLINK);
      END;
5:    ACTIVEECB^.STATUS^.ERROR := STATUS;
END.
```

```
PROCEDURE GETSYSBLOCKS(TCB,ECB,BCB,STATUS)

/*  INPUT:    NONE
    OUTPUT:   ADDRESSES OF A TCB, ECB, AND BCB, IFF
              ALL THREE ARE AVAILABLE, OR ELSE
              THREE "NIL"S -
              STATUS - EITHER NO-ERROR, IN WHICH
              CASE ALL THREE WERE ACQUIRED, OR
              ERR-NMR */

BEGIN
      IF TCBAVAILQHD^.FIRST <> NIL /* TCB AVAIL */
         AND ECBAVAILQHD^.FIRST <> NIL /* ECB AVAIL */
         AND BCBAVAILQHD^.FIRST <> NIL /* BCB AVAIL */
         THEN BEGIN
            TCB := TCBAVAILQHD^.FIRST;
            DELINK(TCB,TCBAVAILQHD,TCB^.TCBQLINK);
            BCB := BCBAVAILQHD^.FIRST;
            DELINK(BCB,BCBAVAILQHD,BCB^.BCBQLINK);
            ECB := ECBAVAILQHD^.FIRST;
            DELINK(ECB,ECBAVAILQHD,ECB^.ECBQLINK);
            STATUS := NO-ERROR;
         END
         ELSE STATUS := ERR-NMR;
END.


PROCEDURE FILL1TCB(ICB, TCB)

/*  INPUT:    ADDR OF AN ICB AND A BLANK TCB
    OUTPUT:   PART OF TCB IS FILLED IN - NOTE
              THAT NOTHING IS DONE THAT AFFECTS ANY
              OTHER CURRENTLY USED SYSTEM DATA
              STRUCTURE ENTRIES (I.E., TCB IS NOT
              LINKED TO ANY QUEUES), ALTHO THE
              BCB LINKED TO THE TCB IS ALSO PARTIALLY
              FILLED */

BEGIN
      TCB^.ICBPTR := ICB;
      BCB := TCB^.BCBQHD^.FIRST;
      BCB^.NUMBEROFMAPREGS := ICB^.MAPSFORVAR;
      BCB^.BEGINNINGMAPREG := ICB^.MAPSFORROOT +
         ICB^.MAPSFOROLAY + 1;
      BCB^.OWNERTCBPTR := TCB;
END.
```

```
PROCEDURE GETVARMEM(TCB,STATUS)

/*  INPUT:    A TCB, WITH A MOSTLY EMPTY BCB HOOKED ON
    OUTPUT:   IF AVAILABLE, REAL MEMORY IS ALLOCATED FOR
              THE VARIABLE SPACE OF THE TASK
              STATUS:
                  NO-ERROR
                  ERR-NMR   */

BEGIN
       BCB := TCB^.BCBQHD^.FIRST;
       IF BCB^.NUMBEROFMAPREGS > REMAININGPAGES
          THEN STATUS := ERR-NMR
          ELSE BEGIN
             FILLREGS(BCB^.NUMBEROFMAPREGS, BCB);
             /* THIS ROUTINE IS WITH THE MEMORY MGMT ROUTINES.
                IT PERFORMS THE ACTUAL ALLOCATION AND PLACES
                THE MAPREGISTER VALUES IN THE BCB. */
             STATUS := NO-ERROR;
          END;
END.

PROCEDURE FILL2TCB(TCB)

/*  INPUT:   PARTIALLY FILLED IN TCB
    OUTPUT:  TCB IS ALMOST COMPLETELY FILLED IN HERE.
             SINCE ALL RESOURCES HAVE BEEN ALLOCATED,
             IT IS NOW OKAY TO LINK THIS TCB TO OTHER
             SYSTEM QUEUES.  THE ASSOCIATED ICB AND BCB
             ARE ALSO UPDATED.  NOTE, HOWEVER, THAT
             FAMILY LINKAGES ARE NOT RPT NOT DONE HERE. */

BEGIN
       BCB := TCB^.BCBQHD^.FIRST;   /* DO BCB FIRST */
       WITH BCB DO BEGIN
          BUFFERID := 0;   /* FOR VARIABLE SPACE */
          USECOUNT := 1;
          IOINPROGRESS := 0;
          SHARE := FALSE; /* MOST OF THESE ARE NOT */
          RELEASEECBPOINTER := NIL; /* RELEVANT FOR THIS */
          OWNERBCBPOINTER := NIL; /* PARTICULAR BCB */
       END;  /* BCB NOW FILLED IN COMPLETELY */
       ICB := TCB^.ICBPTR; /* NOW DO ICB */
       LINK(TCB,ICB^.TCBQHD,TCB^.CLONEQLINK);
       ICB^.USECOUNT := ICB^.USECOUNT + 1;
       /* IF ANY LIMITATION ON THE NUMBER OF COPIES OF
          A PARTICULAR PROGRAM IS TO BE IMPOSED, IT WILL
          HAVE BEEN DONE BEFORE THIS.  SEE ROUTINE SKEDUL,
          WHERE VARIOUS ENTRIES IN THE ICB ARE CHECKED.  */
       /* ARE NOW DONE WITH THE ICB.  */
       WITH TCB DO BEGIN /* START THE TCB */
          EFLAGSACTUAL := 0;
          EFLAGSPOSSIBLE := 0;
          WAITFLAGS := 0;
          FOR IREG := 1 TO 15 DO
```

```
            EFLAGPTR[IREG] := NIL; /* #16 DONE IN SKEDUL */
          KILLINPROGRESS := FALSE;
          LASTBUFFERID := 0;
       END;
       FOR IREG := 1 TO CONTEXTSIZE DO
          TCB^.TASKCONTEXT[IREG] := ICB^.INITIALCONTEXT[IREG];
       ASSIGNID(TCB^.TASKID);
       FOR IREG := 1 TO ICB^.MAPSFORROOT DO
          TCB^.TASKMAP[IREG] := ICB^.TASKMAP[IREG];
       FOR IREG := 1 TO BCB^.NUMBEROFMAPREGS DO
          TCB^.TASKMAP[BCB^.BEGINNINGREG + IREG - 1]
          := BCB^.MAPREGS[IREG];
END.

PROCEDURE ASSIGNID(TASKID)

/*    INPUT:    NOTHING
      OUTPUT:   TASKID - TASK IDENTIFIER THAT IS UNIQUE
                WITHIN THIS PROCESSOR.  MOST LIKELY, JUST
                A POSITIVE INTEGER.  SOMETHING MORE COMPLI-
                CATED MAY BE USED, HOWEVER. */
BEGIN
       LASTID := LASTID + 1;
       TASKID := LASTID;
END.
```

6.2.2.2 POST

```
PROGRAM POST ( ACTIVEECB )
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              DATAWORDS - ADDR OF 3 WORDS OF DATA TO BE SENT
              BACK TO CALLING TASK'S PARENT - ADDR IN PARAM1
    OUTPUT:   STATUS - SUCCESS OR FAILURE CODE:
              NO-ERROR
              ERR-INDEP - TASK HAS NO PARENT
              ERR-AUU   - TASK HAS ALREADY POSTED ONCE */
BEGIN
   CHILDTCB := ACTIVEECB^.TCBPTR;
   PARENTTCB := CHILDTCB^.PARENT;
   IF PARENTTCB := NIL
      THEN STATUS := ERR-INDEP
      ELSE
         IF CHILDTCB^.POSTECB = NIL
            THEN STATUS := ERR-AUU
            ELSE BEGIN
               STATUS := NO-ERROR;
               CHILDTCB^.POSTECB := NIL;
               MOVE 3 WORDS FROM DATAWORDS ARRAY INTO POSTECB^.STATUS;
               EVENTCOMPLETE ( POSTECB );
            END;
   ACTIVEECB^.STATUS^.ERROR := STATUS;
END.
```

6.2.2.3 KILL

```
PROCEDURE KILL(ACTIVEECB)
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              TASKID - OF TASK TO KILL, IN PARAM 1.  IF TASK
                IS KILLING ITSELF, TASKID IS ZERO.  NOTE THAT ALL
                DESCENDANTS OF SPECIFIED TASK WILL ALSO BE KILLED.
              EVENTFLAG - BY WHICH TO NOTIFY THE CALLING TASK.
                NOTE THAT EVEN IF A TASK IS SUICIDING, A COMPLETION
                NOTIFICATION WILL STILL BE PERFORMED.  SEE ROUTINE
                'DISMEMBER' FOR DETAILS.
    OUTPUT:   STATUS - SUCCESS OR FAILURE CODE.  ONE OF:
                NO-ERROR
                ERR-EFLAG - EVENTFLAG ERROR
                ERR-NYC - TASK SPECIFIED IS NEITHER SELF NOR A FIRST-
                GENERATION DESCENDANT OF THE CALLER

      NOTE:  WHILE THE KILLING OF A TASK AND ALL ITS DESCENDANTS
             MAY BE PERFORMED ENTIRELY SYNCHRONOUSLY, THIS IS NOT
             ALWAYS POSSIBLE.  IN THIS CASE, THE PROCESS WILL OCCUR
             ASYNCHRONOUSLY, AND IN POSSIBLY SEVERAL SECTIONS.  AS
             A DESCENDANT TASK'S EVENTS ALL RUN DOWN TO COMPLETION,
             THAT DESCENDANT TASK WILL BE KILLED.  HOWEVER, SINCE
             KILLING A TASK WILL CAUSE US TO FAKE A POST TO ITS
             PARENT, IF ONE IS OUTSTANDING, THE PARENT MAY ALSO BE
             KILLED THEN, AND SO ON UP THE LINE.  */

BEGIN
      /* FIRST CHECK THE TASKID.  IF 0, CALLER IS KILLING
         HIMSELF.  IF NOT, HE HAD BEST BE KILLING A 1ST
         GENERATION CHILD. */
      ROOTID := ACTIVEECB^.PARAM1;
      IF ROOTID = 0 /* IS SUICIDE */
         THEN BEGIN
            ROOTTCB := ACTIVEECB^.TCBPTR; /* ROOTTCB IS FIRST DEATH */
            ACTIVEECB^.SUICIDE := TRUE;
            ACTIVEECB^.KILLCNT := 0;
            /* WE KEEP MUCH HOUSEKEEPING INFORMATION IN THE ACTIVEECB.
               THIS INCLUDES AN INDICATOR OF WHETHER OR NOT THIS IS A
               SUICIDE, HOW MANY TASKS TO KILL(EXCLUDING THE CALLER
               IFF A SUICIDE), AND OTHER INFO DESCRIBED BELOW. */
            GO TO 100; /* WHERE THE SLAUGHTER BEGINS IN EARNEST */
         END
         ELSE BEGIN /* CHECKING OF CALLER'S KIDS */
            TCB := ACTIVEECB^.TCBPTR;
            CTCB := TCB^.CHILDQHD^.FIRST; /* FIRST KID */
            IF CTCB = NIL
               THEN BEGIN /* HE HAS NO KIDS, AND IT'S NOT SUICIDE */
                  ACTIVEECB^.STATUS^.ERROR := ERR-NYC;
                  GO TO 200;
               END
               ELSE BEGIN /* NOW CHECK ALL THE KIDS */
```

```
                    WHILE CTCB <> NIL AND CTCB^.TASKID <> ROOTID
                    DO CTCB := CTCB^.BROTHERQLINK^.FLINK;
                    IF CTCB = NIL /* NONE MATCHED */
                       THEN BEGIN
                          ACTIVEECB^.STATUS^.ERROR := ERR-NYC;
                          GO TO 200;
                       END
                       ELSE BEGIN /* GOT THE LITTLE BEGGAR */
                          ROOTTCB := CTCB;
                          ACTIVEECB^.SUICIDE := FALSE;
                          ACTIVEECB^.KILLCNT := 1;
                          GO TO 100; /* LET THE GAMES BEGIN */
                       END; /* IF CTCB = NIL */
                 END; /* ALSO IF CTCB = NIL */
            END; /* IF ROOTID = 0 */
/*     WE HAVE NOW EITHER CAUGHT AN ERR-NYC AND
.      WHISKED OFF TO 200, OR ELSE IN 'ROOTTCB' WE HAVE THE
       TCB OF THE FIRST TASK TO KILL.  WE MUST ALSO KILL ALL
       HIS DESCENDANTS, EVEN UNTO THE THOUSANDTH GENERATION. */
100:   ACTIVEECB^.ORIGINALCALL := TRUE;
       ROOTKILL(ROOTTCB, ACTIVEECB);
       /* WHEN WE GET BACK, IF EVERYTHING HAS BEEN KILLED(FOR A NON-
          SUICIDE) OR EVERYTHING BUT THE CALLER(FOR A SUICIDE), SET
          THE STATUS TO NO-ERROR AND LET THE OS DO THE CALLCOMPLETION,
          WHICH WILL FAKE INTO AN EVENTCOMPLETION.  IF THE KILLING
          ISN'T FINISHED, SET ORIGINALCALL TO FALSE, SO WHEN IT DOES
          FINISH(ASYNCHRONOUSLY), WE WILL DO THE EVENTCOMPLETION.  */
       IF ACTIVEECB^.KILLCOUNT = 0
          THEN ACTIVEECB^.STATUS^.ERROR := NO-ERROR
          ELSE ACTIVEECB^.ORIGINALCALL := FALSE;
200:
END.
```

```
PROCEDURE ROOTKILL(TCB,KILLECB)
/*  INPUT:    TCB - TASK WHICH IS TO BE KILLED, ALONG
                WITH ALL DESCENDANTS
              KILLECB - THE ECB FROM THE KILL CALL.  WE WILL
                KEEP TRACK OF SEVERAL THINGS IN HERE, INCLUDING
                WHETHER OR NOT THIS IS A SUICIDE KILL, AND THE
                TOTAL NUMBER OF TASKS WHOSE KILL CANNOT BE COM-
                PLETED IMMEDIATELY.
      OUTPUT:   NONE */

BEGIN
      /* MOST OF THE KILLECB HAS BEEN SET UP ALREADY.  IN 'KILL',
         WE SET UP THE SUICIDE FLAG AND THE ORIGINAL VALUE FOR
         KILLCOUNT(0 IF SUICIDE, 1 IF NOT).  THUS, WHEN WE DISMEMBER
         THE LAST TASK(KILLCOUNT GOES BACK TO ZERO), WE CAN SIGNAL
         COMPLETION TO THE CALLER. */
      TCB^.KILLECBPTR := KILLECB;
      /* NOW WHAT WE DO IS KILL ALL LOWER TASKS FIRST, AS MUCH
         AS IS POSSIBLE */
      CTCB := TCB^.CHILDQHD^.FIRST;
      IF CTCB <> NIL /* THEN THERE ARE KIDS */
         THEN BEGIN
            DESCKILL(CTCB,KILLECB);
            FAKEWAIT(CTCB); /* THIS COMPLETES THE KILL OF THIS
            FIRST CHILD */
         END;
      FAKEWAIT(TCB); /* THIS ROUTINE SETS HIM UP AS WAITING
      FOR ALL INCOMPLETE EVENTS, OR, IF THERE ARE NONE,
      HAS THE ACTUAL KILLING DONE */
END.
```

```
PROCEDURE DESCKILL(TCB,KILLECB)
/*  INPUT:    TCB - OF A TASK, ALL OF WHOSE DESCENDANTS
                AND SIBLINGS AND THEIR DESCENDANTS ARE TO BE KILLED
              KILLECB - THE ECB FROM THE ORIGINAL KILL CALL, WHERE
                WE DO SOME HOUSEKEEPING
    OUTPUT.   NONE */

BEGIN
      TCB^.KILLECBPTR := KILLECB;
      KILLECB^.KILLCOUNT := KILLECB^.KILLCOUNT + 1;
      /* NOW WORK OUR WAY DOWN THE TREE AS FAR AS POSSIBLE */
      CTCB := TCB^.CHILDQHD^.FIRST;
      IF CTCB <> NIL
         THEN DESCKILL(CTCB); /* RECURSIVE DESCENT */
      /* ARE NOW EITHER DOWN AS FAR AS POSSIBLE, OR HAVE
         DONE EVERYONE BELOW THIS TASK AND ARE ON THE WAY
         BACK UP.  SO, FIRST SAVE THIS TASK'S BROTHER LINK,
         IF ANY, AND THEN KILL HIM */
      BTCB := TCB^.BROTHERQLINK^.FLINK;
      /* NOW TAKE CARE OF THIS GUY */
      FAKEWAIT(TCB);
      /* NOW, IF THERE WAS A SIB, GET HIM */
      IF BTCB <> NIL /* GO ACROSS TREE */
         THEN DESCKILL(BTCB); /* DESCEND AGAIN */
END.
```

```
PROCEDURE FAKEWAIT(TCB)
/*  INPUT:    TCB - OF A SINGLE TASK TO SET UP FOR
                 RUNDOWN OR TO DISMEMBER
    OUTPUT:   NONE */

BEGIN
      /* FIRST TAKE THIS GUY OFF THE READY-Q, IF ON IT */
      IF TCB^.WAITTYPE = NIL /* HE'S ON IT */
         THEN DELINK(TCB,RDYQHD,TCB^.TCBQLINK);  /* NOW HE'S NOWHERE */
      /* NOW CHECK FOR INCOMPLETE EVENTS */
      IF TCB^.EFLAGSPOSSIBLE <> 0 /* HE HAS SOME */
         THEN BEGIN /* SETTING UP FOR RUNDOWN */
         WITH TCB DO
         BEGIN
            WAITFLAGS := EFLAGSPOSSIBLE;  /* WAIT FOR ALL */
            IF WAITTYPE = NIL /* HE'S ON NOTHING */
               THEN LINK(TCB,SUSPENDQHD,TCBQLINK);  /* PUT ON SUSPENDQ */
            WAITTYPE = WAITALL;  /* WAIT FOR ALL */
         END;  /* WITH */
         END /* SETTING UP FOR RUNDOWN */
         ELSE BEGIN  /* HE HAS NO INCOMPLETE EVENTS.  HOWEVER,
            WE CAN'T JUST DISMEMBER HIM OUT OF HAND, SINCE HE
            MAY 'OWN' A BUFFER FOR WHICH A DESCENDANT HAS AN
            INCOMPLETE I/O TRANSFER.  IF SO, JUST MAKE SURE HE
            GOES TO THE SUSPEND-Q.  OTHERWISE, THOUGH, WE CAN
            DISMEMBER HIM. */
            BCBPTR := TCB^.BCBQHD^.FIRST;
            RELEASABLE := TRUE;
            WHILE BCBPTR <> NIL DO
               BEGIN
                  IF BCBPTR^.IOINPROGRESS <> 0
                     THEN RELEASABLE := FALSE;
                     /* NOTE ONLY HAVE TO CHECK IOINPROGRESS COUNT,
                        SINCE THIS CAN BE NON-ZERO ONLY FOR THE CASE
                        WE WANT */
                     BCBPTR := BCBPTR^.BCBQLINK^.FLINK;
               END;  /* WHILE */
            IF RELEASABLE
               THEN DISMEMBER(TCB) /* DO IT IF WE CAN */
               ELSE BEGIN /* MAKE SURE HE IS ON SUSPENDQ */
                  IF WAITTYPE = NIL /* HE WASN'T */
                     THEN LINK(TCB,SUSPENDQHD,TCB^.TCBQLINK);
                  WAITTYPE := WAITALL;
               END;  /* IF RELEASABLE */
         END;  /* DISMEMBERING */
END.
```

```
PROCEDURE DISMEMBER(TCB)
/*   INPUT:     TCB - OF A KILLED TASK, ALL OF WHOSE
                   EVENTS HAVE COMPLETED
     OUTPUT:    NONE.  ALL RESOURCES(BCB'S AND BUFFERS,
                   TCB, AND ECB'S) ARE RETURNED.  ICB IS ADJUSTED.
                   IF ICB HAD AN OUTSTANDING REMOVE CALL THAT CAN
                   NOW BE COMPLETED, THIS IS DONE.  FINALLY, IF
                   THIS IS THE LAST TASK TO BE DISMEMBERED UNDER
                   THIS KILL CALL, THE KILLECB IS PASSED TO EVENTCOMPLETE*/

BEGIN
      /* DO ICB FIRST */
      ICB := TCB^.ICBPTR;
      ICB^.USECOUNT := ICB^.USECOUNT - 1;
      DELINK(TCB,ICB^.TCBQHD,TCB^.CLONEQLINK);
      /* NOW CHECK FOR INCOMPLETE REMOVE */
      IF ICB^.REMOVEECBPTR <> NIL /* THERE IS ONE */
      AND ICB^.USECOUNT = 0 /* AND WE CAN DO IT */
         THEN BEGIN
            REMECB := ICB^.REMOVEECBPTR; /* GET THE ECB */
            RETURNALLRESOURCES(ICB); /* COMPLETE THE REMOVE */
            REMECB^.STATUS^.ERROR := NO-ERROR;
            EVENTCOMPLETE(REMECB); /* AND SEND THE COMPLETION */
         END;
      /* NOW DO THE TASK */
      /* FIRST SEE IF THERE ARE ANY ECB'S TO RETURN.  THERE CAN
         STILL BE SOME FROM EVENTS THAT COMPLETED FOR WHICH HE DIDN'T
         DO A 'GETSTATUS'. */
      FOR IFLAG := 1 TO 16 DO
         IF TCB^.EFLAGPTR[IFLAG] <> NIL /* GOT ONE */
            THEN BEGIN /* PUT ON FREEQ */
               ECB := TCB^.EFLAGPTR[IFLAG];
               LINK(ECB,ECBAVAILQHD,ECB^.ECBQLINK);
            END;
      /* NOW SEE IF HIS PARENT IS WAITING FOR HIM */
      IF TCB^.POSTECB <> NIL /* YUP, HE IS */
         THEN BEGIN
            PECB := TCB^.POSTECB;
            PECB^.STATUS^.ERROR := ERR-CHILD;
            /* THIS COVERS THE CASE OF A PARENT WAITING FOR A
               CHILD WHO SUICIDED, OR OF A PARENT WHO KILLS HIS
               OWN KID */
            EVENTCOMPLETE(PECB); /* TELL PARENT */
         END;
      /* NOW SEE ABOUT THE KILLING TASK:  DOES IT GET A
         COMPLETION NOW     */
      KECB := TCB^.KILLECBPTR;
      KECB^.KILLCOUNT := KECB^.KILLCOUNT - 1;
      /* NOW WE HAVE DECREMENTED THE COUNT.  IF NONE IS LEFT
         AND IT IS NOT THE ORIGINAL CALL, WE NOTIFY THE CALLER.
         IF IT WAS NOT A SUICIDE, ALL IS WELL.  IF IT WAS, NOW THE
         SUICIDER GETS HIS LAST EVENT.  THUS, IN A SUICIDE, THE CALLER
         WILL ALWAYS BE THE LAST ONE KILLED. */
      IF KECB^.KILLCOUNT = 0
      AND KECB^.ORIGINALCALL = FALSE
```

```
            THEN BEGIN
               KECB^.STATUS^.ERROR := NO-ERROR;
               EVENTCOMPLETE(KECB);  /* NOTIFY THE KILLER */
            END;
         /* AND FINALLY, RETURN THIS GUY'S RESOURCES */
         RETURNALLTASKRESOURCES(TCB);
END.
```

```
PROCEDURE RETURNALLTASKRESOURCES(TCB)
/*  INPUT:     TCB - OF A TASK WHOSE BCBS,
                 BUFFERS, AND TCB CAN BE RETURNED TO FREE-Q'S
    OUTPUT:    NONE */

BEGIN
      BCBPTR := TCB^.BCBQHD^.FIRST;
      IF BCBPTR <> NIL /* THERE ARE BCB'S TO DO */
         THEN BEGIN
            WHILE BCBPTR <> NIL DO BEGIN
               BCBNXT := BCBPTR^.BCBQLINK^.FLINK;
               /* KEEP TRACK OF THE NEXT ONE */
               IF BCBPTR^.OWNER = TRUE
                  THEN RETURNREALMEMORY(BCBTYPE,BCBPTR)
               /* CAN ONLY RETURN THE MEMORY IF HE OWNED IT */
                  ELSE BEGIN /* SEE IF GETTING RID OF THIS GUY MEANS
                     THAT THE OWNER OF THE MEMORY CAN ALSO GO */
                     OBCB := BCBPTR^.OWNERBCBPTR;
                     IF OBCB^.IOINPROGRESS = 0
                        THEN FAKEWAIT(OBCB^.OWNERTCBPOINTER);
                        /* NOTE THAT THIS FAKEWAIT MAY BE REDUNDANT,
                           BUT IT WILL NOT CAUSE ANY REAL GRIEF */
                  END; /* IF OWNER */
               LINK(BCBPTR,BCBAVAILQHD,BCBPTR^.BCBQLINK);
               /* CAN ALWAYS RETURN THE BCB, THO */
            END; /* WHILE */
         END; /* IF */
      LINK(TCB,TCBAVAILQHD,TCB^.TCBQLINK); /* FREE TCB */
END.
```

6.3 MEMORY MANAGEMENT

6.3.1 OVERLAY CONTROL

6.3.1.1 LOADOVERLAY

```
PROCEDURE LOADOVERLAY ( ACTIVEECB )
/*     INPUT:     ACTIVEECB, WHICH SPECIFIES:
                  OVERLAYID - ID OF AN OVERLAY TO WHICH CALLING TASK
                  WILL BE MAPPED - IN PARAM1
       OUTPUT:    CALLING TASK IS MAPPED TO OVERLAY IF LEGAL
                  STATUS - SUCCESS OR ERROR CODE:
                  NO-ERROR
                  ERR-OLAYID - OVERLAY NOT ASSIGNED TO THIS TASK */
BEGIN
       TCB := ACTIVEECB^.TCBPTR;
       ICB := TCB^.ICBPTR;
       FLINK := ICB^.OCBQHD^.FIRST;
       OVERLAYID := ACTIVEECB^.PARAM1;
       WHILE FLINK <> NIL AND FLINK^.OVERLAYID <> OVERLAYID
          DO FLINK := FLINK^.OCBQLINK^.FLINK;
       IF FLINK <> NIL
          THEN BEGIN
             STATUS := NO-ERROR;
             FOR IREG := 1 TO FLINK^.NUMBEROFMAPREGS DO
                TCB^.TASKMAP[FLINK^.BEGINNINGREG+IREG-1] :=
                FLINK^.MAPREGS[IREG];
          END
          ELSE STATUS := ERR-OLAYID;
       ACTIVEECB^.STATUS^.ERROR := STATUS;
END.
```

6.3.1.2 RELEASEOVERLAY

```
PROCEDURE  RELEASEOVERLAY ( ACTIVEECB )
/*     INPUT:    ACTIVEECB, WHICH SPECIFIES:
                 OVERLAYID - ID OF AN OVERLAY WHICH CALLING TASK
                 IS TO BE MAPPED OUT OF - IS PARAM1
       OUTPUT:   CALLING TASK IS MAPPED OUT OF OVERLAY IF LEGAL -
                 STATUS - SUCCESS OR ERROR CODE:
                 NO-ERROR
                 ERR-OLAYID - OVERLAY NOT ASSIGNED TO THIS TASK
                 ERR-MAP    - TASK NOT MAPPED INTO THIS OVERLAY */
BEGIN
       TCB := ACTIVEECB^.TCBPTR;
       ICB := TCB^.ICBPTR;
       FLINK := ICB^.OCBQHD^.FIRST;
       OVERLAYID := ACTIVEECB^.PARAM1;
       WHILE FLINK <> NIL AND FLINK^.OVERLAYID <> OVERLAYID DO
          FLINK := FLINK^.OCBQLINK^.FLINK;
       IF FLINK = NIL
          THEN STATUS := ERR-OLAYID
          ELSE BEGIN
             UNMAPOLAY ( TCB, FLINK, RETURNSTATUS);
             STATUS := RETURNSTATUS;
          END;
       ACTIVEECB^.STATUS^.ERROR := STATUS;
END.

PROCEDURE UNMAPOLAY ( TCB, FLINK, RETURNSTAT )
/*     INPUT:    ADDR OF A TCB AND AN OCB WICH IS TO BE MAPPED
                  OUT OF THIS TASK - TASK NEED NOT BE CURRENTLY
                  MAPPED IN
       OUTPUT:    IF MAPPED IN AT CALL, TASK IS MAPPED OUT AT RETURN
                  RETURNSTAT - EITHER NO-ERROR OR ERR-MAP */
BEGIN
       RETURNSTAT := NO-ERROR;
       FOR IREG := 1 TO FLINK^.NUMBEROFMAPREGS DO
          IF TCB^.TASKMAP[FLINK^.BEGINNINGREG+IREG-1] =
             FLINK^.MAPREGS[IREG]
             THEN
                TCB^.TASKMAP[FLINK^.BEGINNINGREG+IREG-1]
                   := ILLEGAL-MAPREGISTER-VALUE
             ELSE RETURNSTAT := ERR-MAP;
END.
```

6.3.2 BUFFER CONTROL

6.3.2.1 GETBUFFER

```
PROCEDURE GETBUFFER ( ACTIVEECB )

/*       INPUT:    ACTIVEECB, WHICH SPECIFIES:
                   LOGADDR - LOGICAL ADDRESS IN USER'S ADDRESS SPACE
                   AT WHICH BUFFER SHALL START - IN PARAM1
                   NWORDS  - NUMBER OF WORDS IN BUFFER - IN PARAM2
                   SHAREMODE - EITHER "PRIVATE" OR "SHARABLE" - IN PARAM3
         OUTPUT:   BUFFERID - ID OF ASSIGNED BUFFER, IF AVAILABLE - ADDR
                   IS IN PARAM4
                   STATUS - EITHER NO-ERROR, OR ERROR CODE:
                   ERR-NWORDS:   ILLEGAL NUMBER OF WORDS SPECIFIED
                   ERR-NMR:      MEMORY NOT AVAILABLE
                   ERR-ADDR:     LOGADDR NOT ALIGNED ON MAP REGISTER
                                 BOUNDARY OR NOT IN BUFFER AREA*/

BEGIN
TCB:= ACTIVEECB^.TCBPTR;
ICB:= TCB^.ICBPTR;
NWORDS := ACTIVEECB^.PARAM2;
NMAPREGS := ENTIER ( NWORDS / PAGESIZE );
IF ICB^.MAPSFORBUFF < NMAPREGS OR NWORDS < 0
       THEN STATUS := ERR-NWORDS
       ELSE BEGIN
         IF NMAPREGS > REMAININGPAGES
            THEN STATUS := ERR-NMR
            ELSE BEGIN
               LOGADDR := ACTIVEECB^.PARAM1;
               IF LOGADDR NOT ALIGNED TO MAP GRANULARITY
               OR LOGADDR NOT IN TASK'S BUFFER SPACE
                  THEN STATUS := ERR-ADDR
                  ELSE BEGIN
                     BCB := BCBAVAILQHD^.FIRST;
                     IF BCB = NIL
                        THEN STATUS := ERR-NMR
                        ELSE BEGIN
                           DELINK(BCB,BCBAVAILQHD,BCB^.BCBQLINK);
                           TCB^.LATESTBUFFERID := TCB^.LATESTBUFFERID
                              + 1;
                           BCB^.BUFFERID := TCB^.LATESTBUFFERID;
                           ACTIVEECB^.PARAM4^.BUFFERID := BCB^.BUFFERID;
                           STATUS := NO-ERROR;
                           BCB^.NUMBEROFMAPREGS :=
                              NMAPREGS;
                           BCB^.BEGINNINGREG := LOGADDR/PAGESIZE;
                           BCB^.USECOUNT := 1;
                           BCB^.OWNERTCBPTR := TCB;
                           BCB^.OWNERBCBPTR := NIL;
                           BCB^.RELEASEECBPTR := NIL;
                           BCB^.IOINPROGRESS := 0;
                           SHAREMODE := ACTIVEECB^.PARAM3;
```

```
                              IF SHAREMODE = PRIVATE
                                 THEN BCB^.SHARE := FALSE
                                 ELSE BCB^.SHARE := TRUE;
                              FILLREGS(NUMBEROFMAPREGS,BCB);
                              LINK (BCB,TCB^.BCBQHD,BCB^.BCBQLINK);
                           END;
                      END;
                 END;
            END;
            ACTIVEECB^.STATUS^.ERROR := STATUS;
       END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE FILLREGS ( NMAPREGS, BCB )

      /*INPUT:    A BUFFER CONTROL BLOCK WHICH IS TO BE ALLOCATED
                 "NMAPREGS" PAGES OF MEMORY AND TO HAVE THE
                 MAP REGISTER PORTION FILLED IN.
        OUTPUT:  FILLED IN BUFFER CONTROL BLOCK */

BEGIN
      REMAININGPAGES := REMAININGPAGES - NMAPREGS;
      IPAGE := FIRST-AVAILABLE-MEMORY-PAGE-INDEX;
      NEXTREG := 0;
      UNTIL NEXTREG = NMAPREGS DO
         BEGIN
            IF PHYSPAGETABLE[IPAGE] = 0
               THEN BEGIN
                  NEXTREG := NEXTREG + 1;
                  PHYSPAGETABLE[IPAGE] := BCB^.OWNERTCBPTR;
                  BCB^.MAPREGS[NEXTREG] := MAPVALUE ASSOCIATED WITH
                  THIS PHYSICAL PAGE;
                  END
               ELSE IPAGE := IPAGE + 1;
         END;
END.
```

6.3.?.2 EQUATEBUFFER

```
PROCEDURE EQUATEBUFFER( ACTIVEECB )

/*      INPUT:   ACTIVEECB, WHICH SPECIFIES:
                 PARENTTASKID - OF AN ANCESTOR TASK - IN PARAM2
                 PARENTBUFFERID - ID OF A BUFFER ASSIGNED TO THAT
                 TASK - IN PARAM1
        OUTPUT:  BUFFERID - ID UNDER WHICH CALLING TASK MAY MAP TO
                 THAT BUFFER, IF ALLOWED - ADDR IS IN PARAM3
                 STATUS - SUCCESS OR ERROR CODE:
                 NO-ERROR
                 ERR-PARENT    TASK SPECIFIED BY PARENTTASKID IS NOT AN
                               ANCESTOR OF CALLER
                 ERR-BUFFID    BUFFER SPECIFIED BY PARENTBUFFERID IS NOT
                               OWNED BY OR ASSIGNED TO THE ANCESTOR
                 ERR-PERMIT    BUFFER SPECIFIED IS EITHER PRIVATE OR
                               FLAGGED FOR RELEASE
                 ERR-NMR       SYSTEM RESOURCES NOT AVAILABLE */

BEGIN
    TCB := ACTIVEECB^.TCBPTR;
    PARENT := TCB^.PARENTPTR;
    PARENTTASKID := ACTIVEECB^.PARAM2;
    WHILE PARENT <> NIL AND PARENTTASKID <> PARENT^.TASKID
       DO PARENT := PARENT^.PARENTPTR;
    IF PARENT = NIL
       THEN STATUS := ERR-PARENT
       ELSE BEGIN
          FLINK := PARENT^.BCBQHD^.FIRST;
          PARENTBUFFERID := ACTIVEECB^.PARAM1;
          WHILE FLINK <> NIL AND FLINK^.BUFFERID <> PARENTBUFFERID
             DO FLINK := FLINK^.BCBQLINK^.FLINK;
          IF FLINK = NIL OR FLINK^.OWNERBCBPTR <> NIL
             THEN STATUS := ERR-BUFFID
             ELSE BEGIN
                IF FLINK^.SHARE = FALSE
                   OR FLINK^.RELEASEECBPTR <> NIL
                   THEN STATUS := ERR-PERMIT
                   ELSE BEGIN
                      BCB := BCBAVAILQHD^.FIRST;
                      IF BCB = NIL
                         THEN STATUS := ERR-NMR
                         ELSE BEGIN
                            DELINK(BCB,BCBAVAILQHD,BCB^.BCBQLINK);
                            TCB^.LATESTBUFFERID := TCB^.LATESTBUFFERID
                               + 1;
                            BCB^.BUFFERID := TCB^.LATESTBUFFERID;
                            ACTIVEECB^.PARAM3^.MYBUFFERID :=
                               BCB^.BUFFERID;
                            BCB^.RELEASEECBPTR := NIL;
                            FLINK^.USECOUNT := FLINK^.USECOUNT + 1;
                            STATUS := NO-ERROR;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
                    BCB^.OWNERBCBPTR := FLINK;
                    BCB^.OWNERTCBPTR :=
                       FLINK^.OWNERTCBPTR;
                    FOR IREG := 1 TO
                       FLINK^.NUMBEROFMAPREGS
                    DO BCB^.MAPREGS[IREG] :=
                       FLINK^.MAPREGS[IREG];
                    BCB^.NUMBEROFMAPREGS:=
                       FLINK^.NUMBEROFMAPREGS;
                    BCB^.BEGINNINGREG :=
                       FLINK^.BEGINNINGREG;
                    LINK (BCB, TCB^.BCBQHD, BCB^.BCBQLINK);
                 END;
              END;
           END;
        END;
    ACTIVEECB^.STATUS^.ERROR := STATUS;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.3.2.3 MAPTOBUFFER

```
PROCEDURE MAPTOBUFFER ( ACTIVEECB )

/*    INPUT:     ACTIVEECB, WHICH SPECIFIES:
                 BUFFERID - ID OF BUFFER OBTAINED BY GETBUFFER OR
                 EQUATEBUFFER CALL - IN PARAM1
      OUTPUT:    USER IS MAPPED TO SPECIFIED BUFFER, IF LEGAL
                 STATUS - SUCCESS OR ERROR CODE:
                 NO-ERROR
                 ERR-BUFFID -- SPECIFIED BUFFER NOT ASSIGNED TO USER
                 ERR-PERMIT -- BUFFER IS FLAGGED FOR RELEASE */

BEGIN
      TCB := ACTIVEECB^.TCBPTR;
      FLINK := TCB^.BCBQHD^.FIRST;
      BUFFERID := ACTIVEECB^.PARAM1;
      WHILE FLINK <> NIL AND FLINK^.BUFFERID <> BUFFERID DO
            FLINK := FLINK^.BCBQLINK^.FLINK;
            IF FLINK = NIL
               THEN STATUS := ERR-BUFFID
               ELSE
                  IF FLINK^.RELEASEECBPTR <> NIL
                     THEN STATUS := ERR-PERMIT
                     ELSE BEGIN
                        STATUS := NO-ERROR;
                        FOR IREG := 1 TO FLINK^.NUMBEROFMAPREGS DO
                        TCB^.TASKMAP[FLINK^.BEGINNINGREG + IREG - 1]
                           := FLINK^.MAPREGS[IREG];
                     END;
      ACTIVEECB^.STATUS^.ERROR := STATUS;
END.
```

6.3.2.4 RELEASEBUFFER

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE RELEASEBUFFER ( ACTIVEECB )
/*  INPUT:    ACTIVEECB, WHICH SPECIFIES:
              BUFFERID - OF A BUFFER TO BE RELEASED - IN PARAM1
    OUTPUT:   BUFFER IS RELEASED IF POSSIBLE, FLAGGED FOR RELEASE
              IF STILL IN USE
              STATUS - SUCCESS OR ERROR CODE:
              NO-ERROR
              ERR-BUFFID - THIS BUFFERID NOT ASSIGNED TO CALLER
              ERR-EFLAG  - ILLEGAL EVENTFLAG SPECIFIED
              ERR-IOIP   - I/O STILL IN PROGRESS IN BUFFER */

BEGIN
      STATUS := NO-ERROR;
      TCB := ACTIVEECB^.TCBPTR;
      FLINK := TCB^.BCBQHD^.FIRST;
      BUFFERID := ACTIVEECB^.PARAM1;
      WHILE FLINK <> NIL AND FLINK^.BUFFERID <> BUFFERID DO
         FLINK := FLINK^.BCBQLINK^.FLINK;
      IF FLINK = NIL OR FLINK^.BUFFERID = 0
         THEN STATUS := ERR-BUFFID
         ELSE BEGIN
            IF FLINK^.OWNERBCBPTR = NIL /* THIS TASK IS OWNER */
               THEN BEGIN
                  FLINK^.USECOUNT := FLINK^.USECOUNT - 1;
                  IF FLINK^.USECOUNT <> 0
                     THEN BEGIN /* FLAG FOR RELEASE AND UNMAP */
                        FLINK^.RELEASEECBPTR := ACTIVEECB;
                        UNMAPBUFFER ( FLINK, TCB );
                        STATUS := REQUEST-IN-PROGRESS;
                     END
                     ELSE BEGIN /* USECOUNT NOW ZERO */
                        IF FLINK^.IOINPROGRESS <> 0
                           THEN BEGIN /* CAN'T RETURN YET */
                              STATUS := ERR-IOIP;
                              FLINK^.USECOUNT := 1;
                           END
                           ELSE BEGIN /* NO I/O, RETURN BUFFER AND BCB*/
                              UNMAPBUFFER ( FLINK, TCB );
                              RETURNREALMEMORY ( FLINK );
                              DELINK(FLINK,TCB^.BCBQHD,FLINK^.BCBQLINK);
                              LINK(FLINK,BCBAVAILQHD,FLINK^.BCBQLINK);
                           END;
                     END;
               END  /*OWNER = TRUE */
               ELSE BEGIN /* RELEASING FROM OWNER'S DESCENDANT*/
                  IF FLINK^.IOINPROGRESS <> 0
                     THEN STATUS := ERR-IOIP
                     ELSE BEGIN
                     UNMAPBUFFER ( FLINK, TCB );
                     OWNERTCB := FLINK^.OWNERTCBPTR;
                     OWNERBCB := FLINK^.OWNERBCBPTR;
```

```
                    DELINK(FLINK, TCB^. BCBQHD, FLINK^. BCBQLINK);
                    LINK(FLINK, BCBAVAILQHD, FLINK^. BCBQLINK);
                    OWNERBCB^. USECOUNT := OWNERBCB^. USECOUNT - 1;
                    IF OWNERBCB^. USECOUNT = 0 AND
                       OWNERBCB^. IOINPROGRESS = 0 AND
                       OWNERBCB^. RELEASEECBPTR <> NIL
                       THEN BEGIN
                          RETURNREALMEMORY(BCBTYPE, OWNERBCB );
                          DELINK(OWNERBCB, OWNERTCB^. BCBQHD,
                                 OWNERBCB^. BCBQLINK);
                          LINK(OWNERBCB, BCBAVAILQHD,
                                 OWNERBCB^. BCBQLINK);
                          /* ISSUE RELEASE COMPLETION TO OWNER */;
                          RELEASEECBPTR^. STATUS^. ERROR := NO-ERROR;
                          EVENTCOMPLETE ( RELEASEECBPTR );
                       END;
                    END;
                 END;
            END;
      ACTIVEECB^. STATUS^. ERROR := STATUS;
END.
```

UNMAPBUFFER

```
PROCEDURE UNMAPBUFFER ( BCB, TCB )
/*  INPUT:    ADDRESS OF A BUFFER CONTROL BLOCK FROM WHICH A SPECIFIED
              TASK IS TO BE UNMAPPED
    OUTPUT:   TASK IS UNMAPPED */

BEGIN
      FOR IREG := 1 TO BCB^. NUMBEROFMAPREGS DO
         IF TCB^. TASKMAP[BCB^. BEGINNINGREG + IREG - 1] =
            BCB^. MAPREGS[IREG]
            THEN TCB^. TASKMAP[BCB^. BEGINNINGREG + IREG - 1]
               := ILLEGAL-MAPREGISTER-VALUE;
END.
```

RETURNREALMEMORY

```
PROCEDURE RETURNREALMEMORY ( NODETYPE, NODE )
/*  INPUT:    ADDRESS OF A  CONTROL BLOCK WHICH SPECIFIES,
              THROUGH THE CONTENTS OF ITS MAP REGISTERS, PAGES OF
              REAL MEMORY WHICH ARE TO BE RETURNED
    OUTPUT:   MEMORY IS RETURNED TO FREE POOL */

BEGIN
      CASE NODETYPE OF
BCB, OCB, ICB, TCB:
      IF NODE^. NUMBEROFMAPREGS > 0 THEN
      FOR IREG := 1 TO NODE^. NUMBEROFMAPREGS DO BEGIN
```

```
            CONVERT NODE^.MAPREGS[IREG] FROM MAP VALUE TO PAGE NUMBER;
            PHYSPAGETABLE[PAGENUMBER] := 0;
        END;
        REMAININGPAGES := REMAININGPAGES + NODE^.NUMBEROFMAPREGS;
    END.
```

6.4 INTER-TASK COMMUNICATIONS

6.4.1 ROUTER FUNCTIONS

6.4.1.1 SENDMESSAGE

```
PROCEDURE SENDMESSAGE (ECBPTR);

/*THE PARAMETERS STORED IN THE ECB ARE:
   EVENTFLAG:   EFLAG1..EFLAG15, SUSPEND
   TASKNAME:    NAME OF THE RECEIVING TASK IN THE
                   GLOBAL TABLE
   MBUFFSPEC:   RECORD
                BUFFERID: INTEGER;
                ADDRESS: INTEGER;
                SIZE: INTEGER
                END;
   ABUFFSPEC:   SAME AS MBUFFSPEC
   TIMEOUT:        THE MAXIMUM TIME THIS CONVERSATION CAN
                   TAKE.
   MESSAGEID:      ASSIGNED WHEN THE CONVERSATION IS INITIATED.*/

BEGIN
ECBPTR^.MESSAGEID.MESSAGENUMBER := MESSAGENUMBER;
MESSAGENUMBER := MESSAGENUMBER + 1;
ECBPTR^.MESSAGEID.PROCESSORID := THISPROCESSORID;
IF     /*CHECK FOR DESTINATION IN THIS PROCESSOR*/
   GLOBALTABLE [CONFIG, ECBPTR^.TASKNAME] < 0
THEN      /*CAN DO SENDMESSAGE SYNCHRONOUSLY*/
   MESSAGERECEIVED (ECBPTR)
ELSE      /*THE DISPATCHER GETS TO SEND THE MESSAGE*/
   BEGIN
   /*INCREMENT IOINPROGRESS FLAG IN USER BCB'S AND, IF
     NOT OWNED BY THIS USER, IN OWNER'S BCB'S*/
   LINK (ECBPTR, MESSAGEOUTQHEAD, ECBPTR^.ECBQLINK);
   EVENTCOMPLETE (MESSAGEOUTQCB)
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY ... TO DDC

```
PROCEDURE MESSAGERECEIVED (ECBPTR);

/*PLACE THE MESSAGE SPECIFIER ON THE MESSAGE QUEUE OF
  THE RECEIVING TASK, AND INDICATES COMPLETION OF THE
  EVENT ASSOCIATED WITH THE MESSAGE QUEUE*/

BEGIN
MESSAGEQCB := MESSAGETABLE [TASKNAME];
LINK (ECBPTR, MESSAGEQHEAD, ECBPTR^.ECBQLINK);
MESSAGEQCB^.MESSAGEID :=
   ECBPTR^.MESSAGEID;
MESSAGEQCB^.MSLENGTHPTR := ECBPTR^.MSLENGTH;
MESSAGEQCB^.ANSLENGTHPTR := ECBPTR^.ANSLENGTH;
/*DECREMENT IOINPROGRESS FOR USER MESSAGE BCB AND, IF
  NOT OWNED BY THIS USER, IN OWNER'S MESSAGE BCB*/
EVENTCOMPLETE (MESSAGEQCB);
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE FROM COPY FURNISHED TO DDC

6.4.1.2 REQUESTMESSAGE

```
PROCEDURE REQUESTMESSAGE (ECBPTR);

/*THE PARAMETERS STORED IN THE ECB ARE:
   EVENTFLAG:    EFLAG1..EFLAG15, SUSPEND
   MESSAGEIDPTR:   POINTER TO THE USER MESSAGEID PARAMETER
   MSLENGTHPTR:    POINTER TO THE USER MSLENGTH PARAMETER
   ANSLENGTHPTR:   POINTER TO THE USER ANSLENGTH PARAMETER*/

BEGIN
TASKNAME := ECBPTR^.TCBPTR^.ICBPTR^.TASKNAME;
IF    /*CAN THIS TASK RECEIVE MESSAGES?*/
   TASKNAME = NIL
THEN      /*NO, RETURN ERROR*/
   ECBPTR^.STATUS.ERROR := ERNOITC
ELSE
   BEGIN
   MESSAGEQCB := MESSAGETABLE [TASKNAME];
   /*MOVE ECB FIELDS TO QCB*/
   MESSAGEQCB := MESSAGETABLE [TASKNAME];
   MESSAGEQCB^.EFLAG := ECBPTR^.EFLAG;
   MESSAGEQCB^.STATUSPTR := ECBPTR^.STATUSPTR;
   /*FAKE LINKAGE FOR CALLCOMPLETE*/
   ACTIVEECB := MESSAGEQCB;
   END;
END.
```

6.4.1.3 TRANSFERMESSAGE

```
PROCEDURE TRANSFERMESSAGE (ECBPTR);

/*THE PARAMETERS STORED IN THE ECB ARE:
   EVENTFLAG:   EFLAG1..EFLAG15, SUSPEND
   MESSAGEID:      IDENTIFICATION OF THIS CONVERSATION
   MBUFFSPEC:   RECORD
                BUFFERID: INTEGER;
                ADDRESS: INTEGER;
                SIZE: INTEGER
                END;     */
BEGIN
/*FIND INSTRUCTION ECB ON MESSAGEQ FOR THIS TASK*/
FOUND := FALSE;
MESSAGEQCB := GETMESSAGEQCB (ECBPTR^.TCBPTR^.TASKNAME);
INSTRUCTIONECBPTR :=  MESSAGEQCB.QHEAD.FIRST;
WHILE     /*SEARCH INSTRUCTION LIST*/
   INSTRUCTIONECBPTR <> NIL AND NOT FOUND
DO
   BEGIN
   IF     /*IS THIS IT?*/
      ECBPTR^.MESSAGEID := INSTRUCTIONECBPTR^.MESSAGEID
   THEN      /*YES, END SEARCH*/
      FOUND := TRUE
   ELSE      /*NO, KEEP LOOKING*/
      INSTRUCTIONECBPTR := INSTRUCTIONECBPTR^.ECBQLINK.FLINK;
   END;
IF     /*ERROR IF NOT FOUND*/
   NOT FOUND
THEN
   ECBPTR^.STATUS.ERROR := ERMESSAGEID
ELSE      /*FOUND, PROCEED WITH TRANSFER*/
   BEGIN
   IF     /*IS THE SENDER IN THIS PROCESSOR?*/
      ECBPTR^.MESSAGEID.PROCESSORID := THISPROCESSOR
   THEN      /*YES, DO THE TRANSFER SYNCHRONOUSLY*/
      BEGIN
      /*MOVE MESSAGE FROM SENDER'S BUFFER TO RECEIVER'S BUFFER*/
      /*DECREMENT BCB IOINPROGRESS FLAGS FOR SENDER'S
        MESSAGE BUFFER BCB AND, IF SENDER DOES NOT OWN
        THE BCB, THE OWNER'S BCB*/
      ECBPTR^.STATUS.ERROR := NOERROR; /*DONE*/
      END
   ELSE      /*MUST TRANSMIT OVER BUS*/
      BEGIN
      /*INCREMENT THIS PARTY'S BCB IOINPROGRESS FLAG AND,
        IF HE DOES NOT OWN THE BUFFER, THE OWNER'S FLAG*/
      LINK (ECBPTR, TRANSFERMESSAGEQHEAD, ECBPTR^.ECBQLINK);
      EVENTCOMPLETE (TRANSFERMESSAGEQCB);
      END;
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
FUNCTION GETMESSAGEQCB    ( TASKNAME )

/*RETURNS QUEUE CONTROL BLOCK POINTER FROM MESSAGE TABLE FOR
  THIS  TASK*/

BEGIN
GETMESSAGEQCB := MESSAGETABLE  [TASKNAME];
END
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDQ

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.4.1.4 SENDANSWER

```
PROCEDURE SENDANSWER (ECBPTR);

/*THE PARAMETERS STORED IN THE ECB ARE:
   EVENTFLAG:  EFLAG1..EFLAG15, SUSPEND
   MESSAGEID:     IDENTIFICATION OF THIS CONVERSATION
   ABUFFSPEC:  RECORD
               BUFFERID: INTEGER;
               ADDRESS: INTEGER;
               SIZE: INTEGER
               END;    */

BEGIN
/*FIND INSTRUCTION ECB ON MESSAGEQ FOR THIS PROCESS*/
FOUND := FALSE;
MESSAGEQCB := GETMESSAGEQCB (ECBPTR^.TCBPTR^.PROCESSNAME);
INSTRUCTIONECBPTR :=  MESSAGEQCB.QHEAD.FIRST;
WHILE    /*SEARCH INSTRUCTION LIST*/
   INSTRUCTIONECBPTR <> NIL AND NOT FOUND
DO
   BEGIN
   IF    /*IS THIS IT?*/
      ECBPTR^.MESSAGEID := INSTRUCTIONECBPTR^.MESSAGEID
   THEN     /*YES, END SEARCH*/
      FOUND := TRUE
   ELSE     /*NO, KEEP LOOKING*/
      INSTRUCTIONECBPTR := INSTRUCTIONECBPTR^.ECBQLINK.FLINK;
   END;
IF    /*ERROR IF NOT FOUND*/
   NOT FOUND
THEN
   ECBPTR^.STATUS.ERROR := ERMESSAGEID
ELSE     /*FOUND, PROCEED WITH SEND*/
   BEGIN
   IF    /*IS THE SENDER IN THIS PROCESSOR?*/
      ECBPTR^.ANSWERID.PROCESSORID := THISPROCESSOR
   THEN     /*YES, DO THE SEND SYNCHRONOUSLY*/
      BEGIN
      /*MOVE ANSWER FROM RECEIVER'S BUFFER TO SENDER'S BUFFER*/
      /*DECREMENT BCB IOINPROGRESS FLAGS FOR SENDER'S
        ANSWER BUFFER BCB AND, IF SENDER DOES NOT OWN
        THE BCB, THE OWNER'S BCB*/
      ECBPTR^.STATUS.ERROR := NOERROR; /*DONE*/
      /*REMOVE INSTRUCTION ECB FROM MESSAGEQ*/
      DELINK (INSTRUCTIONECBPTR, MESSAGEQCB.QHEAD,
         INSTRUCTIONECBPTR^.ECBQLINK);
      /*SIGNAL SENDER THAT MESSAGE HAS ARRIVED*/
      EVENTCOMPLETE (INSTRUCTIONECBPTR);
      END
```

```
   ELSE      /*MUST TRANSMIT OVER BUS*/
      BEGIN
      /*INCREMENT THIS PARTY'S BCB IOINPROGRESS FLAG AND,
        IF HE DOES NOT OWN THE BUFFER, THE OWNER'S FLAG*/
      LINK (ECBPTR, SENDANSWERQHEAD, ECBPTR^.ECBQLINK);
      EVENTCOMPLETE (SENDANSWERQCB);
      END;
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.4.2 DISPATCHER

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE DISPATCHER;

/*INPUT:  ENQUEUED ECB'S FROM THE BUS INTERFACE DRIVER AND FROM
          THE MODULES OF THE ROUTER.
  OUTPUT: ECB'S FOR THE BUS INTERFACE DRIVER OR EVENT COMPLETION
          TO THE SENDING AND RECEIVING TASKS*/

BEGIN
/*SET UP TIMEOUT LIST*/
WAITFORTOPENTRY (EFLAG1, MESSAGEINSTRUCTIONQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG2, MESSAGEOUTQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG3, SENDANSWERQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG4, TRANSFERMESSAGEQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG5, ANSINSTRUCTIONQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG6, MSBUFFCODEQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG7, ANSBUFFCODEQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG8, COMPLETIONQCB, VAR STATUS);
WHILE    /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITANY (EFLAG1+EFLAG2+EFLAG3+EFLAG4+EFLAG5+EFLAG6
            +EFLAG7+EFLAG8+EFLAG9, EFFOUND);
   /*RETURN WITH QUEUE ENTRY OR TIMEOUT*/
   /*ECBPTR IS SET TO THE ECB OF THE TOP ENTRY FOR EFLAGS 1 THROUGH
     8, AND IS NOT SET FOR EFLAG9, THE TIMEOUT EVENT FLAG*/
   /*THE WAITFORTOPENTRY FOR THE QUEUE FOUND IS REISSUED; THE
     TIMEOUT PORTION OF THE CASE STATEMENT BELOW WILL RESET THE
     PROPER TIMEOUT VALUE*/
   CASE EFFOUND OF
      EFLAG1:       /*MESSAGEINSTRUCTIONQ*/
         BEGIN
         /*DETERMINE DESTINATION*/
         TASKNAME := ECBPTR^.TASKNAME;
         MESSAGERECEIVED (ECBPTR);       /*LINKS TO MESSAGEQ AND
                                           NOTIFIES RECEIVER*/
         END;
      EFLAG2, EFLAG3:        /*MESSAGEOUTQ, SENDANSWERQ*/
         BEGIN
         IF
            EFFOUND = EFLAG2
         THEN
            BEGIN
            /*SET CONVERSATION TIMEOUT IN TIMEOUT LIST.  IF THIS
              TIMEOUT VALUE IS SHORTER THAN THE CURRENT HEAD OF
              THE TIMEOUT LIST, DELETE THE CURRENT TIMEOUT AND
              USE THE NEW TIMEOUT VALUE TO REISSUE THE TIMEOUT.*/
            END;
         IF    /*SHORT MESSAGE OR ANSWER*/
            ECBPTR^.MSLENGTH < ECBSPACEFORMESSAGE
         THEN
```

```
            BEGIN
            /*MOVE MESSAGE OR ANSWER INTO ECB*/
            /*PASS ECB TO BUS INTERFACE DRIVER, INDICATING MESSAGE OR
              ANSWER IN THE INSTRUCTION.  ECB RETURNS ON SENTMESSAGEQ
              FOR MESSAGE, ON COMPLETIONQ FOR ANSWER. */
            END
         ELSE      /*LONG MESSAGE OR ANSWER*/
            BEGIN
            /*PASS ECB TO BUS INTERFACE DRIVER, INDICATING INSTRUCTION
              ONLY.  RETURN FOR BOTH CASES IS ON THE SENTINSTRUCTIONQ. */
            END;
         END;      /*EFLAGS 2 AND 3*/
      EFLAG4, EFLAG5:       /*TRANSFERMESSAGEQ, ANSINSTRUCTIONQ*/
         BEGIN
         IF     /*ANSINSTRUCTIONQ?*/
            EFFOUND := EFLAG5
         THEN      /*YES, GET BUFFER ADDRESS*/
            BEGIN
            MATCHECBPTR := SENTMESSAGEQHEAD.FIRST;
            FOUND := FALSE;
            WHILE
               MATCHECBPTR <> NIL AND NOT FOUND
            DO
               BEGIN
               IF
                  MATCHECBPTR^.MESSAGEID := ECBPTR^.MESSAGEID
               THEN
                  FOUND := TRUE
               ELSE
                  MATCHECBPTR := MATCHECBPTR^.ECBQLINK.FLINK;
               END;
            IF
               NOT FOUND
            THEN
               BEGIN
               /*NOTIFY LERC OF ERROR*/
               GOTO 10
               END;
            /*GET USER'S BUFFER INFORMATION FROM MATCHECB*/
            END
         ELSE      /*TRANSFERMESSAGEQ*/
            /*GET BUFFER INFORMATION FROM REQUEST ECB*/;
         IF     /*IS MESSAGE OR ANSWER IN INSTRUCTION?*/
            INSTRUCTIONLENGTH > BASICINSTRUCTION
         THEN      /*MOVE IT NOW*/
            BEGIN
            /*MOVE DATA IN ECB TO USER BUFFER SPACE*/
            /*SIGNAL EVENT COMPLETE TO USER*/
            GOTO 10
            END;
         ECBPTR^.BUFFERCODE := BUFFERCODE;
         BUFFERCODE := BUFFERCODE + 1;
         /*PASS ECB TO BUS INTERFACE DRIVER, REQUESTING BUFFER CODE
           TRANSMISSION AND RETURN OF THE ECB ON THE COMPLETIONQ
           WITH THE DATA IN THE USER'S BUFFER*/
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
10:             /*EXIT WITH ERROR*/
                END;      /*EFLAGS 4 AND 5*/
             EFLAG6, EFLAG7:        /*MSBUFFCODEQ, ANSBUFFCODEQ*/
                BEGIN
                FOUND := FALSE;
                MATCHECBPTR := SENTINSTRUCTIONQHEAD.FIRST;
                WHILE
                   MATCHECBPTR <> NIL AND NOT FOUND
                DO
                   BEGIN
                   IF
                      MATCHECBPTR^.MESSAGEID := ECBPTR^.MESSAGEID
                   THEN
                      FOUND := TRUE
                   ELSE
                      MATCHECBPTR := MATCHECBPTR^.ECBQLINK.FLINK;
                   END;
                IF
                   NOT FOUND
                THEN
                   BEGIN
                   /*NOTIFY LERC OF ERROR*/
                   GOTO 20
                   END;
                /*GET USER BUFFER SPEC FROM MATCHECB*/
                /*PASS ECB TO BUS INTERFACE DRIVER, REQUESTING THAT
                  THE DATA FROM THE USER'S BUFFER BE TRANSMITTED
                  WITH ECBPTR^.BUFFERCODE.  THE BUS INTERFACE WILL
                  PERFORM ANY NECESSARY PACKETIZATION*/
                /*RETURN IS ON THE SENTMESSAGEQ FOR EFLAG6, AND
                  ON THE COMPLETIONQ FOR EFLAG7*/
20:             /*EXIT WITH ERROR OR SHORT MESSAGE/ANSWER*/
                END;      /*EFLAGS 6 AND 7*/
             EFLAG8:      /*COMPLETIONQ*/
                BEGIN
                /*THE BUS INTERFACE DRIVER RETURNS INFORMATION ON THE
                  NATURE OF THE COMPLETION IN ECBPTR^.SUBFUNC.*/
                CASE ECBPTR^.SUBFUNC OF
                   MESSAGERECEIVED, ANSWERSENT:
                      BEGIN
                      ECBPTR^.STATUS.ERROR := NOERROR;
                      EVENTCOMPLETE (ECBPTR);
                      IF
                         ECBPTR^.SUBFUNC = ANSWERSENT
                      THEN
                         BEGIN
                         FOUND := FALSE;
                         MESSAGEQCB :=
                            GETMESSAGEQCB (ECBPTR^.TCBPTR^.PROCESSNAME);
                         MATCHECBPTR := MESSAGEQCB.QHEAD.FIRST;
                         WHILE
                            MATCHECBPTR <> NIL AND NOT FOUND
                         DO
                            BEGIN
                            IF
```

```
                    MATCHECBPTR^.MESSAGEID := ECBPTR^.MESSAGEID
                  THEN
                    FOUND := TRUE
                  ELSE
                    MATCHECBPTR := MATCHECBPTR^.ECBQLINK.FLINK;
                  END;
                IF
                  NOT FOUND
                THEN
                  BEGIN
                  /*NOTIFY LERC OF ERROR*/
                  GOTO 30
                  END;
                DELINK (MATCHECBPTR, MESSAGEQCB.QHEAD,
                         MATCHECBPTR^.ECBQLINK);
                LINK (MATCHECBPTR, FREEECBQHEAD,
                    MATCHECBPTR^.ECBQLINK);
                END;
              END;      /*FOR MESSAGE RECEIVED OR ANSWER SENT*/
            ANSWERRECEIVED:
              BEGIN
              FOUND := FALSE;
              MATCHECBPTR := SENTMESSAGEQHEAD.FIRST;
              WHILE
                MATCHECBPTR <> NIL AND NOT FOUND
              DO
                BEGIN
                IF
                  MATCHECBPTR^.MESSAGEID := ECBPTR^.MESSAGEID
                THEN
                  FOUND := TRUE
                ELSE
                  MATCHECBPTR := MATCHECBPTR^.ECBQLINK.FLINK;
                END;
              IF
                NOT FOUND
              THEN
                BEGIN
                /*NOTIFY LERC OF ERROR*/
                GOTO 30
                END;
              DELINK (MATCHECBPTR, SENTMESSAGEQCB.QHEAD,
                     MATCHECBPTR^.ECBQLINK);
              LINK (ECBPTR, FREEECBQHEAD,
                  ECBPTR^.ECBQLINK);
              MATCHECBPTR^.STATUS.ERROR := NOERROR;
              EVENTCOMPLETE (MATCHECB);
              END;
            END;
30:       /*EXIT WITH ERROR OR COMPLETION*/
          END;      /*EFLAG 8*/
        EFLAG9:     /*TIMEOUT*/
          /*REMOVE TOP ENTRY IN TIMEOUT LIST*/
          /*REQUEST NEXT TIMEOUT WITH DELAY (EFLAG9, ...);*/
          /*FIND MATCHING ECB IN MESSAGEOUTQ, SENTINSTRUCTIONQ,
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
        OR SENTMESSAGEQ, AND PLACE IN ECBPTR AFTER DELINKING*/
      ECBPTR^.STATUS.ERROR := ERTIMEOUT;
      EVENTCOMPLETE (ECBPTR);
      END;      /*EFLAG 9*/
    END;      /*MAIN CASE STATEMENT*/
  END;      /*INFINITE LOOP*/
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.5 FILE MANAGEMENT AND INPUT/OUTPUT FUNCTIONS

6.5.1 FILE MANAGEMENT

6.5.1.1 CREATE

```
PROCEDURE CREATESYNC (ECBPTR);

/*ENQUEUES REQUEST ECB FOR CREATE*/

BEGIN
LINK (ECBPTR, CREATEQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (CREATEQCB);
END.
```

```
PROCEDURE CREATE;

/*INPUT:   EVENT CONTROL BLOCK FROM CREATEQUEUE*
           CONTAINING:
             FILENAME: FILENAMETYPE;
             FILETYPE: SEGMENTED, CONTIGUOUS;
             FILESIZE:  INTEGER;   /*SIZE OF FILE FOR CONTIGUOUS,
                                    SIZE OF DATA SEGMENT FOR
                                SEGMENTED, GIVEN IN WORDS*/
  OUTPUT:  CREATED FILE OF SPECIFIED NAME, TYPE AND SIZE
             OR ERROR:
                ERFILEEXISTS
                ERCONTIG          INSUFFICIENT CONTIGUOUS SPACE
                ERDIR             DIRECTORY ERROR*/

BEGIN
WHILE    /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   /*SUSPEND UNTIL REQUEST COMES IN*/
   WAITFORTOPENTRY (SUSPEND, CREATEQHEAD, VAR STATUS);
   ECBPTR := STATUS.VALUE1;
   DELINK (ECBPTR, CREATEQHEAD, ECBQLINK);
   GETBUFFER (DIRBLOCKPTR, DIRBLOCKSIZE, BUFFID, VAR STATUS);
   MAPTOBUFFER (BUFFID, VAR STATUS);
   FINDDIRECTORYENTRY (ECBPTR^, DIRBLOCKPTR^, VAR DIRBLOCK,
                          VAR ENTRY);
   IF    /*ERROR IF FILE ALREADY EXISTS*/
      ENTRY <> NIL
   THEN
      ECBPTR^.STAT := ERFILEEXISTS
   ELSE     /*CREATE FILE OF SPECIFIED TYPE*/
      BEGIN
      /*GET FULL FILE IF CONTIGUOUS, OR FIRST SPB IF SEGMENTED*/
      CASE ECBPTR^.FILETYPE OF
         CONTIGUOUS:
            GETCONTIGUOUSSPACE (ECBPTR^.FILESIZE, BUFFER^,
                                VAR STARTBLOCK);
         SEGMENTED:
            GETCONTIGUOUSSPACE (SPBSIZE, BUFFER^,
                                VAR STARTBLOCK);
         END;
      IF    /*WAS ENOUGH SPACE FOUND?*/
         STARTBLOCK := NIL
      THEN     /*NO, RETURN ERROR*/
         ECBPTR^.STATUS.ERROR := ERCONTIG
      ELSE     /*SPACE FOUND, ENTER FILE*/
         BEGIN
         GETFREEDIRECTORYENTRY (ECBPTR, DIRBLOCKPTR, VAR DIRBLOCK,
                                VAR ENTRY);
         IF    /*WAS A FREE ENTRY FOUND?*/
            ENTRY = NIL
         THEN     /*NO, RETURN ERROR*/
            BEGIN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
            ECBPTR^.STATUS.ERROR := ERDIR
            CASE ECBPTR^.FILETYPE OF
               CONTIGUOUS:
                  RELEASECONTIGUOUSSPACE (STARTBLOCK, BUFFER^,
                                          ECBPTR^.FILESIZE);
               SEGMENTED:
                  RELEASECONTIGUOUSSPACE (STARTBLOCK, BUFFER^,
                                          SPBSIZE);
               END;
            END
         ELSE      /*ENTRY FOUND, PROCEED*/
            BEGIN
            DIRBLOCKPTR^. [ENTRY].FILENAME := ECBPTR^.FILENAME;
            DIRBLOCKPTR^. [ENTRY].STARTADDR := STARTBLOCK;
            IF     /*CONTIGUOUS OR SEGMENTED?*/
               ECBPTR^.FILETYPE = CONTIGUOUS
            THEN      /*SET DIRENTRY FOR CONTIGUOUS*/
               BEGIN
               DIRBLOCKPTR^. [ENTRY].ATTRIBUTES.CONTIGUOUS := TRUE;
               DIRBLOCKPTR^. [ENTRY].FILESIZE := ECBPTR^.SIZE;
               /*SET DIRBLOCKPTR^. [ENTRY].ENDADDR EQUAL TO THE DISC
                 BLOCK WHICH REPRESENTS CONTIGUOUS SPACE
                 EQUAL TO N BLOCKS, WHERE N IS:
                  (ECBPTR^.SIZE+DISCBLOCKSIZE-1) DIV DISCBLOCKSIZE*/
               END
            ELSE      /*SEGMENTED*/
               BEGIN
               DIRBLOCKPTR^. [ENTRY].ATTRIBUTES.SEGMENTED := TRUE;
               DIRBLOCKPTR^. [ENTRY].FILESIZE := 0;
               DIRBLOCKPTR^. [ENTRY].DATASEGMENTSIZE := ECBPTR^.SIZE;
               DIRBLOCKPTR^. [ENTRY].ENDADDR := STARTBLOCK;
               END;
            GETDATE (DATE);
            GETTIME (TIME);
            DIRBLOCKPTR^. [ENTRY].CRDATE := DATE;
            DIRBLOCKPTR^. [ENTRY].CRTIME := TIME;
            /*WRITE DIRBLOCK BACK ON THE DISC*/
            ECBPTR^.STATUS.ERROR := NOERROR;
            END;
         END;
      END;
   EVENTCOMPLETE (ECBPTR);
   RELEASEBUFFER (SUSPEND, BUFFID, VAR STATUS);
   END;
END.
```

* The CREATEQUEUE is a queue of ECB's, each containing the information needed to process an application's request to have a file created.

```
PROCEDURE FINDDIRECTORYENTRY (ECB, BUFFER, VAR DIRBLOCK, VAR ENTRY);

/*THIS PROCEDURE DEPENDS UPON THE DIRECTORY STRUCTURE
OF THE DEVELOPMENT OPERATING SYSTEM*/
BEGIN
FOUND := FALSE;
DIRBLOCK := FIRSTDIRBLOCK;
WHILE    /*LOOP UNTIL ENTRY OR END OF DIRECTORY FOUND*/
   DIRBLOCK <= LASTDIRBLOCK
   AND NOT FOUND
DO
   BEGIN
   /*GET DIRBLOCK FROM DISC AND PLACE IN BUFFER*/
   ENTRY := 1;
   WHILE    /*CHECK EACH ENTRY IN THE DIRECTORY BLOCK*/
      ENTRY <= ENTRIESINBLOCK AND NOT FOUND
   DO
      BEGIN
      IF    /*IS THIS THE ENTRY SOUGHT?*/
         BUFFER [ENTRY]. FILENAME =
            ECB. FILENAME
      THEN
         FOUND := TRUE
      ELSE
         /*KEEP TRACK OF ENTRIES CHECKED IN THIS BLOCK*/
         ENTRY := ENTRY + 1;
      END;
   IF    /*IF FOUND, DON'T INCREMENT DIRBLOCK*/
      NOT FOUND
   THEN
      /*DIRBLOCK := NEXT DIRECTORY BLOCK (DEPENDS ON DISC HARDWARE)*/
   END;
IF    /*SEND BACK NIL IN ENTRY IF NOT FOUND*/
   NOT FOUND
THEN
   ENTRY := NIL;
END.
```

```
PROCEDURE GETFREEDIRECTORYENTRY (ECB, BUFFER, VAR DIRBLOCK, VAR ENTRY);

/*THIS PROCEDURE DEPENDS UPON THE DIRECTORY STRUCTURE
OF THE DEVELOPMENT OPERATING SYSTEM*/
BEGIN
FOUND := FALSE;
DIRBLOCK := FIRSTDIRBLOCK;
WHILE    /*LOOP UNTIL ENTRY OR END OF DIRECTORY FOUND*/
   DIRBLOCK <= LASTDIRBLOCK
   AND NOT FOUND
DO
   BEGIN
   /*GET DIRBLOCK FROM DISC AND PLACE IN BUFFER*/
   ENTRY := 1;
   WHILE    /*CHECK EACH ENTRY IN THE DIRECTORY BLOCK*/
      ENTRY <= ENTRIESINBLOCK AND NOT FOUND
   DO
      BEGIN
      IF    /*IS THIS A FREE ENTRY?*/
         BUFFER [ENTRY].FILENAME = NIL
      THEN
         FOUND := TRUE
      ELSE
         /*KEEP TRACK OF ENTRIES CHECKED IN THIS BLOCK*/
         ENTRY := ENTRY + 1;
      END;
   IF    /*IF FOUND, DON'T INCREMENT DIRBLOCK*/
      NOT FOUND
   THEN
      /*DIRBLOCK := NEXT DIRECTORY BLOCK (DEPENDS ON DISC HARDWARE)*/
   END;
IF    /*SEND BACK NIL IN ENTRY IF NOT FOUND*/
   NOT FOUND
THEN
   ENTRY := NIL;
END.
```

```
PROCEDURE GETCONTIGUOUSSPACE (SIZE, BUFFER, VAR STARTBLOCK);


BEGIN
BLOCKSSOUGHT := (SIZE+DISCBLOCKSIZE-1) DIV DISCBLOCKSIZE;
MAPBLOCKINDEX := FIRSTMAPBLOCK;
FOUND := FALSE;
STARTBLOCK := NIL;
WHILE     /*LOOK FOR CONTIGUOUS SPACE*/
   MAPBLOCKINDEX <= LASTMAPBLOCK
   AND NOT FOUND
DO
   BEGIN
   /*READ MAPBLOCKINDEX INTO BUFFER*/
   BITINDEX := 1;
   WHILE     /*CHECK EACH BIT FLAG IN THE BLOCK*/
      BITINDEX <= BITSINMAPBLOCK
      AND NOT FOUND
   DO
      BEGIN
      IF     /*IS THIS BLOCK FREE?*/
         BUFFER [BITINDEX] <> FREE
      THEN      /*NO, KEEP LOOKING*/
         BEGIN
         BLOCKSFOUND := 0;
         STARTBLOCK := NIL
         END
      ELSE      /*ADD TO CONTIGUOUS BLOCK COUNT*/
         BEGIN
         IF     /*RESET STARTBLOCK?*/
            STARTBLOCK := NIL
         THEN      /*YES*/
            STARTBLOCK := /*ADDRESS OF BLOCK REPRESENTED BY
                                MAPBLOCKINDEX [BITINDEX]*/;
         BLOCKSFOUND := BLOCKSFOUND + 1;
         BITINDEX := BITINDEX + 1;
      IF     /*CHECK FOR ENOUGH SPACE FOUND*/
         BLOCKSFOUND = BLOCKSSOUGHT
      THEN      /*SIGNAL END OF LOOP*/
         FOUND := TRUE;
      END;
   /*MAPBLOCKINDEX := NEXT MAP BLOCK*/;
   END;
IF     /*SPACE FOUND?*/
   FOUND
THEN      /*MARK SPACE USED IN MAP FILE*/
   BEGIN
   /*MAPBLOCKINDEX := THE MAP BLOCK CONTAINING STARTBLOCK'S
     FREE/USED BIT*/;
   /*BITINDEX := THE BIT REPRESENTING STARTBLOCK IN MAPBLOCKINDEX*/;
   BLOCKSSET := 0;
   WHILE     /*ALL BITS IN THIS BLOCK*/
      BLOCKSSET < BLOCKSSOUGHT
   DO
      BEGIN
```

```
        WHILE     /*NEW BLOCK IF DONE WITH THIS ONE*/
           BITINDEX <= BITSINMAPBLOCK
           AND BLOCKSSET < BLOCKSSOUGHT
        DO
           BEGIN
           BUFFER [BITINDEX] := USED;
           BITINDEX := BITINDEX + 1;
           BLOCKSSET := BLOCKSSET + 1;
           END;
        /*WRITE MAP BACK ONTO DISC*/
        /*ADJUST MAPBLOCKINDEX TO NEXT MAP BLOCK*/
        BITINDEX := 1;
        END;
     END
ELSE      /*RETURN NOT FOUND INDICATION*/
     STARTBLOCK := NIL;
END.
```

6.5.1.2 DELETE

```
PROCEDURE DELETESYNC (ECBPTR);

/*ENQUEUES REQUEST ECB FOR DELETE*/

BEGIN
LINK (ECBPTR, DELETEQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (DELETEQHEAD);
END.
```

```
PROCEDURE DELETE;

/*INPUT:   EVENT CONTROL BLOCK FROM DELETEQUEUE
  OUTPUT:  FILE DELETED AND BLOCKS RETURNED TO FREE BLOCK LIST
             OR ERROR:
                ERNOFILE
                ERPROTECTED        FILE PROTECTED
                EROPEN             FILE OPEN*/

BEGIN
WHILE      /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, DELETEQHEAD, VAR STATUS);
   ECBPTR := DELETEQCB^.QHEAD.FIRST;
   DELINK (ECBPTR, DELETEQHEAD, ECBPTR^.ECBQLINK);
   FINDDEVICE (ECBPTR, VAR DDBPOINTER);
   FDBPTR := DDBPOINTER^.FDBQHEAD.FIRST;
   WHILE     /*WALK THROUGH THE FDB QUEUE FOR THIS DISC*/
      FDBPTR <> NIL
   DO
      BEGIN
      IF     /*IS THIS THE FILE TO BE DELETED?*/
         FDBPTR^.FILENAME = ECBPTR^.FILENAME
      THEN      /*YES, RETURN ERROR*/
         BEGIN
         ECBPTR^.STATUS.ERROR := ERFILEOPEN
         GOTO 15
         END;
      FDBPTR := FDBPTR^.FDBQLINK.FLINK;
      END;
   GETBUFFER (DIRBUFFADDR, DIRBLOCKSIZE, DIRBUFFID, VAR STATUS);
   MAPTOBUFFER (DIRBUFFID, VAR STATUS);
   GETBUFFER (SPBADDR, SPBSIZE, SPBID, VAR STATUS);
   MAPTOBUFFER (SPBID, VAR STATUS);
   GETBUFFER (MAPBUFFADDR, MAPBLOCKSIZE, MAPBUFFID, VAR STATUS);
   MAPTOBUFFER (MAPBUFFID, VAR STATUS);
   FINDDIRECTORYENTRY (ECBPTR, DIRBUFFADDR, VAR DIRBLOCK,
                          VAR ENTRY);
   IF      /*CHECK FOR FILE NOT FOUND*/
      ENTRY = NIL
   THEN    /*SEND ERROR BACK TO TASK*/
      BEGIN
      ECBPTR^.STAT := ERNOFILE
      GOTO 12
      END;
   BEGIN
   IF     /*CHECK FOR FILE PROTECTED*/
      DIRBUFFADDR [ENTRY].ATTRIBUTES.PROTECTED
   THEN      /*RETURN ERROR*/
      BEGIN
      ECBPTR^.STATUS.ERROR := ERPROTECTED
      GOTO 12
      END;
```

```
      /*REMOVE FILE FROM DIRECTORY BY SETTING FILENAME TO NIL
        (DEPENDS ON DIRECTORY CONVENTIONS OF DEVELOPMENT SYSTEM)*/
      DIRBUFFADDR [ENTRY].FILENAME.NAME := NIL;
      /*WRITE DIRBLOCK BACK ONTO DISC*/
      ECBPTR^.STATUS.ERROR := NOERROR;
      BEGIN
      CASE DIRBUFFADDR [ENTRY].FILETYPE OF
         CONTIGUOUS: BEGIN
                     /*MARK ALL BLOCKS FREE IN MAP*/
                     RELEASECONTIGUOUSSPACE (FILESIZE,
                        MAPBUFFADDR^, DIRBUFFADDR [ENTRY].STARTADDR);
         SEGMENTED:  BEGIN
                     THISSPB := DIRBUFFADDR [ENTRY].STARTADDR;
                     WHILE    /*FOR ALL SPB'S IN FILE*/
                        THISSPB <> NIL
                     DO
                        BEGIN
                        /*READ THISSPB FROM DISC*/
                        NEXTSPB := SPBADDR^.NEXTSPB;
                        DSPINDEX := 1;
                        WHILE    /*ALL DATA SEGS IN SPB*/
                           DSPINDEX <= MAXDSP
                           AND SPBADDR^.DATASEGPTR [DSPINDEX] <> NIL
                        DO
                           BEGIN
                           RELEASECONTIGUOUSSPACE
                              (DSPSIZE, MAPBUFFADDR^,
                               SPBADDR^.DATASEGPTR [DSPINDEX]);
                           DSPINDEX := DSPINDEX + 1;
                           END;
                        RELEASECONTIGUOUSSPACE (SPBSIZE,
                           MAPBUFFER, THISSPB);
                        THISSPB := NEXTSPB;
                        IF    /*ARE THERE MORE SPB'S?*/
                           SPBADDR^.DATASEGPTR [DSPINDEX] <> NIL
                        THEN     /*NO*/
                           THISSPB := NIL;       /*SPACE RELEASED*/
                        END;
                     END;
         END;     /*END CASE*/

12:   /*EXIT HERE WITH ERNOFILE OR ERPROTECTED*/
      RELEASEBUFFER (SUSPEND, SPBID, VAR STATUS);
      RELEASEBUFFER (SUSPEND, MAPBUFFID, VAR STATUS);
      RELEASEBUFFER (SUSPEND, DIRBUFFID, VAR STATUS);

15:   /*RETURN HERE WITH ERFILEOPEN*/
      EVENTCOMPLETE (ECBPTR);
      END;
   END.
```

```
PROCEDURE RELEASECONTIGUOUSSPACE (STARTADDR, MAPBUFFER, SIZE);

   /*INPUT:  STARTING DISC ADDRESS AND SIZE OF CONTIGUOUS
             AREA TO BE RELEASED.
     OUTPUT: CONTIGUOUS SPACE RELEASED TO FREE BLOCK LIST*/

BEGIN
/*CALCULATE SECTORS IN RELEASE AREA*/
BLOCKSSOUGHT := (SIZE + DISCBLOCKSIZE - 1) DIV DIRBLOCKSIZE;
/*MAPBLOCKINDEX := THE MAP BLOCK CONTAINING STARTBLOCK'S
  FREE/USED BIT*/;
/*BITINDEX := THE BIT REPRESENTING STARTBLOCK IN MAPBLOCKINDEX*/;
BLOCKSSET := 0;
WHILE    /*ALL BITS IN THIS BLOCK*/
   BLOCKSSET < BLOCKSSOUGHT
DO
   BEGIN
   /*READ MAP BLOCK AT MAPBLOCKINDEX*/
   WHILE    /*NEW BLOCK IF DONE WITH THIS ONE*/
      BITINDEX <= BITSINMAPBLOCK
      AND BLOCKSSET < BLOCKSSOUGHT
   DO
      BEGIN
      BUFFER [BITINDEX] := FREE;
      BITINDEX := BITINDEX + 1;
      BLOCKSSET := BLOCKSSET + 1;
      END;
   /*WRITE MAP BACK ONTO DISC*/
   /*ADJUST MAPBLOCKINDEX TO NEXT MAP BLOCK*/
   BITINDEX := 1;
   END;
END.
```

6.5.1.3 RENAME

```
PROCEDURE RENAMESYNC (ECBPTR);

/*ENQUEUES THE REQUEST ECB FOR RENAME*/

BEGIN
LINK (ECBPTR, RENAMEQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (RENAMEQCB);
END.
```

```
PROCEDURE RENAME;

/*INPUT:   EVENT CONTROL BLOCK FROM RENAMEQUEUE,
           CONTAINING:
             OLDFILE: FILENAME;
             NEWFILE: FILENAME;   /*REPLACES OLDFILE IN DIRECTORY*/
  OUTPUT:  RENAMED FILE
             OR ERROR:
                ERNOFILE
                ERPROTECTED        FILE PROTECTED
                EROPEN             FILE OPEN*/

BEGIN
WHILE      /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, RENAMEQHEAD, VAR STATUS);
   ECBPTR := RENAMEQCB^.QHEAD.FIRST;
   DELINK (ECBPTR, RENAMEQHEAD, ECBPTR^.ECBQLINK);
   FINDDEVICE (ECBPTR, VAR DDBPOINTER);
   FDBPTR := DDBPOINTER^.FDBQHEAD.FIRST;
   WHILE     /*WALK THROUGH THE FDB QUEUE FOR THIS DISC*/
      FDBPTR <> NIL
   DO
      BEGIN
      IF     /*IS THIS THE FILE TO BE RENAMED?*/
         FDBPTR^.FILENAME = ECBPTR^.OLDFILENAME
      THEN      /*YES, RETURN ERROR*/
         BEGIN
         ECBPTR^.STATUS.ERROR := ERFILEOPEN
         GOTO 15
         END;
      FDBPTR := FDBPTR^.FDBQLINK.FLINK;
      END;
   GETBUFFER (DIRBUFFADDR, DIRBLOCKSIZE, DIRBUFFID, VAR STATUS);
   MAPTOBUFFER (DIRBUFFID, VAR STATUS);
   FINDECTORYENTRY (ECBPTR, DIRBUFFADDR, VAR DIRBLOCK,
                    VAR ENTRY);
   IF      /*CHECK FOR FILE NOT FOUND*/
      ENTRY = NIL
   THEN   /*SEND ERROR BACK TO TASK*/
      BEGIN
      ECBPTR^.STAT := ERNOFILE
      GOTO 12
      END;
   IF     /*CHECK FOR FILE PROTECTED*/
      DIRBUFFADDR [ENTRY].ATTRIBUTES.PROTECTED
   THEN      /*RETURN ERROR*/
      BEGIN
      ECBPTR^.STATUS.ERROR := ERPROTECTED
      GOTO 12
      END;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
       /*RENAME FILE BY SETTING FILENAME TO NEWFILENAME*/
       DIRBUFFADDR [ENTRY].FILENAME.NAME := ECBPTR^.NEWFILENAME;

       /*WRITE DIRBLOCK BACK ONTO DISC*/
       ECBPTR^.STATUS.ERROR := NOERROR;   /*SUCCESS*/

12:    /*EXIT HERE WITH ERNOFILE OR ERPROTECTED*/
       RELEASEBUFFER (SUSPEND, DIRBUFFID, VAR STATUS);

15:    /*RETURN HERE WITH ERFILEOPEN*/
       EVENTCOMPLETE (ECBPTR);
       END;
    END.
```

6.5.1.4 SETATTRIBUTES

```
PROCEDURE SETATTRSYNC (ECBPTR);

/*ENQUEUES THE REQUEST ECB FOR SETATTRIBUTES*/

BEGIN
LINK (ECBPTR, SETATTRIBUTESQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (SETATTRIBUTESQCB);
END.
```

```
PROCEDURE SETATTRIBUTES;

/*INPUT:   EVENT CONTROL BLOCK FROM SETATTRIBUTESQUEUE,
           CONTAINING:
             FILENAME -
             ATTRIBUTES = RECORD
                            READPROTECT: BOOLEAN;
                            WRITEPROTECT: BOOLEAN;
                            PROTECTED: BOOLEAN
                            END;
  OUTPUT:  ATTRIBUTES OF REQUESTED DISC FILE
           SET AS REQUESTED OR ERROR:
                ERFILEOPEN
                ERNOFILE*/

BEGIN
WHILE      /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, SETATTRIBUTESQHEAD, VAR STATUS);
   ECBPTR := SETATTRIBUTESQCB^.QHEAD.FIRST;
   DELINK (ECBPTR, SETATTRIBUTESQHEAD, ECBPTR^.ECBQLINK);
   FINDDEVICE (ECBPTR, VAR DDBPOINTER);
   IF    /*DISC OR OTHER DEVICE?*/
      DDBPTR^.ATTRIBUTES.DIRECTORY
   THEN      /*DISC, FIND FILENAME*/
      BEGIN
      FDBPTR := DDBPOINTER^.FDBQHEAD.FIRST;
      FOUND := FALSE;
      WHILE    /*WALK THROUGH THE FDB QUEUE FOR THIS DISC*/
         FDBPTR <> NIL AND NOT FOUND
      DO
         BEGIN
         IF    /*IS THIS THE FILE?*/
            FDBPTR^.FILENAME = ECBPTR^.FILENAME
         THEN      /*YES, CAN'T SET ATTRIBUTES*/
            BEGIN
            ECBPTR^.STATUS.ERROR := ERFILEOPEN;
            FOUND := TRUE
            END;
         FDBPTR := FDBPTR^.FDBQLINK.FLINK;
         END;
      IF    /*IF OPEN, CAN'T SET ATTRIBUTES*/
         NOT FOUND
      THEN      /*NOT OPEN, SET ATTRIBUTES*/
         BEGIN
         GETBUFFER (DIRBUFFADDR, DIRBLOCKSIZE, DIRBUFFID, VAR STATUS);
         MAPTOBUFFER (DIRBUFFID, VAR STATUS);
         FINDECTORYENTRY (ECBPTR, DIRBUFFADDR, VAR DIRBLOCK,
                          VAR ENTRY);
         IF      /*CHECK FOR FILE NOT FOUND*/
            ENTRY = NIL
         THEN   /*SEND ERROR BACK TO TASK*/
            ECBPTR^.STAT := ERNOFILE
```

```
            ELSE      /*MOVE ATTRIBUTES*/
               BEGIN
               DIRBUFFADDR [ENTRY]. ATTRIBUTES := ECBPTR^. ATTRIBUTES;
               /*WRITE DIRECTORY ENTRY BACK TO DISC*/
               ECBPTR^. STATUS. ERROR := NOERROR;
               END;
            END;
         END
      ELSE      /*OTHER DEVICE, CAN'T SET ATTRIBUTES*/
         ECBPTR^. STATUS. ERROR := ERDEV;
      EVENTCOMPLETE (ECBPTR);
      END;
   END.
```

6.5.1.5 GETATTRIBUTES

```
PROCEDURE GETATTRSYNC (ECBPTR);

/*ENQUEUES THE REQUEST ECB FOR GETATTRIBUTES*/

BEGIN
LINK (ECBPTR, GETATTRIBUTESQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (GETATTRIBUTESQCB);
END.
```

```
PROCEDURE GETATTRIBUTES;

/*INPUT:  EVENT CONTROL BLOCK FROM GETATTRIBUTESQUEUE,
          CONTAINING:
          FILENAME
          ATTRIBUTESPTR: POINTER;
  OUTPUT: ATTRIBUTES OF REQUESTED FILE OR DEVICE
            OR ERROR:
               ERNOFILE*/

BEGIN
WHILE     /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, GETATTRIBUTESQHEAD, VAR STATUS);
   ECBPTR := GETATTRIBUTESQCB^.QHEAD.FIRST;
   DELINK (ECBPTR, GETATTRIBUTESQHEAD, ECBPTR^.ECBQLINK);
   FINDDEVICE (ECBPTR, VAR DDBPOINTER);
   IF    /*DISC OR OTHER DEVICE?*/
      DDBPTR^.ATTRIBUTES.DIRECTORY
   THEN     /*DISC, FIND FILENAME*/
      BEGIN
      FDBPTR := DDBPOINTER^.FDBQHEAD.FIRST;
      FOUND := FALSE;
      WHILE    /*WALK THROUGH THE FDB QUEUE FOR THIS DISC*/
         FDBPTR <> NIL AND NOT FOUND
      DO
         BEGIN
         IF    /*IS THIS THE FILE?*/
            FDBPTR^.FILENAME = ECBPTR^.FILENAME
         THEN     /*YES, RETURN ATTRIBUTES*/
            BEGIN
            ECBPTR^.ATTRIBUTESPTR^.ATTRIBUTES :=
                     FDBPTR^.ATTRIBUTES;
            ECBPTR^.STATUS.ERROR := NOERROR;
            FOUND := TRUE;
            END;
         FDBPTR := FDBPTR^.FDBQLINK.FLINK;
         END;
      IF    /*WERE THE ATTRIBUTES FOUND?*/
         NOT FOUND
      THEN     /*NO, GO TO THE DISC*/
         BEGIN
         GETBUFFER (DIRBUFFADDR, DIRBLOCKSIZE, DIRBUFFID, VAR STATUS);
         MAPTOBUFFER (DIRBUFFID, VAR STATUS);
         FINDECTORYENTRY (ECBPTR, DIRBUFFADDR, VAR DIRBLOCK,
                          VAR ENTRY);
         IF     /*CHECK FOR FILE NOT FOUND*/
            ENTRY = NIL
         THEN   /*SEND ERROR BACK TO TASK*/
            ECBPTR^.STAT := ERNOFILE
         ELSE     /*MOVE ATTRIBUTES*/
            ECBPTR^.ATTRIBUTESPTR^.ATTRIBUTES :=
               DIRBUFFADDR [ENTRY].ATTRIBUTES;
```

AD-A076 501 INTERNATIONAL COMPUTING CO BETHESDA MD F/G 9/2
VESSEL TRAFFIC SERVICES PROCESSING/DISPLAY SUBSYSTEM DETAILED S--ETC(U)
JUL 79 D A COHN , F T MICKEY DOT-CG-81-78-1833
UNCLASSIFIED USCG-D-66-79 NL

4 OF 4
AD A
076501

END
DATE
FILMED
3-81
DTIC

```
            ECBPTR^. STATUS. ERROR := NOERROR;
         END
      ELSE      /*OTHER DEVICE, GET DEVICE ATTRIBUTES*/
         BEGIN
         ECBPTR^. ATTRIBUTESPTR^. ATTRIBUTES := DDBPOINTER^. ATTRIBUTES;
         ECBPTR^. STATUS. ERROR := NOERROR;
         END;
      EVENTCOMPLETE (ECBPTR);
      END;
   END.
```

6.5.2 INPUT/OUTPUT OPERATIONS

6.5.2.1 OPEN

```
PROCEDURE OPEN (ECBPTR);

/*INPUT: POINTER TO CURRENT EVENT CONTROL BLOCK,
       CONTAINING:
          FILENAME:   FILENAMETYPE;
          CHANNELPTR: POINTER TO USER'S CHANNEL PARAMETER;
  OUTPUT:   CHANNEL OPENED TO THE SPECIFIED DEVICE
          OR ERROR:
             ERFILENAME
             ERNOCHANNEL    ALL CHANNELS IN USE
             ERNOFILE
             ERDEV
             ERDIR*/

BEGIN
CHANINDEX := 1;
WHILE    /*FIND FREE CHANNEL*/
   CHANNELTABLE [CHANINDEX] <> NIL
   AND CHANINDEX <= MAXCHANNEL
DO
   CHANINDEX := CHANINDEX + 1;
IF    /*FREE CHANNEL FOUND?*/
   CHANINDEX > MAXCHANNEL
THEN     /*YES, RETURN ERROR*/
   BEGIN
   ECBPTR^.STATUS.ERROR := ERNOCHANNEL;
   GOTO 10
   END;
FINDDEVICE (ECBPTR, VAR DDBPOINTER);
IF    /*CHECK FOR DEVICE OPENED ALREADY (NOT FOR DISC)*/
   NOT DDBPOINTER.ATTRIBUTES.DIRECTORY
   AND DDBPOINTER^.FDBQHEAD.FIRST <> NIL
THEN     /*RETURN ERROR*/
   BEGIN
   ECBPTR^.STATUS.ERROR := ERFILEOPEN;
   GOTO 10
   END;
IF    /*CHECK FOR DEVICE OR FILE*/
   DDBPOINTER^.ATTRIBUTES.BLOCKED
THEN     /*DISC OR TAPE, NOTIFY TASK*/
   BEGIN
   /*SET CHANNEL TO PREVENT CONFLICT*/
   ECBPTR^.CHANNELPTR^.CHANNEL := CHANINDEX;
   CHANNELTABLE [CHANINDEX] := DUMMYFDBPTR;
   LINK (ECBPTR, OPENQHEAD, ECBPTR^.ECBQLINK);
   EVENTCOMPLETE (OPENQHEAD);
   GOTO 10
   END;
```

```
      DDBPOINTER^. ATTRIBUTES. AVAILABLE := TRUE;
      INITDEVICE (DDBPOINTER, VAR STATUS);
      IF    /*DEVICE ERROR?*/
         STATUS. ERROR <> NOERROR
      THEN     /*YES, NOTIFY USER*/
         BEGIN
         /*RETURN ERDEV AND DEVICE STATUS REGISTERS*/
         ECBPTR^. STATUS. ERROR := STATUS;
         DDBPOINTER^. ATTRIBUTES. AVAILABLE := FALSE;
         GOTO 10
         END;
      /*GET FILE DESCRIPTOR BLOCK AND FILL IT IN*/
      FDBPTR := FREEFDBQCB. FDBQHEAD. FIRST;
      IF    /*NO FDB'S - NOT EXPECTED IN VTS*/
         FDBPTR = NIL
      THEN
         BEGIN
         ECBPTR^. STATUS. ERROR := ERNMR;
         GOTO 10
         END;
      DELINK (FDBPTR, FREEFDBQCB. QHEAD, FDBPTR^. FDBQLINK);
      CHANNELTABLE [CHANINDEX] := FDBPTR;
      FDBPTR^. DDBPOINTER := DDBPOINTER;
      LINK (FDBPTR, DDBPOINTER^. FDBQHEAD, FDBPTR^. FDBQLINK);
      FDBPTR^. FILENAME := DDBPOINTER^. FILENAME;
      FDBPTR^. ATTRIBUTES := DDBPOINTER^. ATTRIBUTES;
      /*INDICATE FUNCTION COMPLETED*/
      ECBPTR^. STATUS. ERROR := NOERROR;
10:      /*EXIT HERE WITH ERROR OR ASYNC OPEN*/
      END.
```

```
PROCEDURE OPENASYNC;

/*INPUT:  POINTER TO CURRENT EVENT CONTROL BLOCK,
          CONTAINING:
             FILENAME:    FILENAMETYPE;
             CHANNELPTR:  POINTER TO USER'S CHANNEL PARAMETER;
  OUTPUT:    CHANNEL OPENED TO THE SPECIFIED DEVICE
             OR ERROR:
                ERFILENAME
                ERCHANNEL
                ERNOFILE
                ERDEV
                ERDIR*/

BEGIN
WHILE     /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   /*WAIT FOR OPEN REQUEST*/
   WAITFORTOPENTRY (SUSPEND, OPENQCB, VAR STATUS);
   ECBPTR := OPENQHEAD.FIRST;
   DELINK (ECBPTR, OPENQHEAD, ECBPTR^.ECBQLINK);
   FINDDEVICE (ECBPTR, VAR DDBPOINTER);
   /*DEVICE POINTER WILL NOT BE NIL; CHECKED IN IOFUNCTIONS*/
   IF     /*DISC OR TAPE?*/
      DDBPOINTER.ATTRIBUTES.DIRECTORY
   THEN      /*DISC*/
      BEGIN
      GETTOPENTRY (DDBPOINTER^.FDBQHEAD, VAR FDBPTR);
      WHILE     /*CHECK FOR FILE OPEN*/
         FDBPTR <> NIL
      DO
         BEGIN
         IF
            ECBPTR^.FILENAME = FDBPTR^.FILENAME
         THEN      /*FILE OPEN, RETURN ERROR*/
            BEGIN
            ECBPTR^.STATUS.ERROR := ERFILEOPEN;
            GOTO 10
            END;
         GETNEXTENTRY (DDBPOINTER^.FDBQHEAD, VAR FDBPTR);
         END;
      GETBUFFER (DIRBUFFADDR, DIRBLOCKSIZE, DIRBUFFID, VAR STATUS);
      MAPTOBUFFER (DIRBUFFID, STATUS);
      FINDDIRECTORYENTRY (ECBPTR, DIRBUFADDR, VAR DIRBLOCK,
                            VAR ENTRY);
      IF     /*CHECK FOR FILE NOT FOUND*/
         ENTRY = NIL
      THEN      /*RETURN ERROR*/
         BEGIN
         ECBPTR^.STATUS.ERROR := ERNOFILE;
         GOTO 5
         END;
      /*OPEN FILE*/
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
      /*GET FILE DESCRIPTOR BLOCK AND FILL IT IN*/
      FDBPTR := FREEFDBQCB.FDBQHEAD.FIRST;
      IF     /*NO FDB'S - NOT EXPECTED IN VTS*/
         FDBPTR = NIL
      THEN
         BEGIN
         ECBPTR^.STATUS.ERROR := ERNMR;
         GOTO 5
         END;
      DELINK (FDBPTR, FREEFDBQCB.QHEAD, FDBPTR^.FDBQLINK);
      FDBPTR^.DISCDIRECTORYENTRY := DIRBLOCK [ENTRY];
      FDBPTR^.CHANNEL := ECBPTR^.CHANNELPTR^.CHANNEL;
      CHANNELTABLE [FDBPTR^.CHANNEL] := FDBPTR;
      FDBPTR^.DDBPOINTER := DDBPOINTER;
      LINK (FDBPTR, DDBPOINTER^.FDBQHEAD, FDBPTR^.FDBQLINK);
      /*SET UP SEQUENTIAL I/O INFORMATION*/
      FDBPTR^.BUFFER := NIL;
      FDBPTR^.SEQPOSITION.BLOCK := NIL;
      FDBPTR^.SEQPOSITION.WORD := 0;
      FDBPTR^.TCBPTR := ECBPTR^.TCBPTR;
      ECBPTR^.STATUS.ERROR := NOERROR;
5:    /*EXIT WITH ERROR ON DISC FILE OPEN*/
      RELEASEBUFFER (SUSPEND, DIRBUFFID);
      END
   ELSE      /*TAPE*/
      DDBPOINTER^.ATTRIBUTES.AVAILABLE := TRUE;
      TAPEFILE := CONVERTTODECIMAL (ECBPTR^.FILENAME.NAME);
      IF     /*CHECK TO SEE THAT NAME WAS NUMERIC*/
         TAPEFILE = NIL
      THEN      /*RETURN ERROR*/
         BEGIN
         ECBPTR^.STATUS.ERROR := ERFILENAME;
         GOTO 10
         END;
      /*POSITION TAPE TO FILE TAPEFILE; IF A DOUBLE
        END OF FILE IS ENCOUNTERED BEFORE TAPEFILE
        IS FOUND, STATUS.ERROR CONTAINS EREOF*/

      IF     /*CHECK FOR DEVICE ERROR*/
         STATUS.ERROR = ERDEV
      THEN      /*MARK DEVICE UNAVAILABLE*/
         DDBPOINTER.ATTRIBUTES.AVAILABLE := FALSE;
      IF     /*CHECK FOR EOF*/
         STATUS.ERROR <> NOERROR
      THEN      /*RETURN EOF OR DEVICE ERROR*/
         BEGIN
         ECBPTR^.STATUS.ERROR := STATUS.ERROR;
         GOTO 10
         END;
      /*GET FILE DESCRIPTOR BLOCK AND FILL IT IN*/
      FDBPTR := FREEFDBQCB.FDBQHEAD.FIRST;
      IF     /*NO FDB'S - NOT EXPECTED IN VTS*/
         FDBPTR = NIL
      THEN
         BEGIN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
            ECBPTR^. STATUS. ERROR := ERNMR;
            GOTO 10
            END;
         DELINK (FDBPTR, FREEFDBQCB. QHEAD, FDBPTR^. FDBQLINK);
         FDBPTR^. DISCDIRECTORYENTRY. FILENAME := ECBPTR^. FILENAME;
         FDBPTR^. DISCDIRECTORYENTRY. ATTRIBUTES :=
            DDBPOINTER^. ATTRIBUTES;
         FDBPTR^. CHANNEL := ECBPTR^. CHANNELPTR^. CHANNEL;
         CHANNELTABLE [FDBPTR^. CHANNEL] := FDBPTR;
         FDBPTR^. DDBPOINTER := DDBPOINTER;
         LINK (FDBPTR, DDBPOINTER^. FDBQHEAD, FDBPTR^. FDBQLINK);
         FDBPTR^. TCBPTR := ECBPTR^. TCBPTR;
         ECBPTR^. STATUS. ERROR := NOERROR;
         END;
10:   /*LABEL FOR ERROR EXIT*/
      EVENTCOMPLETE (ECBPTR);
      IF    /*IF ERROR, UNDO CHANNEL SET BY OPEN*/
         ECBPTR^. STATUS. ERROR <> NOERROR
      THEN
         BEGIN
         CHANNELTABLE [ECBPTR^. CHANNELPTR^. CHANNEL] := NIL;
         ECBPTR^. CHANNELPTR^. CHANNEL := NIL;
         END;
      END;
   END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.5.2.2 CLOSE

```
PROCEDURE CLOSE (ECBPTR);

/*INPUT:   ECB CONTAINING:
             CHANNEL: INTEGER;
  OUTPUT:  FILE CLOSED OR ERROR:
             ERNOTOPEN*/

BEGIN
FDBPTR := CHANNELTABLE [ECBPTR^.CHANNEL];
IF      /*CHECK FOR FILE NOT OPEN*/
   FDBPTR := NIL
THEN       /*RETURN ERROR*/
   ECBPTR^.STATUS.ERROR := ERNOTOPEN
ELSE       /*FILE OPEN, NOW CLOSE IT*/
   BEGIN
   IF      /*BLOCK-STRUCTURED DEVICE?*/
      FDBPTR^.ATTRIBUTES.BLOCKED
   THEN       /*YES, PROCESS ASYNCHRONOUSLY*/
      BEGIN
      LINK (ECBPTR, CLOSEQHEAD, ECBPTR^.ECBQLINK);
      EVENTCOMPLETE (CLOSEQCB);
      END
   ELSE       /*DEVICE CAN BE CLOSED SYNCHRONOUSLY*/
      BEGIN
      /*RELEASE CHANNEL ASSIGNMENT*/
      CHANNELTABLE [ECBPTR^.CHANNEL] := NIL;
      /*REMOVE FDB FROM DDB QUEUE OF FDB'S*/
      DELINK (FDBPTR, FDBPTR^.DDBPOINTER^.FDBQHEAD, FDBPTR^.FDBQLINK);
      /*RETURN FDB TO LIST OF FREE FDB'S*/
      LINK (FDBPTR, FREEFDBQHEAD, FDBPTR^.FDBQLINK);
      ECBPTR^.STATUS.ERROR := NOERROR;
      END;
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE CLOSEASYNC;

/*INPUT: POINTER TO FILE CONTROL BLOCK
  OUTPUT:   CHANNEL CLOSED*/

BEGIN
WHILE    /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, CLOSEQCB, VAR STATUS);
   ECBPTR := CLOSEQHEAD.FIRST;
   IF    /*IS THE DEVICE DISC?*/
      FDBPTR^.ATTRIBUTES.DIRECTORY
   THEN     /*IS THERE STILL DATA IN THE SEQUENTIAL BUFFER?*/
      BEGIN
      IF    /*IS THERE A BUFFER?*/
         FDBPTR^.BUFFER <> NIL
      THEN     /*YES, CHECK FOR DATA*/
         BEGIN
         IF
            FDBPTR^.SEQPOSITION.WORDS <> 0
         THEN     /*PARTIAL BUFFER EXISTS*/
            BEGIN
            /*WRITE BUFFER TO DISC AT SEQPOSITION.BLOCK*/
            END;
         /*ANY DATA HAS BEEN WRITTEN, RETURN BUFFER*/
         FREEBUFFERLIST [BUFFER^.BUFFERID].FDBPTR := NIL;
         IF    /*SEGMENTED FILE WITH SPB BUFFER?*/
            FDBPTR^.CURRENTSPBBUFFER <> NIL
         THEN     /*YES, RETURN IT*/
            FREEBUFFERLIST [CURRENTSPBBUFFER^.BUFFERID].FDBPTR := NIL;
         END;
      /*NO BUFFER OR BUFFER RELEASED*/
      END
   ELSE     /*MAGNETIC TAPE*/
      BEGIN
      IF    /*PARTIAL BUFFER PRESENT?*/
         FDBPTR^.BUFFER <> NIL
         AND FDBPTR^.SEQPOSITION.WORDS <> 0
      THEN     /*PAD IT AND WRITE IT*/
         BEGIN
         /*FILL BUFFER WITH ZEROS FROM SEQPOSITION.WORDS
           TO DDBPOINTER^.BLOCKSIZE*/
         /*WRITE BUFFER TO TAPE*/
         END;
      /*RELEASE TAPE BUFFER*/
      FREEBUFFERLIST [FDBPTR^.BUFFER^.BUFFERID] := NIL;
      IF    /*WRITING A FILE?*/
         FDBPTR^.ATTRIBUTES.WRITEPROTECT
      THEN     /*YES, WRITE DOUBLE END OF FILE*/
         /*REQUEST DOUBLE END OF FILE FROM TAPE DRIVER*/;
      END
   /*RELEASE CHANNEL ASSIGNMENT*/
   CHANNELTABLE [ECBPTR^.CHANNEL] := NIL;
```

```
   /*REMOVE FDB FROM DDB QUEUE OF FDB'S*/
   DELINK (FDBPTR, FDBPTR^.DDBPOINTER^.FDBQHEAD, FDBPTR^.FDBQLINK);
   /*RETURN FDB TO LIST OF FREE FDB'S*/
   LINK (FDBPTR, FREEFDBQHEAD, FDBPTR^.FDBQLINK);
   ECBPTR^.STATUS.ERROR := NOERROR;
   EVENTCOMPLETE (ECBPTR);
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

6.5.2.3 IORESET

```
PROCEDURE IORESET (ECBPTR);

/*INPUT:   REQUEST ECB FROM IOFUNCTIONS CONTAINING:
               TASKID: INTEGER;
  CALLS IORESETASYNC*/

BEGIN
LINK (ECBPTR, IORESETQHEAD, ECBPTR^.ECBQLINK);
EVENTCOMPLETE (IORESETQHCB);
END.
```

```
PROCEDURE GETTCBPOINTER(TASKID, TCBPTR)

  /*  INPUT:     TASKID -- ID OF A TASK
      OUTPUT:    TCBPTR -- EITHER A POINTER TO THE TCB OF
                 THE TASK WHOSE ID WAS IN TASKID, OR NIL IF TCB
                 NOT FOUND */

  BEGIN
     /* FIRST SEE IF TASK IS ON READY-Q */
     TCB := RDYQHD^.FIRST;
     WHILE TCB <> NIL AND TCB^.TASKID <> TASKID
        DO TCB := TCB^.TCBQLINK^.FLINK;
     IF TCB = NIL
        THEN   /* TASK IS NOT ON READY-Q */
        BEGIN   /* SO SEARCH SUSPEND-Q */
           TCB := SUSPENDQHD^.FIRST;
           WHILE TCB <> NIL AND TCB^.TASKID <> TASKID
              DO TCB := TCB^.TCBQLINK^.FLINK;
        END;
     TCBPTR := TCB; /* TCB IS EITHER NIL OR DESIRED [illegible] */
  END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
PROCEDURE IORESETASYNC;

/*INPUT:   REQUEST ECB FROM IORESET CONTAINING:
              TASKID: INTEGER;
  OUTPUT: ALL FILES FOR CALLING TASK CLOSED*/

BEGIN
WHILE     /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITFORTOPENTRY (SUSPEND, IORESETQCB, VAR STATUS);
   ECBPTR := IORESETQHEAD.FIRST;
   DELINK (ECBPTR, IORESETQHEAD, ECBPTR^.ECBQLINK);
   GETTCBPOINTER (ECBPTR^.TASKID, VAR TCBPTR);
   IF     /*INVALID TASK ID?*/
      TCBPTR = NIL
   THEN      /*YES, RETURN ERROR*/
      ECBPTR^.STATUS.ERROR := ERTASKID
   ELSE      /*FIND THIS TASK'S OPEN CHANNELS*/
      BEGIN
      FOR        /*FIND DEVICES OPENED BY THIS TASK*/
         CHAN := 1 TO MAXCHANNEL
      DO
         BEGIN
         FDBPTR := CHANNELTABLE [CHAN].FDBPTR;
         IF     /*CLOSE THIS CHANNEL?*/
            FDBPTR <> NIL
            AND FDBPTR^.TCBPTR = TCBPTR
         THEN      /*YES*/
            CLOSE (SUSPEND, CHAN, VAR STATUS);
         END;
      ECBPTR^.STATUS.ERROR := NOERROR;
      END;
   EVENTCOMPLETE (ECBPTR);
   END;
END.
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDQ

6.5.2.4 READ AND WRITE FUNCTIONS

```
PROGRAM READWRITE (ECBPTR);

/*INPUT: ECB WITH PARAMETERS AS SPECIFIED IN THE CALLING
         SEQUENCES FOR READR, READSQ, READLN, WRITER, WRITESQ,
         AND WRITELN IN                         SECTION 5.5.2.4
  OUTPUT: THE REQUEST ECB PASSED TO READWRITEASYNC OR ERROR:
            ERCHANNEL        INVALID CHANNEL NUMBER
            ERNOTOPEN        FILE OR DEVICE NOT OPEN
            ERREADPROTECT
            ERWRITEPROTECT
            ERNOTAVAIL       DEVICE NOT AVAILABLE*/

BEGIN
/*PERFORM INITIAL ERROR CHECKS*/
FDBPTR := CHANNELTABLE [ECBPTR^.CHANNEL];
IF    /*FILE NOT OPEN*/
   FDBPTR = NIL
THEN
   BEGIN
   ECBPTR^.STATUS.ERROR := ERFILENOTOPEN;
   GOTO 10
   END;
IF    /*IS THE DEVICE AVAILABLE?*/
   NOT FDBPTR^.DDBPTR^.ATTRIBUTES.AVAILABLE
THEN     /*NO, RETURN ERROR - USER MUST CLOSE CHANNEL*/
   BEGIN
   ECBPTR^.STATUS.ERROR := ERNOTAVAIL;
   GOTO 10
   END;
IF    /*PROTECTION VIOLATION*/
   ECBPTR^.SUBFUNC = READR
   OR ECBPTR^.SUBFUNC = READSQ
   OR ECBPTR^.SUBFUNC = READLN
THEN
   BEGIN
   IF    /*CHECK READ PROTECT*/
      FDBPTR^.ATTRIBUTES.READPROTECT
   THEN
      BEGIN
      ECBPTR^.STATUS.ERROR := ERREADPROTECT;
      GOTO 10
      END;
   END
ELSE     /*MUST BE A WRITE*/
   BEGIN
   IF    /*CHECK WRITE PROTECT*/
      FDBPTR^.ATTRIBUTES.WRITEPROTECT
   THEN
      BEGIN
      ECBPTR^.STATUS.ERROR := ERWRITEPROTECT;
      GOTO 10
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
        END;
      END;
    /*MOVE USER DATA AREA ADDRESS AND MAPS INTO FDB*/
    /*INCREMENT IOINPROGRESS FLAG IN USER BCB, AND
      IF NOT OWNED BY THIS USER, IN OWNER'S BCB*/
    /*PASS REQUEST TO READWRITEASYNC*/
    LINK (ECBPTR, READWRITEQHEAD, ECBPTR^.QLINK);
    EVENTCOMPLETE (READWRITEQCB);
10: /*EXIT WITH NOTCOMPLETE OR ERROR*/
    END.
```

```
PROGRAM READWRITEASYNC;

/*INPUT: ECB WITH PARAMETERS AS SPECIFIED IN THE CALLING
        SEQUENCES FOR READR, READSQ, READLN, WRITER, WRITESQ,
        AND WRITELN IN SECTION 5.5.2.4
  OUTPUT: COMPLETION OF THE REQUESTED FUNCTION OR ERROR:
           EREOF              END OF FILE
                                 (VALUE1 HAS WORD COUNT FOR ACTUAL
                                 TRANSFER IN SEQUENTIAL READS)
           ERFILEADDR         FILE ADDRESS BEYOND FILE
           ERDIR              DIRECTORY ERROR
           ERDEV              UNCORRECTABLE DEVICE ERROR
                                 (VALUE1 AND VALUE2 CONTAIN THE
                                 RETURNED STATUS REGISTERS FOR
                                 THE DEVICE)
           ERNOSPACE          OUT OF ROOM ON BLOCKED DEVICE
           ERNORANDOM         INVALID REQUEST FOR DEVICE OR FILE*/

BEGIN
/*SET UP SYSTEM BUFFER TABLE FOR SEQUENTIAL I/O*/
BUFFID := 0;
WHILE     /*# OF BUFFERS DEPENDS ON PROCESSOR*/
   BUFINDEX < BUFFERSFORTHISPROCESSOR
DO
   BEGIN
   GETBUFFER (BUFFADDR, IOBUFSIZE, PRIVATE, VAR BUFFID, VAR STATUS);
   FREEBUFFERTABLE [BUFFID].FDBPTR := NIL;
   END;
WAITFORTOPENTRY (EFLAG1, READWRITEQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG2, FURTHERACTIONQCB, VAR STATUS);
WAITFORTOPENTRY (EFLAG3, COMPLETEQCB, VAR STATUS);
WHILE     /*INFINITE LOOP*/
   TRUE
DO
   BEGIN
   WAITANY (EFLAG1 + EFLAG2 + EFLAG3, VAR EFFOUND);
   /*GET STATUS RETURN AND RE-ISSUE QUEUE REQUEST*/
   GETSTATUS (EFFOUND, VAR STATUS);
   CASE EFFOUND OF
      EFLAG1:
         BEGIN
         ECBPTR := STATUS.VALUE1;     /*RETURNED ECB POINTER*/
         DELINK (ECBPTR, READWRITEQHEAD, ECBPTR^.QLINK);
         WAITFORTOPENTRY (EFLAG1, READWRITEQCB, VAR STATUS);
         END;
      EFLAG2:
         BEGIN
         FDBPTR := STATUS.VALUE1;     /*RETURNED FDB POINTER*/
         ECBPTR := FDBPTR^.ECBPTR;   /*ASSOCIATED ECB*/
         DELINK (FDBPTR, FURTHERACTIONQHEAD, FDBPTR^.QLINK);
         WAITFORTOPENTRY (EFLAG2, FURTHERACTIONQCB, VAR STATUS);
         END;
      EFLAG3:
         BEGIN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
          FDBPTR := STATUS.VALUE1;     /*RETURNED FDB POINTER*/
          ECBPTR := FDBPTR^.ECBPTR;    /*ASSOCIATED ECB*/
          DELINK (FDBPTR, COMPLETEQHEAD, FDBPTR^.QLINK);
          WAITFORTOPENTRY (EFLAG3, COMPLETEQCB, VAR STATUS);
          END;
       END;      /*END CASE*/


    IF      /*CHECK FOR ERROR*/
       ECBPTR^.STATUS.ERROR < 0
    THEN      /*CANCEL I/O REQUEST*/
       BEGIN
       IF      /*CHECK FOR SEVERE ERROR*/
          ECBPTR^.STATUS.ERROR < SEVEREERROR
       THEN      /*MARK DEVICE UNAVAILABLE*/
          BEGIN
          DDBPOINTER := CHANNELTABLE [ECBPTR^.CHANNEL]^.DDBPOINTER;
          DDBPOINTER^.ATTRIBUTES.AVAILABLE := FALSE;
          END;
       GOTO 10      /*SIGNAL EVENT COMPLETE*/
       END;


    /*SELECT STAGE OF PROCESSING TO BE PERFORMED*/
    CASE EFFOUND OF
       EFLAG1:      /*INITIATION STAGE*/
          BEGIN
          FDBPTR := CHANNELTABLE [ECBPTR^.CHANNEL];
          CASE ECBPTR^.SUBFUNC OF
             READR, WRITER:
                BEGIN
                IF      /*CHECK FOR REQUEST WITHIN FILE*/
                   FDBPTR^.ATTRIBUTES.CONTIGUOUS
                   AND FDBPTR^.FILESIZE <
                      ECBPTR^.FILEADDR + ECBPTR^.WORDS
                THEN      /*ERROR, RETURN TO USER*/
                   BEGIN
                   ECBPTR^.STATUS.ERROR := ERFILEADDR;
                   GOTO 10
                   END;
                /*CHECK ALIGNMENT OF REQUEST*/
                IF
                   ECBPTR^.FILEADDRESS
                      MOD FDBPTR^.DDBPOINTER^.BLOCKSIZE = 0
                   AND ECBPTR^.WORDS MOD FDBPTR^.DDBPTR^.BLOCKSIZE = 0
                THEN      /*CAN PERFORM IN USER BUFFER*/
                   BEGIN
                   /*INDICATE USER DATA SPEC FOR TRANSFER*/
                   FDBPTR^.BUFFER := NIL;
                   CASE ECBPTR^.SUBFUNC OF
                      READR:
                         FDBPTR^.DEVICEDRIVERREQUEST := READBLOCKS;
                      WRITER:
                         FDBPTR^.DEVICEDRIVERREQUEST := WRITEBLOCKS;
                      END;
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
          /*ASK FOR FDB BACK ON "OPERATION COMPLETE" Q*/
          FDBPTR^.DRIVERRETURNQHEAD := COMPLETEQHEAD;
          LINK (FDBPTR, BLOCKEDDEVICEDRIVERQHEAD,
               FDBPTR^.QLINK);
          /*NOTIFY DEVICE DRIVER THAT REQUEST HAS BEEN QUEUED*/
          END
       ELSE      /*NEED A BUFFER*/
          BEGIN
          /*CALCULATE USER DATA SECTORS NEEDED*/
          ENDADDR := ECBPTR^.FILEADDRESS + ECBPTR^.WORDS;
          /*ASSUMES 'DIV' PERFORMED BEFORE '-' */
          BUFFWORDS := (ENDADDR + DISCBLOCKSIZE - 1) DIV
                          DISCBLOCKSIZE
             - FILEADDRESS DIV DISCBLOCKSIZE;
          GETBUFFER (IOBUFFADDR, BUFFWORDS, VAR IOBUFFID,
             VAR STATUS);
          MAPTOBUFFER (IOBUFFID, VAR STATUS);
          /*POINT FDBPTR^.BUFFER TO THIS BUFFER'S BCB*/
          CASE ECBPTR^.SUBFUNC OF
             READR:   FDBPTR^.DRIVERRETURNQCB := COMPLETEQCB;
             WRITER:  FDBPTR^.DRIVERRETURNQCB :=
                FURTHERACTIONQCB;
             END;
          FDBPTR^.DEVICEDRIVERREQUEST := READBLOCKS;
          LINK (FDBPTR, BLOCKEDDEVICEDRIBERQHEAD,
             FDBPTR^.FDBQLINK);
          END;
       END;      /*END READR, WRITER INITIATION*/
    READSQ, WRITESQ, READLN, WRITELN:
       IF     /*DISC OR TAPE?*/
          FDBPTR^.ATTRIBUTES.BLOCKED
       THEN      /*YES, CHECK FOR BUFFER NEEDED*/
          BEGIN
          IF
             FDBPTR^.BUFFER := NIL
          THEN      /*ALLOCATE READ/WRITE BUFFER*/
             BEGIN
             FOUND := FALSE;
             BUFFID := 1;
             WHILE     /*LOOK THROUGH TABLE
                BUFFID < BUFFERSFORTHISPROCESSOR
                AND NOT FOUND
             DO
                BEGIN
                IF
                   FREEBUFFERTABLE [BUFFID].FDBPTR := NIL
                THEN
                   BEGIN
                   FOUND := TRUE;
                   FDBPTR^.BUFFER :=
                      /*BCB POINTER FOR BUFFER*/;
                   END;
                BUFFID := BUFFID + 1;
                END;
             IF     /*OUT OF BUFFERS?*/
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
        NOT FOUND
     THEN    /*NOT EXPECTED - SEE TEXT*/
        BEGIN
        ECBPTR^.STATUS.ERROR := ERNOBUFFER;
        GOTO 10
        END;
     END;
IF     /*FIRST TAPE REQUEST?*/
   NOT FDBPTR^.ATTRIBUTES.DIRECTORY
   AND FDBPTR^.DEVICEDRIVERREQUEST = NIL
THEN      /*YES, SET PROTECTION*/
   CASE ECBPTR^.SUBFUNC OF
      READSQ, READLN:
         FDBPTR^.ATTRIBUTES.WRITEPROTECT := TRUE;
      WRITESQ, WRITELN:
         FDBPTR^.ATTRIBUTES.READPROTECT := TRUE;
      END;
/*CALCULATE NEW POSITION OF WORD
  SEQUENCE POINTER*/
NEWPOSITION:= FDBPTR^.SEQPOSITION.WORD
   +ECBPTR^.WORDS;
IF     /*WILL THE BUFFER BE SUFFICIENT?*/
   NEWPOSITION < FDBPTR^.DDBPOINTER^.BLOCKSIZE
THEN      /*YES*/
   CASE ECBPTR^.SUBFUNC OF
      READSQ, READLN:
         BEGIN
         /*MOVE WORDS FROM BUFFER*/
         FDBPTR^.SEQPOSITION.WORD :=
            NEWPOSITION;
         END;
      WRITESQ, READLN:
         BEGIN
         /*MOVE WORDS TO BUFFER*/
         FDBPTR^.SEQPOSITION.WORD :=
            NEWPOSITION;
         END;
      END;  /*OF CASE*/
ELSE      /*NO, READ OR WRITE BUFFER*/
   CASE ECBPTR^.SUBFUNC OF
      READSQ, READLN:
         BEGIN
         /*MOVE WORDS LEFT IN THIS BUFFER*/
         FDBPTR^.SEQPOSITION.WORD := 0;
         ECBPTR^.WORDS := NEWPOSITION
            - FDBPTR^.DDBPOINTER^.BLOCKSIZE;
         FDBPTR^.DRIVERRETURNQCB :=
            FURTHERACTIONQCB;
         FDBPTR^.DEVICERIVERREQUEST :=
            READBLOCKS;
         /*ASK DRIVER TO READ THE NEXT BLOCK*/
         END;
      WRITESQ, READLN:
         BEGIN
         /*MOVE WORDS LEFT IN THIS BUFFER*/
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
                                   FDBPTR^. SEQPOSITION. WORD := 0;
                                   ECBPTR^. WORDS := NEWPOSITION
                                      - FDBPTR^. DDBPOINTER^. BLOCKSIZE;
                                   FDBPTR^. DRIVERRETURNQCB :=
                                      FURTHERACTIONQCB;
                                   FDBPTR^. DEVICERIVERREQUEST :=
                                      WRITEBLOCKS;
                                   /*ASK DRIVER TO WRITE THIS BLOCK*/
                                   END;
                                END;  /*OF CASE*/
                           END      /*OF SEQUENTIAL OP'NS FOR DISC AND TAPE*/
                        ELSE     /*DEVICE, DRIVER DOES READ OR WRITE*/
                           BEGIN
                           /*MOVE USER BUFFER SPEC TO FDB*/
                           FDBPTR^. DRIVERRETURNQCB := COMPLETEQCB;
                           FDBPTR^. DEVICEDRIVERREQUEST := ECBPTR^. SUBFUNC;
                           LINK (FDBPTR, SEQUENTIALDEVICEDRIVERQHEAD,
                              FDBPTR^. FDBQLINK);
                           /*ASK DEVICE DRIVER TO TRANSFER CHARACTERS*/
                           END;      /*OF SEQ. DEVICE PROCESSING*/
                        END;      /*OF CASE FOR INITIATION OF I/O*/
                  END;      /*OF INITIATION STAGE*/


               EFLAG2:      /*FURTHER ACTION STAGE*/
                  BEGIN
                  CASE ECBPTR^. SUBFUNC OF
                     READR:
                        BEGIN
                        /*MOVE DATA FROM FDBPTR^. BUFFER TO USER'S BUFFER*/
                        RELEASEBUFFER (SUSPEND, FDBPTR^. BUFFER^. BUFFERID,
                           VAR STATUS);
                        ECBPTR^. STATUS. ERROR := NOERROR;
                        GOTO 10
                        END;
                     WRITER:
                        BEGIN
                        /*MOVE USER'S DATA TO FDBPTR^. BUFFER*/
                        FDBPTR^. DRIVERRETURNQCB := COMPLETEQCB;
                        FDBPTR^. DEVICEDRIVERREQUEST := WRITEBLOCKS;
                        /*ASK DRIVER TO WRITE BLOCKS BACK TO DISK*/
                        END;
                     READSQ, READLN, WRITESQ, WRITELN:
                        BEGIN
                        IF    /*NEED DATA SEGMENT IF SEGMENTED*/
                           FDBPTR^. ATTRIBUTES. SEGMENTED
                        THEN
                           BEGIN
                           GETDATASEGMENT (FDBPTR, VAR STATUS);
                           IF    /*ERROR?*/
                              STATUS. ERROR <> NOERROR
                           THEN     /*RETURN TO USER*/
                              BEGIN
                              ECBPTR^. STATUS := STATUS;
                              GOTO 10
```

```
      END;
    END;
  /*CALCULATE NEW POSITION OF WORD
    SEQUENCE POINTER*/
  NEWPOSITION:= FDBPTR^.SEQPOSITION.WORD
    +ECBPTR^.WORDS;
  IF     /*WILL THE BUFFER BE SUFFICIENT?*/
    NEWPOSITION < FDBPTR^.DDBPOINTER^.BLOCKSIZE
  THEN      /*YES*/
    CASE ECBPTR^.SUBFUNC OF
      READSQ, READLN:
        BEGIN
        /*MOVE WORDS FROM BUFFER*/
        FDBPTR^.SEQPOSITION.WORD :=
          NEWPOSITION;
        END;
      WRITESQ, READLN:
        BEGIN
        /*MOVE WORDS TO BUFFER*/
        FDBPTR^.SEQPOSITION.WORD :=
          NEWPOSITION;
        END;
      END;  /*OF CASE*/
  ELSE      /*NO, READ OR WRITE BUFFER*/
    CASE ECBPTR^.SUBFUNC OF
      READSQ, READLN:
        BEGIN
        /*MOVE WORDS LEFT IN THIS BUFFER*/
        FDBPTR^.SEQPOSITION.WORD := 0;
        ECBPTR^.WORDS := NEWPOSITION
          - FDBPTR^.DDBPOINTER^.BLOCKSIZE;
        FDBPTR^.DRIVERRETURNQCB :=
          FURTHERACTIONQCB;
        FDBPTR^.DEVICERIVERREQUEST :=
          READBLOCKS;
        /*ASK DRIVER TO READ THE NEXT BLOCK*/
        END;
      WRITESQ, READLN:
        BEGIN
        /*MOVE WORDS LEFT IN THIS BUFFER*/
        FDBPTR^.SEQPOSITION.WORD := 0;
        ECBPTR^.WORDS := NEWPOSITION
          - FDBPTR^.DDBPOINTER^.BLOCKSIZE;
        FDBPTR^.DRIVERRETURNQCB :=
          FURTHERACTIONQCB;
        FDBPTR^.DEVICERIVERREQUEST :=
          WRITEBLOCKS;
        /*ASK DRIVER TO WRITE THIS BLOCK*/
        END;
      END;  /*OF CASE*/
  END;      /*OF SEQUENTIAL OP'NS FOR DISC AND TAPE*/
  ELSE      /*DEVICE, DRIVER DOES READ OR WRITE*/
    BEGIN
    /*MOVE USER BUFFER SPEC TO FDB*/
    FDBPTR^.DRIVERRETURNQCB := COMPLETEQCB;
```

```
                    FDBPTR^.DEVICEDRIVERREQUEST := ECBPTR^.SUBFUNC;
                    LINK (FDBPTR, SEQUENTIALDEVICEDRIVERQHEAD,
                       FDBPTR^.FDBQLINK);
                    /*ASK DEVICE DRIVER TO TRANSFER CHARACTERS*/
                    END;      /*OF SEQ. DEVICE PROCESSING*/
           END;     /*OF FURTHER ACTION STAGE*/


        EFLAG3:     /*COMPLETION STAGE*/
           BEGIN
           IF    /*DISC RANDOM WRITE?*/
              ECBPTR^.SUBFUNC = WRITER
           THEN     /*YES, RELEASE BUFFER*/
              RELEASEBUFFER (SUSPEND, FDBPTR^.BUFFER^.BUFFERID,
                 VAR STATUS);
           ECBPTR^.STATUS.ERROR := NOERROR;
10:        EVENTCOMPLETE (ECBPTR);
           /*DECREMENT IOINPROGRESS FLAG FOR USER BCB AND,
             IF NOT OWNED BY THIS USER, IN THE OWNER'S
             BCB*/
           END;
        END;     /*END OF STAGE CASE*/
     END;     /*INFINITE LOOP*/
END.
```

6.6 RECOVERY AND RECONFIGURATION

6.6.1 DATA STRUCTURES AND VARIABLE DECLARATIONS

```
TYPE
   TASKID =
      RECORD
         TASKID, PROCESSORNUMBER:  INTEGER
      END;
   ERRORTYPE = (BADDATAPRODUCED, BADDATARECEIVED,
                MSGNOTANSWERED, UNABLETOINSTALL,
                UNABLETOSCHEDULE);
   FCB =    /* FAILURE CONTROL BLOCK */
      RECORD
         FLINK, BLINK:  ^FCB;
         SENDERID, DESTNAME:  TASKID;
         ERRORCODE:  ERRORTYPE;
         FAILURENUMBER, RETRYCOUNT, RECOVERYCOUNT:  INTEGER;
         TIME1STREPORT, TIMERECENTREPORT:  TIME;
      END;
VAR
   FCBQUEUEHEAD:  ^FCB; /* POINTER TO THE FIRST FCB IN THE QUEUE */
   FCBQTAIL:  ^FCB; /* POINTER TO THE LAST FCB IN THE QUEUE */
   FCBAVAILQHEAD:  ^FCB; /* POINTER TO THE FIRST FCB IN THE
                              QUEUE OF AVAILABLE FCB'S */
   MERCATIVESTATUS:  (ACTIVE, INACTIVE); /* DESCRIBES THE ABILITY OF
                                            THE MERC TO SEND CONFIG-
                                            URATION MESSAGES */
   NOPREVIOUSCONFIGMSG:  BOOLEAN; /* TRUE MEANS THAT THE MESSAGE ONE
                                     WAS NOT AVAILABLE */
   CURRENTCONFIG:  INTEGER; /* DESCRIBES CURRENT CONFIGURATION */
   ROUTERTABLE:  ARRAY [1..MAXNOTASKS] OF RECORD
      TASKNAME:  TASKNAMETYPE;
      TASKID, PROCESSOR:  INTEGER
   END;
   CONFIGTABLE:  ARRAY [1..MAXNOCONFIGURATIONS] OF
                    ARRAY [1..MAXNOTASK] OF RECORD
      TASKNAME:  TASKNAMETYPE; PROCESSOR:  INTEGER
   END;
   VIRTUALTODISPLAYPCMAP:
      ARRAY [1..MAXNODSPLAYPROCESSORS] OF INTEGER;
      /* MAP OF VIRTUAL STATIONS TO DISPLAY PROCESSORS;
      ELEMENTS ARE THE INTEGER IDENTIFIER FOR THE PROCESSOR */
   PCSTATUS:  ARRAY [1..MAXNOPROCESSORS] OF (UP, DOWN);
      /* NOTE:  ELEMENTS 1-3 OF PCSTATUS CORRESPOND TO THE THREE MAIN
            PROCESSORS */
         /* WE ASSUME THERE IS A FUNCTION "CURRENTTIME" WHICH RETURNS
         THE CURRENT TIME OF DAY. */
```

6.6.2 PROCEDURE DELARATIONS

6.6.2.1 FAILUREDETECTED

```
PROCEDURE FAILUREDETECTED (SENDERID, DESTNAME, ERRORCODE, FAILURENUMBER);
/*THIS PROCEDURE IS CALLED BY AN APPLICATION TASK
  WHENEVER AN ERROR IS DETECTED */
VAR CURRENTFCB, NEWFCB: ^FCB;
BEGIN
IF
   FCBAVAILQHEAD = NIL
THEN
   /* SINCE THERE IS NO AVAILABLE FCB, WE WILL TAKE
   THE LAST FCB FROM THE FCB QUEUE */
   BEGIN
   NEWFCB:= FCBQTAIL;
   FCBQTAIL:= FCBQTAIL^.BLINK;
   FCBQTAIL^.FLINK:= NIL
   END

ELSE
   /* GET AN FCB FROM THE QUEUE OF AVAILABLE FCB'S */
   BEGIN
   NEWFCB:= FCBAVAILQHEAD;
   FCBAVAILQHEAD:=
      FCBAVAILQHEAD^.FLINK;
   FCBAVAILQHEAD^.BLINK:= NIL
   END;
/* FILL IN THE INFORMATION IN THE FCB */
NEWFCB^.SENDERID:= SENDERID;
NEWFCB^.DESTNAME:= DESTNAME;
NEWFCB^.ERRORCODE:= ERRORCODE;
NEWFCB^.FAILURENUMBER:= FAILURENUMBER;
NEWFCB^.TIME1STREPORT:= CURRENTTIME;

/* LOG THE ERROR */
/* SENDMESSAGE TO LOGN, WHERE N IS THE CURRENT PROCESSOR
NUMBER, CONTAINING THE INFORMATION CONTANED IN
THE RECORD POINTED BY NEWFCB */

/* SEE IF THIS ERROR HAS OCCURRED BEFORE */
CURRENTFCB:= FCBQUEUEHEAD;
WHILE
   CURRENTFCB <> NIL AND
   FCBNOTFOUND          /* FUNCTION CALL */
DO
   CURRENTFCB:= CURRENTFCB^.FLINK;

IF
   CURRENTFCB = NIL
THEN
   /* THIS IS THE FIRST TIME THE ERROR HAS OCCURRED
   SO PLACE IT IN THE FCB QUEUE. */
   WITH CURRENTFCB^ DO
      BEGIN
      CURRENTFCB:= NEWFCB;
      RETRYCOUNT:= 0;
```

```
        RECOVERYCOUNT:= 0;
        TIMERECENTREPORT:= TIME1STREPORT;
        FLINK:= FCBQUEUEHEAD;
        BLINK:= NIL;
        FLINK^.BLINK:= CURRENTFCB;
        FCBQUEUEHEAD:= CURRENTFCB
        END

ELSE

    /* THIS ERROR HAS OCCURRED BEFORE SO RESET
    THE VALUE OF "THE MOST RECENT TIME THE
    ERROR HAS OCCURRED" AND RETURN THE SPARE
    FCB TO THE AVAILABLE QUEUE */
    WITH NEWFCB^ DO
        BEGIN
        CURRENTFCB^.TIMERECENTREPORT:=
            TIME1STREPORT;
        FLINK:= FCBAVAILQHEAD;
        BLINK:= NIL;
        FLINK^.BLINK:= NEWFCB;
        FCBAVAILQHEAD:= NEWFCB
        END;
CURRENTFCB^.RETRYCOUNT:=
    CURRENTFCB^.RETRYCOUNT + 1;
IF
    CURRENTFCB^.RETRYCOUNT < RETRYLIMIT
    AND CURRENTFCB^.ERRORCODE <> BADDATAPRODUCED
THEN

    /* ALLOW THE APPLICATIONS PROGRAM TO ATTEMPT THE
        OPERATION AGAIN */

ELSE

    /* TOO MANY ERRORS HAVE OCCURRED SO THE TASK
    MUST BE BAD AND MUST BE RECOVERED */
    WITH CURRENTFCB^ DO
        BEGIN
        RECOVERYCOUNT:= RECOVERYCOUNT + 1;
        RETRYCOUNT:= 0;
        IF
            RECOVERYCOUNT > RECOVERYLIMIT
        THEN
            /* THE PROCESSOR MUST BE BAD */
            /* DISCONNECT THE PROCESSOR */
        ELSE
            TASKRECOVERY (DESTNAME)
        END
END /* OF PROCEDURE FAILUREDETECTED */
```

```
FUNCTION FCBNOTFOUND:   BOOLEAN;

/*THIS FUNCTION IS USED IN PROCEDURE FAILUREDETECTED TO
  DETERMINE IF A NEWLY CONSTRUCTED FCB HAS AN
  IDENTICAL ENTRY ALREADY IN THE QUEUE */
BEGIN
IF
   CURRENTFCB = NIL
THEN
   FCBNOTFOUND: = TRUE
ELSE
   WITH CURRENTFCB^ DO
   IF
      SENDERID = NEWFCB^. SENDERID AND
      ERRORCODE = NEWFCB^. ERRORCODE AND
      DESTNAME = NEWFCB^. DESTNAME AND
      FAILURENUMBER = NEWFCB^. FAILURENUMBER
   THEN
      FCBNOTFOUND: = FALSE
   ELSE
      FCBNOTFOUND: = TRUE
END /* OF PROCEDURE FCBNOTFOUND */
```

```
PROCEDURE TASKRECOVERY (TASK:   TASKID);

/* THIS PROCEDURE SENDS A "RECOVER" MESSAGE TO THE TASK
RECOVERY TASK ON THE PROCESSOR ON WHICH
"TASK" IS RUNNING */

VAR
I:   INTEGER;

BEGIN

/* FIND THE PROCESSOR WHERE THE TASK IS RUNNING */
I:= 1
WHILE
   I <= MAXNOTASKS AND
   GLOBALTABLE[I].TASKID <> TASK.TASKID
DO
I:= I + 1;

IF I > MAXNOTASKS THEN /* NOTIFY LERC OF INTERNAL ERROR */
ELSE
   /* SENDMESSAGE "RECOVER <TASK.TASKID>" TO
      TASKRECOVERYN <N IS GLOBALTABLE[I].PROCESSOR> */

END /* OF PROCEDURE TASKRECOVERY */;
```

```
PROCEDURE LOGN;

/* THIS PROCEDURE MAINTAINS AN ERROR LOG FOR PROCESSOR
   NUMBER N */

VAR
NEWFCB:    ^FCB;

BEGIN
WHILE      /* INFINITE LOOP */
   TRUE
DO
   BEGIN
   /* WAIT UNTIL A MESSAGE ARRIVES*
   /*PLACE THE FCB FROM MESSAGE IN VARIABLE NEWFCB */
   /*WRITE NEWFCB TO DISK */
   END
END /* OF PROCEDURE LOGN */;
```

```
PROCEDURE TASKRECOVERYN;

/* THIS PROCEDURE RECOVERS TASKES RUNNING ON PROCESSOR N.
THIS PROCEDURE MUST ALSO BE RUNNING ON PROCESSOR N */

VAR
TASKID:   INTEGER;

BEGIN
WHILE     /* INFINITE LOOP */
   TRUE
DO
   BEGIN
   /* WAIT FOR A MESSAGE */
   /* PLACE THE TASK ID FOUND IN MESSAGE IN
      VARIABLE TASKID */
   /* RESCHEDULE (TASKID) */
   END
END /* OF PROCEDURE TASKRECOVERYN */;
```

6.6.2.2 LERCUPDATE

```
PROCEDURE LERCUPDATE;

VAR
CURRENTFCB:   ^FCB;
DIAGERRORS:   ARRAY [1..MAXNODIAGERRS] OF BOOLEAN;
PROCESSORFAILURE:   BOOLEAN;
I:   INTEGER;

BEGIN
/* RUN DIAGNOSTICS AND FOR EACH DIAGNOSTIC CHECK SET THE
CORRESPONDING ELEMENT IN DIAGERRORS TO TRUE IF AN
ERROR WAS DETECTED, OTHERWISE SET IT TO FALSE */

/* SEE IF THERE WERE ANY ERRORS DETECTED BY THE
DIAGNOSTIC AND IF THERE WERE, THEN VERIFY THEM */
PROCESSORFAILURE:= FALSE;
FOR I:= 1 TO MAXNODIAGERRS DO
IF
   DIAGERRORS[I]
THEN
   BEGIN
   /* RUN DIAGNOSTIC NUMBER "I" AGAIN */
   IF DIAGERRORS[I] THEN PROCESSORFAILURE:= TRUE
   END;

IF
   PROCESSORFAILURE
THEN

   /* A VERIFIED ERROR HAS BEEN DISCOVERED, THUS THE
   PROCESSOR MUST BE CONSIDERED BAD AND MUST BE
   DISCONNECTED */
   BEGIN
   /* NOTIFY MAINTENANCE OF THE PROBLEMS */
   /* DISCONNECT PROCESSOR */
   END;
/* DELETE THOSE FCB'S WHICH HAVE BEEN AROUND SO LONG
THAT THE ERRORS WERE PROBABLY TRANSIENT AND HAVE
DONE */
CURRENTFCB:= HEADFCBQUEUE;
WHILE
   CURRENTFCB <> NIL
DO
IF
   (CURRENTTIME - CURRENTFCB^.TIMERECENTREPORT) >
   FCBRETAINTIMELIMIT
THEN
   WITH CURRENTFCB^ DO
      BEGIN
      IF BLINK = NIL THEN
         HEADFCBQUEUE:= FLINK;
      IF FLINK = NIL THEN
         TAILFCBQUEUE:= BLINK;
      BLINK^.FLINK:= FLINK;
      FLINK^.BLINK:= BLINK;
```

```
        FLINK:= HEADFCBAVAILQUEUE;
        BLINK:= NIL;
        FLINK^.BLINK:= CURRENTFCB;
        HEADFCBEVAILQUEUE:= CURRENTFCB
        END;
/* RESET WATCHDOG TIMER */
/* SENDMESSAGE "I AM OPERATIONAL" TO ALL MERC'S */

END /* OF PROCEDURE LERCUPDATE */
```

6.6.2.3 MERCUPDATE

```
PROCEDURE MERCUPDATE;

VAR
CONFIG:   INTEGER;


BEGIN
   IF
      /* TIME T1 HAS EXPIRED */
   THEN
      /* COLLECT MESSAGES SENT FROM ALL LERC'S AND SET EACH
         ELEMENT OF PCSTATUS TO "UP" IF A MESSAGE WAS
         RECEIVED, OR TO "DOWN" IF NO MESSAGE WAS RECEIVED */

   IF
      MERCACTIVESTATUS = ACTIVE
   THEN
      COMPUTECONFIGANDSENDIT
   ELSE
      IF
         /* "CONFIG" MESSAGE HAS ARRIVED */
      THEN
         BEGIN
         /* PLACE "CONFIG" IN CURRENTCONFIG */
         /* PLACE "VIRTUALTODISPLAYPCMAP" IN
            VIRTUALTODISPLAYPCMAP */
         IF
            NEWCONFIG <> OLDCONFIG
         THEN
            RECONFIGURE (NEWCONFIG);
         END
      ELSE
         IF
            /* TIME T2PROCESSORNUMBER HAS EXPIRED */
         THEN      /* ACTIVE MERC MUST BE BAD, BECOME ACTIVE */
            BEGIN
            MERCACTIVESTATUS := ACTIVE;
            COMPUTECONFIGANDSENDIT
            END;
END;      /* OF MERCUPDATE */
```

```
PROCEDURE COMPUTECONFIGANDSENDIT;

/* THIS PROCEDURE IS CALLED BY MERCUPDATE TO
DETERMINE THE CURRENT CONFIGURATION AND TO
SEND THE CONFIGURATION MESSAGE TO THE OTHER
MERC'S */

VAR
I:   INTEGER;

BEGIN

/* COMPUTE THE CURRENT CONFIGURATION NUMBER WHICH
CORRESPONDS TO THE NUMBER IN PART 1 OF THE
ROUTING/RECONFIGURATION TABLE */
IF
   PCSTATUS[1] = DOWN
THEN
   CONFIG:= 1
ELSE
   CONFIG:= 0;
IF
   PCSTATUS[2] = DOWN
THEN
   CONFIG := CONFIG + 2;
IF
   PCSTATUS[3] = DOWN
THEN
      CONFIG:= CONFIG + 4;
IF
   CONFIG = 4
THEN
   CONFIG:= 3
ELSE
   IF
      CONFIG = 3
   THEN
      CONFIG:= 4;
CONFIG:= CONFIG + 1;
IF
   PCSTATUS [ VIRTUALTODISPLAYPCMAP[1] ] = DOWN
THEN
   /* THE WATCH SUPERVISOR'S DISPLAY PROCESSOR IS DOWN
   SO ASSIGN THE WATCH SUPERVISOR'S VIRTUAL PROCESSOR
   TO THE FIRST DISPLAY PROCESSOR WHICH IS UP */
   BEGIN
   I:= 2;
   WHILE
      I < MAXNODISPLAYPROCESSORS AND
      PCSTATUS [ VIRTUALTODISPLAYPCMAP[1] ] =
      DOWN
   DO
      I:= I + 1;
   IF
      I < MAXNODISPLAYPROCESSORS
```

```
   THEN
      BEGIN
      VIRTUALTODISPLAYPCMAP[1]:=
      VIRTUALTODISPLAYPCMAP[1];
      /* SEND MESSAGE TO DISPLAY I THAT IT
      IS NOW THE WATCH SUPERVISOR'S STATION */
      END
   ELSE
      /* NOTIFY MAINTENANCE THAT ALL DISPLAY
      PROCESSORS ARE DOWN */
   END;
/* MAP ANY VIRTUAL STATIONS ASSIGNED TO DOWNED
         DISPLAY PROCESSORS TO THE DISPLAY PROCESSOR
         USED BY VIRTUAL STATION 1 (THE WATCH
         SUPERVISOR) */
FOR
   I := 2 TO MAXNODISPLAYPROCESSORS
DO
   IF
      PCSTATUS [VIRTUALTODISPLAYPCMAP [1]] = DOWN
   THEN
      VIRTUALTODISPLAYMAP [I] :=
         VIRTUALTODISPLAYMAP [1];
/* SEND MESSAGE CONTAINING "CONFIG" AND
   "VIRTUALTODISPLAYPCMAP" TO ALL MERCS */

END      /* OF COMPUTECONFIGANDSENDIT */
```

```
PROCEDURE RECONFIGURE (CONFIG: INTEGER);

/* THIS PROCEDURE SCHEDULES AND WOUNDS THE NECESSARY
   TASKS TO FIT THE NEW CONFIGURATION.  WHEN A TASK
   IS WOUNDED, IT IS PLACED ON A QUEUE OF WOUNDED TASKS
   AND IS ALLOWED TO CONTINUE PROCESSING ITS OUTSTANDING
   MESSAGES.  THE WOUND QUEUE IS CHECKED PERIODICALLY, AND
   THOSE TASKS WITH NO MORE OUTSTANDING MESSAGES ARE
   KILLED */

BEGIN
FOR
   I := 1 TO MAXNOTASKS
DO
   WITH CONFIGTABLE [CONFIG] DO
      IF
         PROCESSOR = THISPROCESSORID
         AND /* TASK NAME IS NOT IN CONFIGURATION TABLE */
      THEN
         SCHEDULE (SUSPEND, TASKNAME, STATUS)
      ELSE
         IF
            PROCESSOR = THISPROCESSORID
            AND /* TASK NAME IS IS CONFIGURATION TABLE */
         THEN
            /* WOUND THIS TASK */
END      /* OF RECONFIGURE */
```

7 FAULT TOLERANCE

7.1 ENVIRONMENT

Design Objectives

The objective of fault tolerance is not to build a non-failing system, but to build a system that can continue to operate in the presence of failures. In the presence of a failure, the system must continue to operate, possibly with a reduction in its capabilities, and must recover from the failure quickly enough to prevent the interruption of critical functions. The recovery discussed here is based on the presence of redundant system components and, therefore, depends on the restoration of a failed component to operational capability before another such component can fail.

The failures that must be guarded against are those originating in both the hardware and software whether caused by:

- Implementation errors
- Component failure
- Inaccurate specifications
- Human error.

Both transient as well as solid (permanent) failures will be covered.

Hardware and Software

The symptoms of failures and the characteristics of error handling and recovery procedures are usually quite similar for both hardware and software failures. This fact should be utilized in developing consistent and common procedures to deal with both types of failure.

Processes (**References to processes are equivalent to references to tasks**)

Applications programs are written as a collection of relatively short processes. Processes should be easy to restart, self-checking and verifying, and should have clean interfaces to other processes and data bases utilized.

Redundancy

There is a redundancy of all hardware and software components in the system. The system is sized such that the failure of a hardware or software component will not cause total failure of the system, but merely cause it to operate at perhaps reduced capacity, even though that is not always the case.

Time Constraints

Although this is a real-time management information system, the response time criteria does allow the restart and reexecution of a process. The time constraints also allow human intervention to accomplish system recovery in most instances, particularly those in which it is the most appropriate recovery mechanism to be employed.

Occasional short outages are acceptable; however, lengthy outages are unacceptable, even if they are scheduled.

Cost-Effective Fault Tolerance

Factors contributing to cost-effectiveness of fault tolerance are:

- Modular design and implementation of both hardware and software
- Implementation of fault-tolerance capabilities primarily in software.

Scope of System

One attribute or characteristic of the VTS system that greatly simplifies the entire problem of achieving fault tolerance is that the system is completely defined and tightly-bounded with respect to both the types of procedures that will be run, or services that will be provided, as well as the exact nature of inputs and outputs of these procedures and services. There is a well-defined limit to the range of each reply or response.

Severity of the Results of a Failure

The possible results of a failure are listed below in increasing order of severity.

- Discard results of one process (whether completed or not) - small loss of time, no loss of data
- Loss of one message
- Loss of a hardware or software component with a spare /redundant component available
- Momentary total system outage with automatic recovery quickly executed
- Momentary total system outage with complete reload/ restart automatically executed
- Failure requiring human intervention to restart
- Failure requiring human intervention to diagnose.

7.2 GENERAL APPROACH FOR ACHIEVING FAULT-TOLERANCE

The goal of fault tolerance is that failures, errors and bugs of all types be anticipated. Redundancy of components and isolation or containment of the failure will be used to provide tolerance of faults, whether they are:

- Permanent or transient
- Hardware or software based
- Design or implementation errors in hardware or software (applications or operating system).

Figure 7-1 shows the stages and alternatives in the failure detection and recovery process.

The components of a fault-tolerance system are:

- Redundancy
  - Physical components
  - Processes/Code
  - Data base
  - Consistency checks
  - No real value without error handling and recovery capabilities
- Error Handling
  - Error detection
  - Error containment or isolation
  - Error identification
  - Error reporting
  - Error verification
  - Error logging

FAULT-TOLERANCE SEQUENCE:

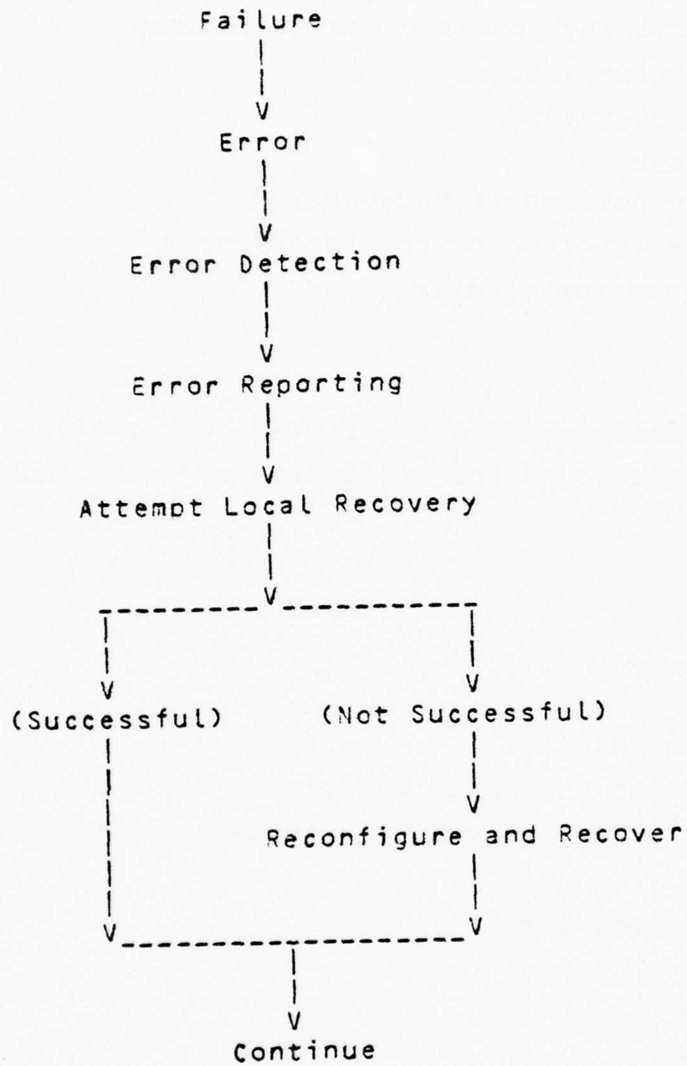


Figure 7-1 Failure Detection & Recovery Process

. Recovery

- Determining possible configurations
- Isolating failure
- Reconfiguration
- Restart
- Safe-shutdown if necessary
- Reconstruction of complete system
- Restoration of data bases

The following failure modes are listed by location of occurrence to provide a background for discussion of fault tolerance.

Hardware

. Main Processor

- Halts (dies)
- Produces erroneous results
- Goes wild - continually transmits nonsense

. Interprocessor Bus

- Open
- Stuck
- Intermittent errors

. Primary Memory

- Module fails
- Word/bit fails
- Intermittent

. Storage Devices

- Device failure
- Bad sectors

. I/O Interface

- Fails (open)
- Stuck-on ("Hogs" bus)
- Intermittent errors

. Watchstanders' I/O Devices

. Display Processor

- Halts (dies)
- Produces erroneous results
- Goes wild - continually transmits nonsense, retaining control of the interprocessor bus or deluging the watchstander with erroneous data.

Tasks (The Execution of Code)

. A server task dies before completing

. The processor in which the task is executing fails and halts

. Task requesting a service dies before the service is completed; therefore, the server has no destination to send its results

- The task cannot be invoked (perhaps due to an inability to identify the task or that the task cannot be found or that the resources necessary to activate the task cannot be made available).

- Task continually transmits messages

- Task destroys executing copy of code and causes total task failure

- A "permanent" lock is placed on a global resource.

Inter-Task Communication

- No destination for message

  - Addressed process not currently active
  - Erroneous message generated

- No answer received

- ITC mechanism flooded or overloaded with messages

Data Base

- Redundant copies of data base become inconsistent (This might be detected when an update is sent to all copies of the data base and it is found that it cannot be performed on one of the copies.)

- An update is given to a data base that it cannot execute

- Internal consistency check of a data base fails
- Information requested from a data base is not present
- The processor on which the data base resides dies
- The data base manager dies or otherwise fails to respond properly

Code

- Internal consistency check of block of code fails
- Program malfunctions and copy of code being utilized is suspected
- Program "bugs" appear
- Design errors are discovered

7.2.1 Redundancy

System redundancy affords both good and bad aspects. The existence of duplicate components of any type always presents the opportunity for disagreement and the subsequent requirement for some independent or third-party method to resolve the disagreement. Redundancy, therefore, has to be treated differently for application to its two main goals: detecting, checking and verifying a failure; or providing a spare component for replacement.

A spare or replacement copy of a resource is of no value in achieving fault-tolerance if it does not function properly itself. Therefore, there must be a continual checking and verifying of the operational status of spares.

One approach, or technique, employed to ensure the "usability" of redundant resources is to maintain all components in an active state rather than a passive or stand-by status. This is accomplished by rotating the designation of primary components or by operating all available resources at reduced load levels.

The operating system must maintain a current map of available hardware and software resources. However, the operating system must always be skeptical of the picture it has of its current environment. All failures must be detected and reported so that failed components may be repaired or replaced. Otherwise, the inherent reliability of the system degrades.

7.2.2 Error Handling

A key design practice should be to make use of expected responses to establish the status of other tasks and to avoid overloading the system with excessive "are you well?" messages. Embedded checks (within tasks and data structures) can often detect failures that are missed by even "good" diagnostic routines. Verification of the operation of various components of the system is an important aspect of error detection. The operating system must check its own status and behavior as well as the application programs. Use of diagnostic and check-out routines should be maximized within the constraints.

Understanding the nature of a failure in a running system usually requires very accurate and complete knowledge of the state of the machine at the instant of the failure - not later.

7.2.3 Recovery

Recovery is an integral part of fault-tolerant operation and, in a real-time system, may be the most difficult objective to achieve. The characteristics of the VTS do alleviate this problem somewhat since the system is basically "self-correcting" with respect to operations on the data base and is tolerant of small divergences while convergence to the "correct state" is taking place.

7.3 HARDWARE FAULT TOLERANCE

The general mechanisms or procedures for achieving hardware fault tolerance are:

- Provide spare resources
- Failure containment
- Ability to selectively shut-down components, utilizing a unique address, even if the component has gone "wild." Also ability to selectively reactivate and reinitialize components.

All of the above goals or requirements are facilitated by highly modular design of the hardware. There should be no master HALT, RESTART, or RESET signals. It is essential that the system permit the physical removal and reinstallation of hardware components without disturbing the remainder of the running system. The removal capability must not be impaired by any failure mode of the component.

In general, the system should automatically recognize its current operational environment that is assumed to be changing on a dynamic basis. It should identify all available resources. This action is executed by the software; however, it is impossible unless the hardware design/logic does support such actions.

A basic conflict does exist between the requirement to protect the currently "working system" and the need to "absorb" new or recovered resources.

Diagnostic and check-out routines can be difficult to design and execute on a large variety of human interface devices. They often require large amounts of human response.

Hardware Fault Tolerance Models

Figures 7-2, 7-3 and 7-4 show the models, or diagrams, for the three system configurations being studied in detail by this project. The component blocks of these diagrams and some major subsystems are discussed here in terms of their characteristics and the fault tolerance features they should include.

7.3.1 Memory

The most common failure will probably be memory parity errors.

Hardware Checks

All of the memory installed on the system should contain Error Checking and Correcting (ECC) hardware. That is, the memory should have built in the memory boards error detection and correction capability at the byte (8-bit) level. This will detect and correct most of the memory error problems that result from hardware failures.

The main problem with ECC memory is the lack of recorded information regarding memory errors that are occurring in a particular portion of memory and being corrected automatically. If the integrity (inherent reliability) of the system is to be maintained, automatically corrected errors must also be reported and recorded so that a failure pattern in a particular memory module can be detected.

There are a number of error patterns that can occur within the bytes that will only be detected (not corrected) by the ECC memory. Some error patterns will not be detected or corrected. Both of these latter classes of errors must be handled by software actions.

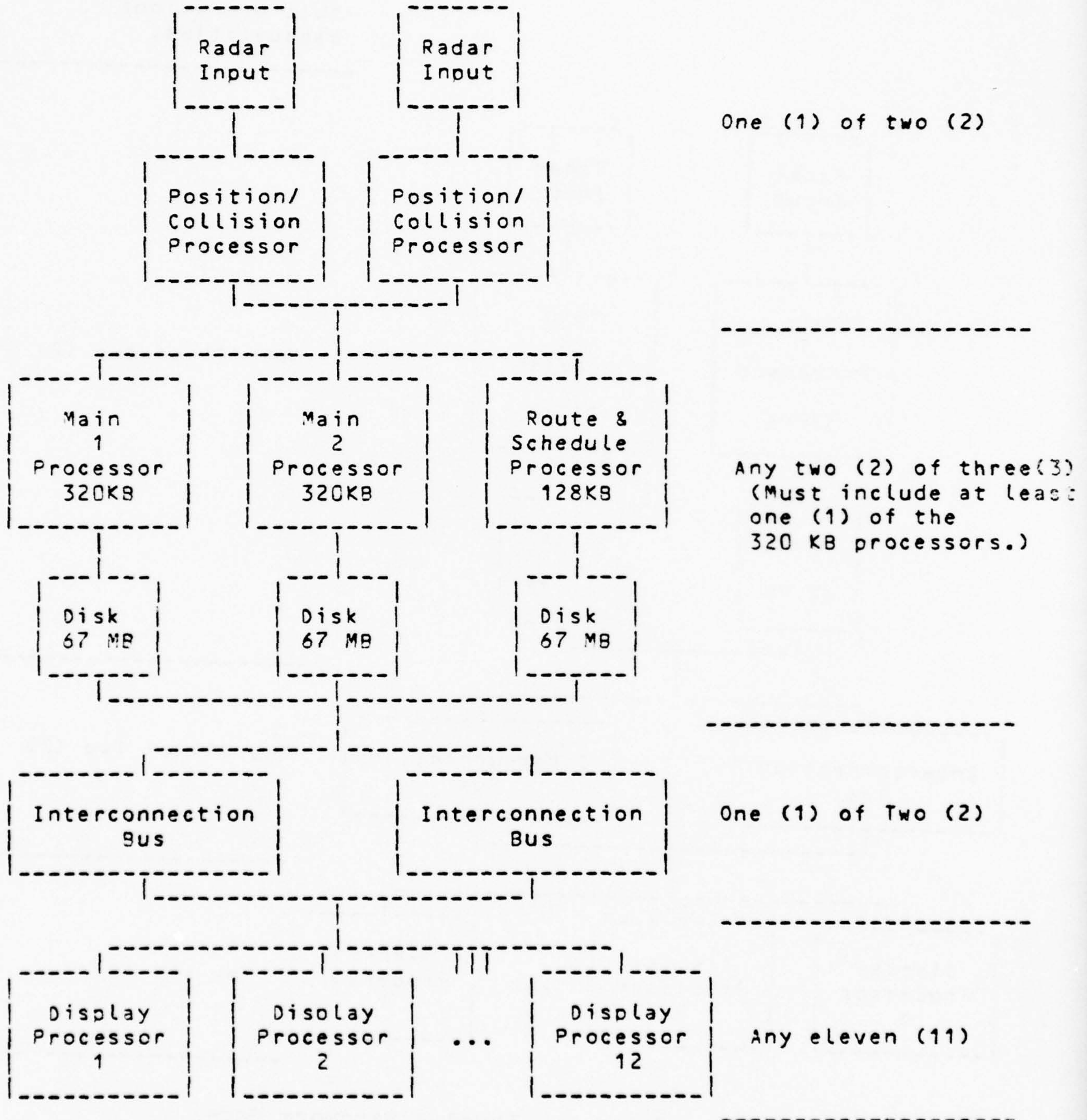


Figure 7-2. Class C - Level 4 Hardware Model

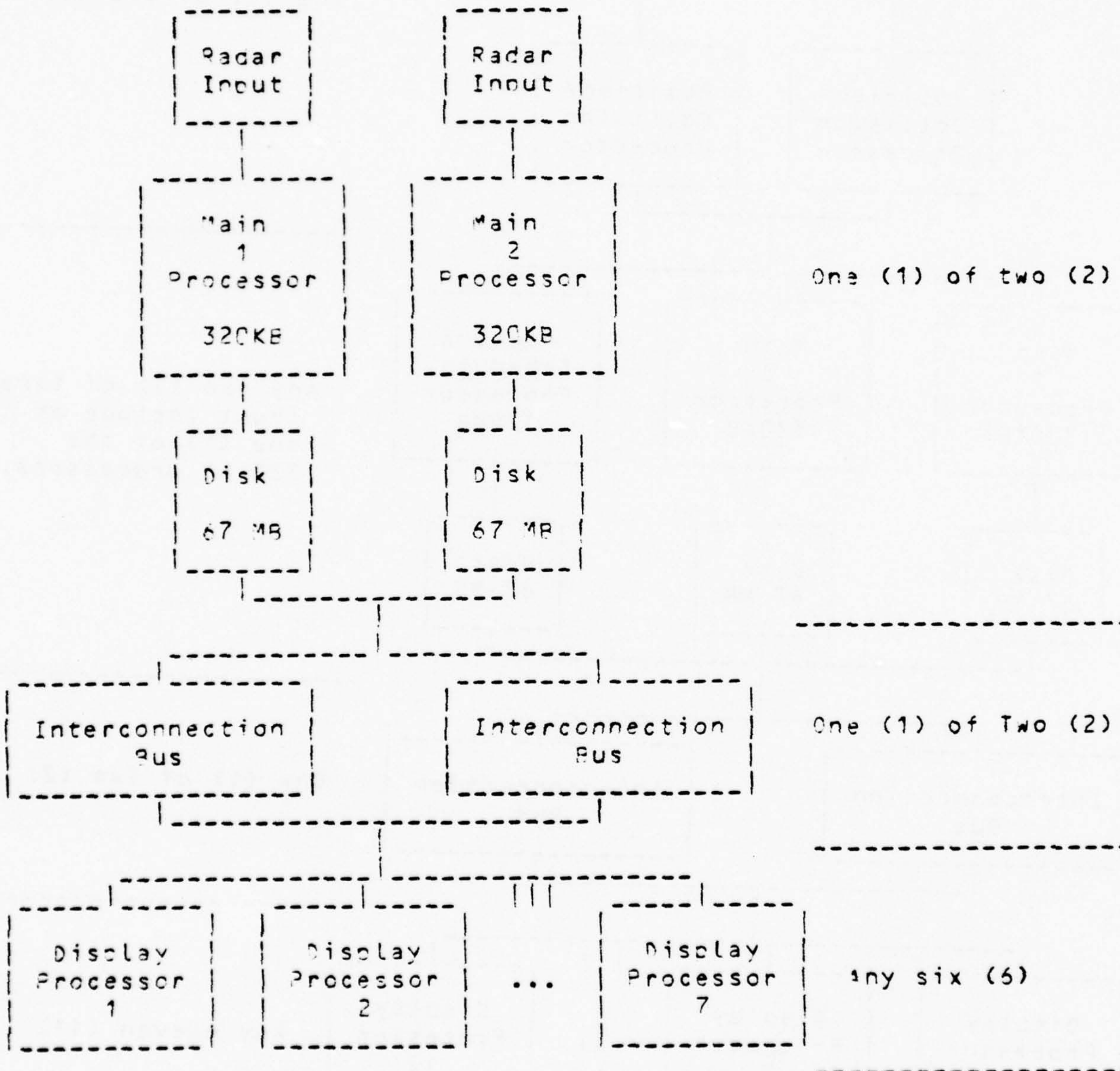


Figure 7-3. Class B - Level 4 Hardware Model

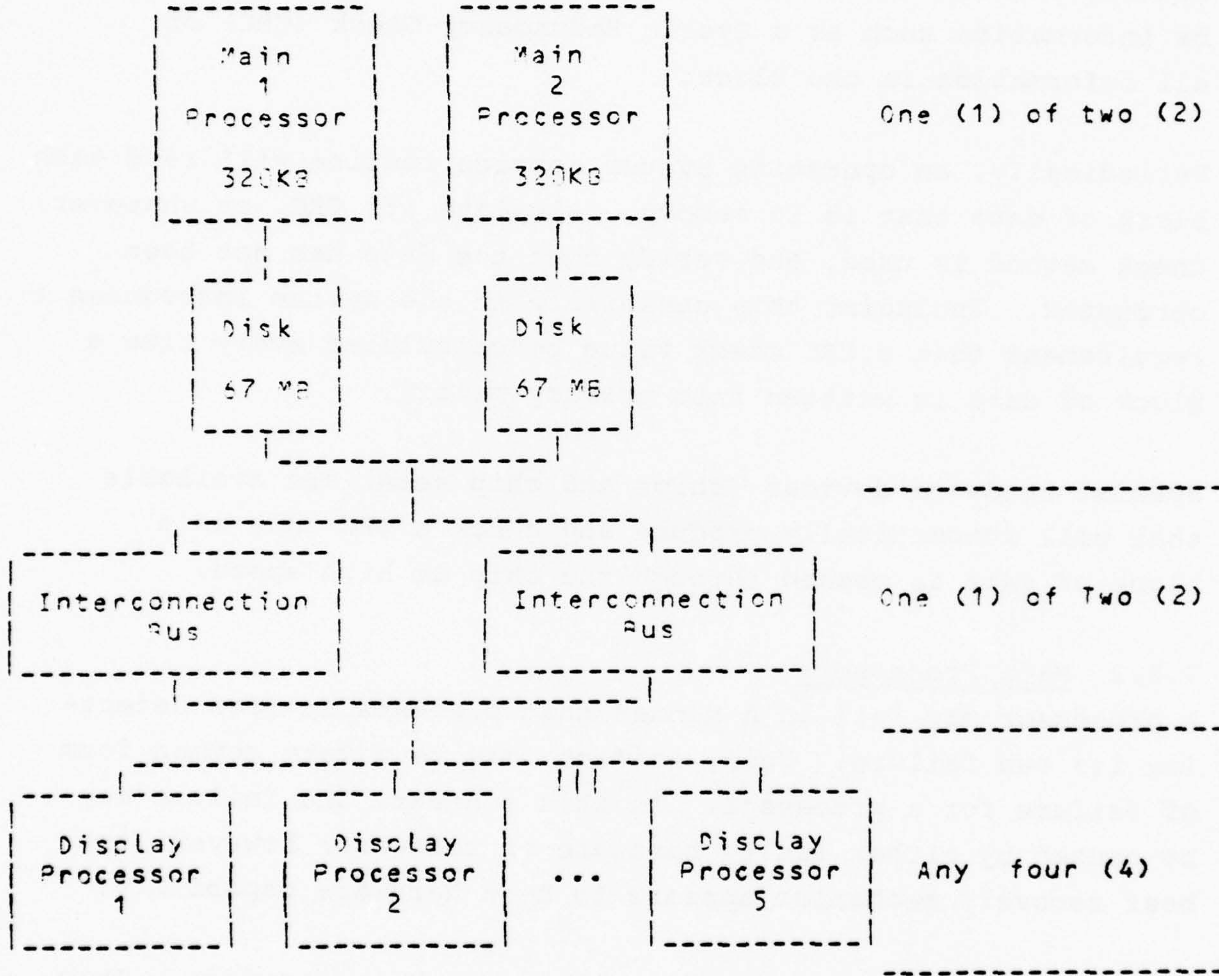


Figure 7-4. Class A - Level 4 Hardware Model

Software Procedures

The integrity of information stored in the memory of the system will be verified by utilizing software check routines. All data will be divided into constant fixed-sized blocks. (The size of these blocks should be no larger than the size of a page). A portion of each block of data will be reserved for control information. The control information included will be information such as a Cyclic Redundancy Check (CRC) of all information in the block.

Periodically, an operating system service routine will read each block of data that is in memory, calculate its CRC, or whatever check method is used, and verify that the data has not been corrupted. Including this capability in the system introduces a requirement that a CRC check value be calculated every time a block of data is written into primary memory.

Special hardware devices (chips and chip sets) are available that will automatically produce and check a CRC code as a block of data is passed through the chip at high speed.

7.3.2 Main Processors

A processor may fail in a manner that prevents it from detecting its own failure. This, in fact, may be a very common form of failure for a processor. In this context, the failure may be caused by either faulty hardware or software; however, the best recovery mechanism appears to be a hardware capability.

There will be a watchdog timer within the bus interface. This timer may be a monostable flip-flop (one-shot), or may be a presettable countdown timer using the 60 Hertz AC line frequency as its input. The time constant of the one-shot or the countdown value will be slightly longer than the running time of the longest single task in the system. Between the execution of each task, the operating system will address a register within the bus interface that will reset the timer to its initial state or countdown value. If the timer is

not reset before this time expires, the timer will disconnect the errant processor from the interprocessor bus.

Consequently, the normal mode for any processor is to be disconnected from the interconnection bus. It is only when a processor has convinced itself that it is healthy and when it continues to execute the continuing series of reset commands on the watchdog timer that it is allowed to remain connected to the interconnection bus.

7.3.3 Discs

The integrity of all information stored on the disc is verified and corrected in the same manner as that used to verify the contents of primary memory. However, it is obvious that the cycle (time between checks) for information stored on disc should be much longer than that utilized for primary memory checks. When it is detected that information stored either in primary memory or on the disc has become corrupted, a recovery operation must be initiated.

If the page in primary memory has not been altered (this information is maintained by the operating system in order to determine which pages have to be written out), a new copy of that page is obtained from the storage device and the integrity of the copy in primary memory is restored. If the page in primary memory has been written to, the only recovery mechanism possible is to signal a failure in the task utilizing that page and for that task to be reinitialized and restarted.

If a block of data on the disc is identified as corrupted, that block must be replaced by a copy from the back-up copy of the data base.

7.3.4 Display Processors

The display processor will play a key role in process management with respect to requests that are entered by the watchstander. The display processor will maintain a record of all service requests entered that have not yet been satisfied as well as information as to which processor is providing the service requested. The display processor will then be in a position to detect the failure of the server task or server processor by a prolonged silence. The display processor would then convey this information to the Reconfiguration Controller and reinitiate a request for the services in a normal manner as soon as the system is again declared operational by the reconfiguration master. There are a number of situations in which two processes are intercommunicating and if one of these dies it could leave the other one in a "hung" condition. This situation will be handled by:

- Requiring a positive acknowledgement task-to-task for instances where such information is needed

- Making use of watchdog timers to ensure that a task that is waiting for an answer does not wait too long in the event that the answerer has died.

To illustrate, the state diagram for the process DELETE-VESSEL (see Figure 7-5) will be used. At state 4 the vessel has been properly identified, the record has been displayed, and the watchstander has been queried "okay to delete." There is no need for the display processor to acknowledge the receipt of the record to be displayed, nor to acknowledge receipt of the question. If the question is not answered in a suitable length of time, the delete record process will make the assumption that the watchstander station is inoperative, it will recover from whatever condition it may have reached by being half-way through a delete situation and notify the error reporting center that it suspects that that watchstander station is inoperative.

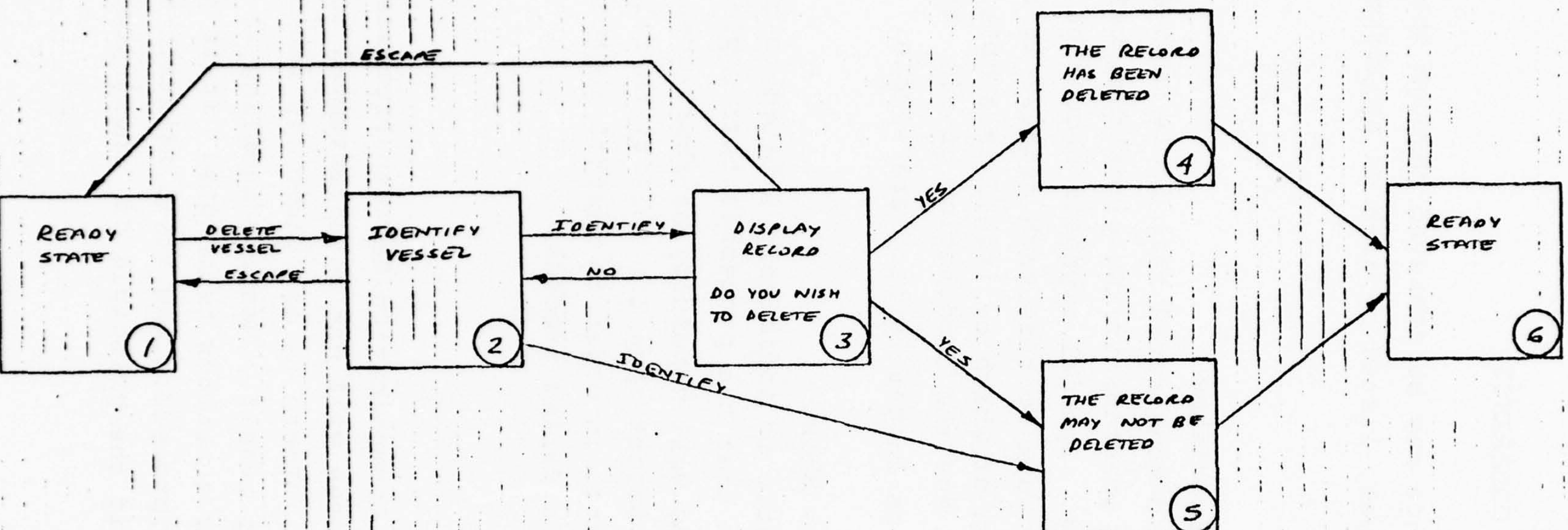


Figure 7-5 Delete Vessel

7.4 SOFTWARE FAULT TOLERANCE

Software fault tolerance may be achieved through the use of the following techniques, listed by area of application of the technique.

Program Design

- Utilize Simple Algorithms
  It has been well demonstrated that it is much easier to prepare reliable programs if the algorithms being implemented are simple. Special emphasis should be paid to partitioning the work so that each process is decomposed into pieces, each of which by itself is simple and straightforward, and the combination of all, and the interrelationships in particular, are simple.

- Keep Each Process Short
  This characteristic is intimately related to the preceding one in that the processes that implement simple algorithms will hopefully be short ones. It also has a good impact upon the ability to roll back the flow of control in order to restart a process and correct for a failed process.

Message Handler

- Utilize Passive Communication Between Processes
  Executing processes will not be interrupted to accept normal messages. There is no "panic" in the message handling process itself. Message handling is implemented utilizing a mailbox technique. It is possible to utilize this procedure and still maintain satisfactory response times if the principle of "short processes" is adhered to.

. Utilize Watchdog Timers
A watchdog timer will be set by the message handler each time a message is sent. If that message is not acknowledged before the runout of the watchdog timer, the message handler must take corrective actions.

Data Base

The data base management system and the data base itself are probably the "weak links" in the fault tolerance system due to the much lower state-of-the-art in data base software. The software is usually quite large and it is often very difficult to subdivide the functions into short processes from which recovery is easy.

Because of the very nature of its organization, use, and modification, it is especially difficult (and time-consuming) to recover from any but the most trivial of failures in the data base.

7.5 ERROR DETECTION AND REPORTING

The following considerations apply to the detection and reporting of all types of errors.

- The processor will be disconnected by the watchdog timer if it hangs up or enters an endless loop (e.g., failures that are "non-detectable" by any other means).

- The LERC periodically runs check-out routines against the hardware and control software.

- Every process is designed and implemented to perform local (i.e., process) failure and error detection.

- All errors or failures detected are reported to the LERC which logs them. Processors without discs will have their log maintained by a support process on a main processor.

- All errors reported are verified by the LERC. (This is to avoid recovery on transient errors.)

  - For diagnostic errors, verification involves rerunning the diagnostic.

  - For process errors, verification involves the following:

    - Roll-back and reexecution. If failure reappears, then

    - KILL and RESCHEDULE. If failure reappears, then

- Attempt recovery

. For verified errors, local recovery is initiated

. Periodically, the error log is examined for failure patterns that have not resulted in solid, verified errors

7.5.1 Process Failure

Local Detection

A process can attempt to detect its own failure by performing various self checks. For example, a process can verify the validity of its computed data. If a process determines that it has failed, then, either the computational portion or the error detection portion of the process has failed. Either of these cases indicates a failure in the process. Thus, the process must inform the LERC that a recovery of that process is required.

Remote Detection

A failure in a given process can be detected by other processes which are interacting with the given process. Three basic errors can be detected:

. Bad data received
. Message not answered
. Unable to INSTALL or SCHEDULE a process

The LERC on the processor on which the error detecting process is running is the one to which the error is initially reported.

7.5.2 Main Processor Failure

Local Detection

The LERC is responsible for making a local determination of the health of a processor. It makes this determination based upon two sources of data. The first source is the LERC diagnostic routines which are executed periodically under control of the LERCUPDATE routine. These routines are designed to detect errors in both hardware and software. The second source of data are error reports received from application processes. The LERC logs these and verifies them in an attempt to distinguish transient errors from permanent errors.

Remote Detection

There are three ways that a failure in a processor can be remotely detected. First, if the operating system of the processor fails to reset the watchdog timer, the processor will be automatically disconnected. Second, the LERCs on other processors will keep a log of the inability of their processes to communicate with processes on another processor. Third, all MERCs will log "I AM OPERATIONAL" messages from all main processors. Failure of a main processor to send this message will result in that processor being disconnected by MERCUPDATE.

7.5.3 Position/Collision Processor Failure

Local Detection

The local detection of a processor failure is the responsibility of the LERC. The functions performed by the LERC in the Position/ Collision Processor are the same as those performed by the LERC in the main processor.

Remote Detection

There are two ways that a failure in a Position/Collision Processor can be remotely detected. First, if the operating system of the processor fails to reset the watchdog timer, the processor will be automatically disconnected. In addition, the Position/Collision Processors possess MERCs which are distinct from those on the Main processors. These MERCs will be referred to as PCMERCs. The PCMERCs communicate with each other to determine which processor is active. The LERCs on each processor will regularly send "I AM OPERATIONAL" messages to the other processor's PCMERC. If a PCMERC does not receive an "I AM OPERATIONAL" message after a fixed period of time, the PCMERC will attempt to help its processor become the active Position/Collision Processor .

7.5.4 Display Processor Failure

Local Detection

The local detection facilities are the same as those found on the Position/Collision Processor with the important additional feature that the human operator is also an integral part of the error detection process.

Remote Detection

There are two ways a failure of a Display Processor can be remotely detected. First, if the operating system of the Display Processor does not reset the watchdog timer after a fixed period of time, the Display Processor will be automatically disconnected. In addition, the LERC on a Display Processor is required to periodically send an "I AM OPERATIONAL" message to all MERCs. If the MERC acting as reconfiguration controller does not receive this message after a fixed period of time, it will conclude that the Display Processor has failed and modify the configuration of the system accordingly.

7.5.5 Bus Failure

Local Detection

Bus interface unit hardware performs local detection of bus failure.

Remote Detection

Bus failure remotely detected as failures in message handler.

7.5.6 Data Base Failure

Local Detection

Local detection of data base failure is performed by DBMS.

Remote Detection

Failure in the data base can be detected remotely by a process which has discovered inconsistencies in the data it has obtained from the data base. This process will inform the LERC on its processor that it has experienced a "Bad Data Received" error. The DBMS responds to the same sequence of retry and recovery messages as is used for process failures.

7.5.7 Other Hardware Failures

The LERC on the processor to which the hardware is attached performs periodically diagnostic routines which will check for any failures in the attached hardware.

7.6 ERROR RECOVERY

The following assumptions are made concerning recovery from detected errors:

- Local recovery is attempted for all verified errors except incapacitating hardware errors. (Incapacitating failures include almost all solid hardware errors except errors in an I/O device or a memory module which can be logically removed from active use).

- For incapacitating hardware errors, the LERC executes a system disconnect and issues a failure notice to the "maintenance" console, if one exists.

- For non-incapacitating hardware errors, the LERC immediately issues a failure notice to the "maintenance" console and then returns control to the Executive.

- To recover from a verified process failure, attempt each of the following in the order given.

    - Roll-back and retry process
    - KILL and RESCHEDULE the process
    - INSTALL a new image, i.e.:

        - Identify all TCBs associated with the ICB of the failed process

        - KILL all of these TCBs preserving enough information to restart the good tasks where they were interrupted

- REMOVE the old image
- INSTALL a new image
- SCHEDULE the new tasks and restore the good tasks which were interrupted.

- If none of the techniques above succeed, processor reconfiguration is required.

7.6.1 Process Failure

A process can be recovered in three ways. First, a process can be rolled back to its last message and have execution reinitiated. If this fails after a given number of attempts, a second more radical recovery technique can be attempted. In this case, the process is KILLED and RESCHEDULED. Failure of this recovery attempt, results in the last recovery attempt. This involves KILLing all processes executing from the same image as the process which is in trouble, INSTALLing a new image, restoring all processes except the troubled process to their states prior to the KILLing operation, and RESCHEDULing the troubled process. If all of these recovery operations fail, the LERC must conclude that the processor is bad and disconnect the processor.

7.6.2 Main Processor Failure

As mentioned above the LERC has the capability to disconnect the processor if it determines that it is bad. In addition, if the operating system fails to reset the watchdog timer after a given interval, the processor will be automatically disconnected. The task of reconfiguring the system is performed by one of the MERCs. All the MERCs record the "I AM OPERATIONAL" messages from all processors. One MERC acting as Reconfiguration Controller will send out a configuration number corresponding to the configuration indicated by the reported messages. This MERC can also disconnect any bad processors utilizing the bus interface unit of that processor. Each MERC examines the configuration number received from the MERC acting as Reconfiguration Controller. If

this differs from the previously obtained configuration number, a process is started which will KILL and SCHEDULE the proper processes to reflect the new configuration.

7.6.3 Position/Collision Processor Failure

The LERC on each Position/Collision Processor has the capability to disconnect the processor if it determines that the processor is bad. The processor will also be disconnected if the operating system fails to reset the watchdog timer within a given time interval. Both processors contain PCMERCs which accept "I AM OPERATIONAL" messages from the other processor. If the active processor fails to send this message during a given time interval, the PCMERC on the inactive processor will initiate a takeover of the active duties by the processor on which it is running. This process consists of disconnecting the other Position/Collision Processor and activating the transmission facilities on the good Position/Collision Processor.

7.6.4 Display Processor Failure

The Display Processors can be disconnected by the LERC and by failures to reset the watchdog timer as described for the other processes. The reconfiguration task is performed by the MERC acting as the Reconfiguration Controller. The MERC in addition to sending the current configuration number to all other MERCs sends PART III of the Routing/Reconfiguration Table. The Reconfiguration Controller MERC also periodically sends PART II of the Routing/Reconfiguration Table to all MERCs.

7.6.5 Bus Failure

The failed bus is disconnected from all processors and the alternate bus is utilized for all traffic.

7.6.6 Data Base Failure

Recovery is the responsibility of the DBMS.

7.6.7 Other Hardware Failures

The LERC upon verifying a failure on a piece of attached hardware will be responsible for "disconnecting" the hardware and notifying maintenance.

7.7 RECONFIGURATION

All tasks are INSTALLed on their respective processors at all times. When spare processors are available, the READY tasks are allocated to the available processors in a manner so as to balance the load while maintaining "families and hierarchies" or "groups" on the same machine in order to minimize interprocessor communication load. Current configuration information is maintained in the Routing/Reconfiguration Table.

The only reconfiguration controlled automatically by the system is the deletion and addition of "main" processors. Reconfiguration and recovery after reconfiguration with respect to display processors is controlled by the watchstanders themselves except that any Virtual Display Stations assigned to a failed display processor will be automatically switched to the display processor currently servicing the Watch Supervisor. Figure 7-6 shows the possible failure and recovery states for the Class C, Level 4 system.

7.7.1 Current Configuration Information

There are three system tables which define the current operating environment of VTS/OS. The first is the Configuration Table, shown in Figure 7-7. This table has two indexes: as shown in the figure, the column index represents the current configuration, indicating which of the main and position/collision processors are currently in normal operation; the row index corresponds to the names of all tasks which can receive messages via Inter-task Communications. The entries in the table contain the processor in which each task resides for that configuration. The Router functions use this table to determine whether or not a receiving task is local to the sender's processor, and thus will automatically respond to a reconfiguration when a new column index is received from the controlling MERC.

FINITE STATE DIAGRAM

DEPICTING ALLOWABLE CHANGES IN CONFIGURATION

THAT DO NOT RESULT IN DEGRADED PERFORMANCE

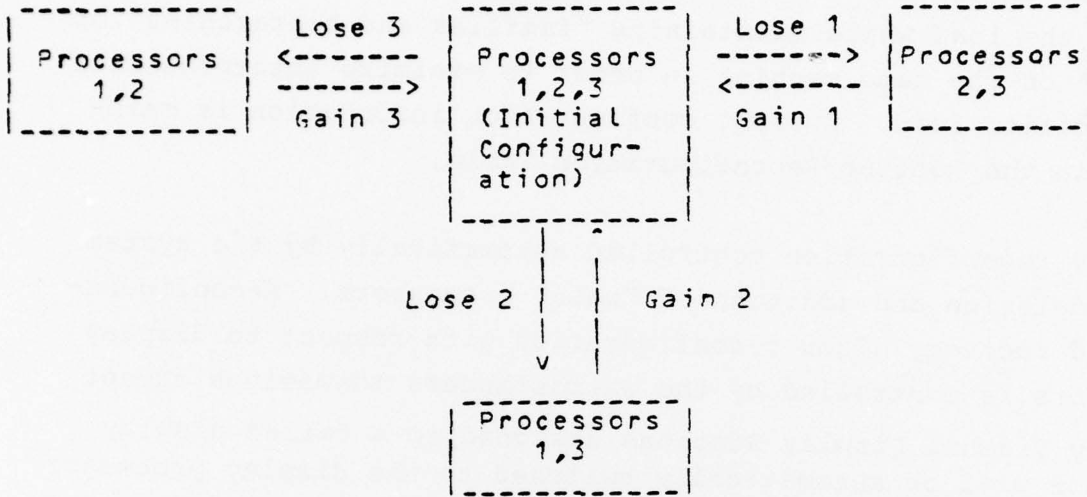


Figure 7-6. Class C, Level 4 System

	<---- Provide Full ---> Capability				<--- Reduced ---> Capability		
Processors Operating	ALL	1&2	1&R/S	2&R/S	1	2	R/S
Configuration ID	1	2	3	4	5	6	7
T							
a							
s							
k							
I							
D							
s							

Figure 7-7. Configuration Table

Associated with this Configuration Table is a table of message input queue addresses for use by the Router and Dispatcher, shown in Figure 7-8. For all tasks which may be active in a particular processor, its message queue address table entry points to the incoming message queue for that task. Thus, even during reconfiguration of the system or reinstallation of a task, any messages arriving for that task will be placed on the proper queue to be removed when that task is ready to receive the message.

The third table, shown in Figure 7-9, represents a correspondence between physical processor addresses and logical identification of Watchstander Station tasks which must receive messages. In order to avoid having the applications programs know the physical processor responsible for waterway cells affected by an alert, this table provides a correspondence between physical CPU addresses and logical Watchstander Station addresses. Stations are assigned logical addresses when the system is initialized. When a Display Processor fails, it may be replaced by a spare or have its functions reassigned to the Watch Supervisor by altering only the physical processor address entry in this table.

```
-----------------------------
T          |-----|
           |     |
a          |-----|
           |     |  Pointers to message QCB's
s          |-----|
           |     |
k          |-----|
           |     |
           |-----|
           |     |
I          |-----|
           |     |
n          |-----|
           |     |
s          |-----|
           |     |
-----------|-----|
```

Figure 7-8. Message Input Queue Table

Virtual Station		Physical Display Processor
	1	
	2	
	3	
	4	
	5	
	6	
	7	
	8	
	9	
	10	
	11	
	12	

Figure 7-9. Virtual Display Station Table

8 RECOMMENDATIONS AND CONCLUSIONS

In the course of this design effort, ICC has found that performing the detailed design of an operating system in the absence of hardware specifications and development software specifications has both significant advantages and significant disadvantages.

One advantage is that it is possible to relegate the actual details of manipulating the hardware to very low-level routines, which tends to encourage top-down design. A more significant advantage lies in the fact that the design process is not immediately constrained by the lack of features which are discovered to be helpful, nor by the presence of others which are discovered to be either awkward to work with or awkward to work around. Useful features can be assumed to exist; awkward ones can be assumed not to exist.

The disadvantages lie in the very same freedom, and the possibility of abusing this freedom. Constraints external to the design process may force selection of hardware and software lacking one or more of the helpful features upon whose existence much of the design is predicated, or conversely possessing one or more of the awkward features whose absence is critical to the design. Furthermore, since it would be foolish to assume the existence of features which are helpful but rarely found, the design will not take advantage of such features if, in fact, they are actually available after the selection of hardware and software has been made.

In the area of hardware selection, ICC has identified some constraints on the final choice which are of varying criticality. One highly critical constraint, upon which a great deal of the design is based, is the requirement that the selected processor use at least 16 mapping registers to span a single task's address space.

ICC believes that the design in this report will prove to be quite unworkable if an attempt is made to implement it on a processor with fewer registers.

A related constraint, but of much lower criticality, is that it would be useful in the extreme if the memory protection hardware allowed three levels of access to memory (in descending order of privilege: write, read, and execute). An acceptable, although less valuable, alternative would be to have write-protection alone. However, the design is not predicated upon the assumption that any form of memory protection is provided, although a higher degree of protection will obviously afford greater safety of operation.

In the area of manufacturer-provided development software, ICC has made some assumptions concerning the structure and format of program object files. These are detailed in Section 6.2.1.1, the INSTALL routine. However, these assumptions are not central to much of the design of VTS/OS, and have all been encapsulated within the INSTALL algorithm.

Significantly greater constraints have become apparent as a result of ICC's choice of PASCAL as the basis for a Program Design Language. These constraints, which render it virtually impossible to implement the entire operating system in pure PASCAL (as defined in the PASCAL User Manual and Report, by Kathleen Jensen and Niklaus Wirth), can all be traced back to the fact that PASCAL was initially designed as a teaching language. One goal of the PASCAL design was to have it be as independent as possible of any features of processor architecture. As a result, the PASCAL programmer is hampered by an inability to perform any but the simplest manipulations of data structure elements, since these are "hidden" by PASCAL. The strong variable typing in PASCAL, which is highly appropriate for an applications program, proves to be a hindrance in an operating system routine.

The inability to manipulate memory addresses, and to access memory indirectly through these addresses, is also a hindrance. Perhaps the most awkward constraint concerns the definition of the Event Control Block. Although the ability to define and use these ECB's in flexible and varying ways is crucial to the design of VTS/OS, PASCAL makes this procedure extremely tedious, and occasionally not feasible.

The development operating system should be evaluated both in terms of its capabilities for compiling and linking together programs and subprograms for the operating system and applications, and in terms of the compatibility of its disc directory structure with that outlined for VTS/OS in Section 4.5. The disc directory contents shown in this report are a minimal subset necessary for VTS/OS; the directory structure of VTS/OS may be modified to conform to the development system, but the indicated directory information must exist in some format to support the VTS/OS file management functions.

DAT
FILM